# Описание проекта

Написание статей и добавление к ним комментариев.

# API

1. /register - Регистрация пользователя с ролью ADMIN
2. /register - Регистрация пользователя с ролью USER
3. /login – любой пользователь может залогиниться, если он до этого зарегистрировался.
4. /getAllUsers – только для администратора
5. /getUser - администратор и суперадминистратор имеет доступ кол всем записям, пользователь – только к своей записи
6. /updateUser – изменить пользователя может только сам пользователь, независимо от того, какая у него роль. Кроме этого, супер администратор может менять любую запись
7. /deleteUser - администратор имеет доступ кол всем записям, пользователь – только к своей записи!!! Если пользователь, которого хотят удалить имеет роль ADMIN, то удалить его может только SUPER\_ADMINISTRATOR

!!!SUPER ADMINISTRATOR создается «хардкодно» при загрузке приложения.

# Логика по которой будет построено приложение

1. Роли – всего будет создано 3 роли
   1. ROLE\_USER
   2. ROLE\_ADMIN
   3. ROLE\_SUPERADMIN
2. Права – всего будет создано 3 права
   1. READ
   2. WRITE
   3. DELETE

## Таблица

|  |  |  |
| --- | --- | --- |
| USER | ADMINISTRATOR | SUPERADMINISTRATOR |
| Может смотреть свой профиль | Может смотреть любой профиль | Может смотреть любой профиль |
| Не может удалять профиль | Может удалять свой профиль | Может удалять любой профиль |
| Может менять свой профиль | Может менять свой профиль | Может менять любой профиль |

# Создание пользователя (без прав и ролей)

1. Создаем приложение security-template
2. Добавляем [pom](#_POM)
3. [Подключаем базу данных(H2, которая будет хранить данные в файл, что бы не терять данные при каждой перезагрузке)](#_Настройки_подключения_к)
4. [Создание пользователя](#_Создаем_пользователя)
5. [Добавление пользователя](#_Добавление_пользователя)
6. [Проверка функциональности добавления пользователя](#_Проверка_функциональности_добавлени)

## POM

1. <dependencies>  
    <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
    </dependency>  
    <dependency>  
    <groupId>org.projectlombok</groupId>  
    <artifactId>lombok</artifactId>  
    <optional>true</optional>  
    </dependency>  
    <!-- modelMapper-->  
    <dependency>  
    <groupId>org.modelmapper</groupId>  
    <artifactId>modelmapper</artifactId>  
    <version>2.3.7</version>  
    <!-- end of modelMapper-->  
    </dependency>  
    <!-- Validator-->  
    <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-validation</artifactId>  
    </dependency>  
    <!-- spring security-->  
   <!-- <dependency>-->  
   <!-- <groupId>org.springframework.boot</groupId>-->  
   <!-- <artifactId>spring-boot-starter-security</artifactId>-->  
   <!-- </dependency>-->  
    <!-- end of spring security-->  
    <dependency>  
    <groupId>io.jsonwebtoken</groupId>  
    <artifactId>jjwt</artifactId>  
    <version>0.9.0</version>  
    </dependency>  
    <dependency>  
    <groupId>com.h2database</groupId>  
    <artifactId>h2</artifactId>  
    <scope>runtime</scope>  
    </dependency>  
    <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-data-jpa</artifactId>  
    </dependency>  
    <!--end of jwwt-->  
    <dependency>  
    <groupId>jakarta.xml.bind</groupId>  
    <artifactId>jakarta.xml.bind-api</artifactId>  
    <version>2.3.2</version>  
    </dependency>  
    <!-- Runtime, com.sun.xml.bind module -->  
    <dependency>  
    <groupId>org.glassfish.jaxb</groupId>  
    <artifactId>jaxb-runtime</artifactId>  
    <version>2.3.2</version>  
    </dependency>  
    </dependencies>  
     
    <build>  
    <plugins>  
    <plugin>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-maven-plugin</artifactId>  
    <configuration>  
    <excludes>  
    <exclude>  
    <groupId>org.projectlombok</groupId>  
    <artifactId>lombok</artifactId>  
    </exclude>  
    </excludes>  
    </configuration>  
    </plugin>  
    </plugins>  
    </build>

## Настройки подключения к базе данных –

spring.datasource.url=jdbc:h2:file:./users;DB\_CLOSE\_DELAY=-1  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.username=user  
spring.datasource.password=password  
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect  
spring.h2.console.enabled=true  
spring.h2.console.path=/h2-console  
spring.h2.console.settings.trace=false  
spring.jpa.open-in-view = true  
spring.hibernate.hbm2ddl.auto=update  
spring.jpa.hibernate.ddl-auto=create

Проверяем базу данных

<http://localhost:8080/h2-console/>

Изображение выглядит как текст

Автоматически созданное описание

База создана!!!

## Создаем пользователя

1. Создаем общий класс, в котором будут храниться все общие поля. В данном случае – это id.

Изображение выглядит как текст

Автоматически созданное описание

!!! что бы следующие классы наследники корректно прочитали все поля – классу BaseEntity необходима аннотация @MappedSuperclass

1. Строим класс Users

У пользователя должны быть поля:

* Uuid (unique)
* Email
* FirstName
* SecondName
* Password (закодированный)
* confirmPassword (пока мы просто хардкодно пропишем, что пользователь подтвердил свой электронный адресс.)

@Entity  
@Table  
@AllArgsConstructor  
@NoArgsConstructor  
@Getter  
@Setter  
@Builder  
@ToString  
public class User extends BaseEntity{  
 @Column(nullable = false)  
 private String uuidUser;  
  
 @Column(nullable = false)  
 private String firstName;  
  
 private String secondName;  
  
 @Column(nullable = false)  
 private String userEmail;  
  
 @Column(nullable = false)  
 private String password;  
 */\*\* noe all users has confirmed email. We dont need this functionality  
 \* in this project  
 \*/* @Column(nullable = false)  
 private Boolean confirmEmail = true;  
}

PS: все ролик и права мы добавим пользователям позже. После того, как у нас заработает аутентификация и авторизация.

1. Запускаем –проверяем в базе данных –

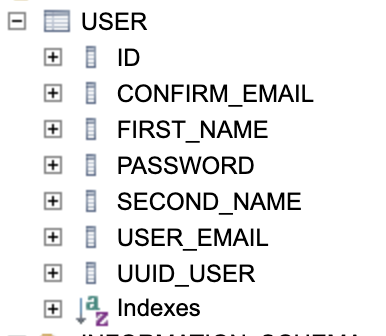


Таблица со всеми нужными нам полями создана.

## Добавление пользователя

1. Пишем контроллер

@RestController  
@RequestMapping("/users/v1")  
public class UserController {  
  
 final  
 UserServiceImpl userService;  
  
 public UserController(UserServiceImpl userService) {  
 this.userService = userService;  
 }  
  
@PostMapping()  
public ResponseEntity register(@RequestBody UserDto userDto){  
 return new ResponseEntity(  
 userService.createUser(userDto), HttpStatus.*CREATED*);  
}  
}

1. Пишем класс UserDto. По логике нашего приложения, пользователь может зарегистрироваться в роли Администратора в том случае, если к него есть определенный секретный код. Роли мы пока не ставим, но подготовиться к этому процессу можем. Поэтому в UserDto кроме полей для создания UserEntity будет еще одно необязательное поле – code

@AllArgsConstructor  
@NoArgsConstructor  
@Getter  
@Setter  
@Builder  
@ToString  
public class UserDto {  
 private String firstName;  
 private String secondName;  
 private String userEmail;  
 private String password;  
 private String code = null;  
}

1. Пишем класс UserServiceImpl, в котором, в методе createUser и будет прописана логика по созданию пользователя.

Изображение выглядит как текст

Автоматически созданное описание

Кроме этого, был добавлен конструктор в UserServiceImpl, был добавлен Bean в класс BeanConfig (для того, что бы кодировать пароль), был добавлен класс UserRepository

Изображение выглядит как текст

Автоматически созданное описание

## Проверка функциональности добавления пользователя

JSON ->

{

"userEmail":"user@mail.com",

"password":"password",

"firstName":"Users First Name",

"secondName":"Users First Name"

}

Отправляем запрос – получаем Изображение выглядит как текст

Автоматически созданное описание

Проверяем базу - 

# Подключаем Security. Добавляем авторизацию и аутентификацию.

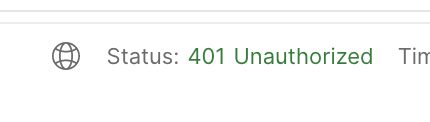
1. [ВключаемSpringSecurity](#_ВключаемSpringSecurity)
2. [Конфигурация SpringSecurity](#_Конфигурация_SpringSecurity)
3. [Добавляем аутентификацию](#_Добавляем_аутентификацию)
4. [Добавляем авторизацию](#_Добавляем_авторизацию)

## ВключаемSpringSecurity

Добавляем в Pom.xml зависимость

1. <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-security</artifactId>  
   </dependency>

Пробуем создать нового пользователя



Получаем 401 ошибку. Она говорит о том, что для запроса необходимо пройти аутентификацию.

Это не логично, пользователь, который еще не зарегистрировался не может пройти аутентификацию.

Что бы исправить это – необходимо конфигурировать Spring Security

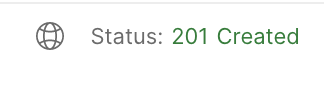
## Конфигурация SpringSecurity

Добавляем класс WebSwcurity

@EnableWebSecurity  
public class WebSecurity extends WebSecurityConfigurerAdapter {  
  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http.csrf().disable()  
 .authorizeRequests()  
 */\*\*  
 \* api for registration user permit all  
 \*/* .antMatchers(HttpMethod.*POST*, SecurityConstants.*SIGN\_UP\_URL*).permitAll()

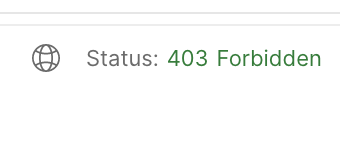
*/\*\*  
 \* all others should have authenticated  
 \*/* .anyRequest().authenticated();  
 */\*\*  
 \* clean headers after all request  
 \*/* http.headers().frameOptions().disable();  
 }  
}

1. Проверяем в postman -> получаем



1. Создаем тестовый метод в контроллере и пробуем его вызвать

@PostMapping("/test")  
public String test(){  
 return "I am work: Test method";  
}



Получаем 403 ошибку, которая говорит, что доступ к методу разрешен только тем, кто прошел аутентификацию.

Пробуем попасть в базу данных – получаем ту же ошибку.

Что бы ее исправить, необходимо дать доступ к этой точке всем пользователям.

Добавляем в метод configure класс WebSecurityConfig

*/\*\*  
 \* access to DB permit all  
 \*/*.antMatchers(SecurityConstants.*ACCESS\_TO\_DB*).permitAll()

Класс SecurityConstants

public class SecurityConstants {  
 public static final String *SIGN\_UP\_URL* = "/users/v1";  
 public static final String *ACCESS\_TO\_DB* = "/h2-console/\*\*";  
}

Проверяем – доступ к базе данных появился!

## Добавляем аутентификацию

### Идея токена

Идея в том, что при логине пользователь получит свой токен, по которому его будет определять юзер сервис и давать ему доступ ко всему

### Создание токена

Создаем класс LoginRequestDto. Именно объект этого класса должен отправить пользователь при «логине»

public class LoginRequestDto {  
 private String userEmail;  
 private String password;  
}

Теперь пришло время создавать фильтр для аутентификации. Фильтр, который в конечном счете будет проверять – что это за пользователь, есть ли он в БД и если все ОК – будет генерироваться JWT токен, по которым пользователь потом будет проходить аутентификацию.

1. Создаем класс AutheficationFilter

public class AuthenticationFilter extends UsernamePasswordAuthenticationFilter   
}

1. В классе создаём метод attemptAuthentication – метод, который будет автоматически вызываться каждый раз, когда пользователь будет отправлять запрос по адресу localhost:8080/login  
   Метод получает того пользователя, который был отправлен в реквесте (модель пользователя – LoginRequestDto

@Override  
public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)  
 throws AuthenticationException {  
 try {  
 LoginRequestDto creds = new ObjectMapper()  
 .readValue(request.getInputStream(), LoginRequestDto.class);  
 return authenticationManager.authenticate(  
 new UsernamePasswordAuthenticationToken(  
 creds.getUserEmail(),  
 creds.getPassword(),  
 new ArrayList<>()  
 )  
 );  
 } catch (IOException e) {  
 throw new RuntimeException();  
 }  
}

1. После того, как метод получает пользователя, он вызывает метод loadUserByUsername
2. Метод loadUserByUsername будет расположен в UserServiceImpl. Это метод6 который должен физически проверить – существует ли такой пользователь с базе данных. Так как он работает с User – размещаем его в UserServiceDto

!!!Для того, что бы получить правильный loadByUser (это метод, который наследуется от класса UserDetails (SpringSecurity), необходимо, что бы интерфейс/ UserService наследовал этот класс

public interface UserService extends UserDetailsService

1. После того, как пользователь будет получен, метод attemptAuthentication сравнит этих двух пользователей, и если все в порядке – вызовет метод successAuthentication, который сгенерирует Token и добавит его в request header

@Override  
protected void successfulAuthentication(HttpServletRequest request,  
 HttpServletResponse response,  
 FilterChain chain,  
 Authentication authResult) throws IOException, ServletException {  
 String userName = ((User)authResult.getPrincipal()).getUsername();  
  
 String token = Jwts.*builder*()  
 .setSubject(userName)  
 .setExpiration(new Date(System.*currentTimeMillis*() + SecurityConstants.*EXPIRATION\_TIME*))  
 .signWith(SignatureAlgorithm.*HS512*, SecurityConstants.*getTokenSecret*())  
 .compact();  
 response.addHeader(SecurityConstants.*HEADER\_STRING*, SecurityConstants.*TOKEN\_PREFIX* + token  
 );  
}

1. Для того, что бы метод сработал были добавлены –

Класс, который получает из application.properties значение для

tokenSecret=superSecretToken

и добавляет его в Environment

@Component  
public class AppProperties {  
  
 @Autowired  
 private Environment env;  
  
 public String getTokenSecret()  
 {  
 return env.getProperty("tokenSecret");  
 }  
}

1. Так же был добавлен в класс BeanConfig – Bean для SpringApplicationContext

@Bean  
public SpringApplicationContext springApplicationContext() {  
 return new SpringApplicationContext();  
}

@Bean(name = "AppProperties")  
public AppProperties getAppProperties() {  
 return new AppProperties();  
}

1. Для того, что бы дать классу UserServiceImpl доступ к ApplicationContext –

public class SpringApplicationContext implements ApplicationContextAware {  
 private static ApplicationContext *CONTEXT*;  
  
 @Override  
 public void setApplicationContext(ApplicationContext context) throws BeansException {  
 *CONTEXT* = context;  
 }  
  
 public static Object getBean(String beanName) {  
 return *CONTEXT*.getBean(beanName);  
 }  
}

1. Теперь необходимо добавить этот фильтр в конфигурацию WebSecurity

…

*/\*\*  
 \* all others should have authenticated  
 \*/* .anyRequest().authenticated()  
 .and().addFilter(new AuthenticationFilter(authenticationManager()));  
*/\*\**

*…*

Кроме этого, тут же необходимо метод

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.userDetailsService(userDetailsService).passwordEncoder(bCryptPasswordEncoder);  
}

Который будет использовать attemptAuthentication когда ему нужно будет расшифровать пароль, который получен из базы данных.

1. Что бы получить доступ к userDetailsService и bCryptPasswordEncoder

private final UserService userService;  
private final BCryptPasswordEncoder bCryptPasswordEncoder;  
  
public WebSecurity(UserService userService, BCryptPasswordEncoder bCryptPasswordEncoder) {  
 this.userService = userService;  
 this.bCryptPasswordEncoder = bCryptPasswordEncoder;  
}

1. Реализовываем метод loadByUser

@Override  
 public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {  
 User user = userRepository.findByUserEmail(email);  
 ;  
 if (user == null) {  
// *TODO тут будет exception* }  
 return   
 new org.springframework.security.core.userdetails.User(user.getUserEmail(),  
 user.getPassword(), true,  
 true, true,   
 true, new ArrayList<>());  
   
 }

Меняем путь к логину. По умолчанию спринг строит свой адресс. Но нам он не нравится. Мы хотим “/users/login”

Для этого, и для подключения фильтра аутентификации дописываем в WebSecurity

*/\*\*  
 \* login user now by link .../users/login  
 \*/*protected AuthenticationFilter getAuthenticationFilter() throws Exception {  
 final AuthenticationFilter filter = new AuthenticationFilter(authenticationManager());  
 filter.setFilterProcessesUrl("/users/login");  
 return filter;  
}

Проверяем – запускаем

localhost:8080/users/login

{

"userEmail":"user@mail.com",

"password":"password"

}

Получаем ответ 200 OK

Теперь в Headers response можно увидеть токен ->

Изображение выглядит как стол

Автоматически созданное описание

Следующий шаг – создание фильтра, который должен проверить – релевантный ли токен у пользователя при доступах к методам.

Пока мы все еще будем получать 403 ошибку в методе “/test” потому что мы не написали функционал, который будет проверять токен

Изображение выглядит как текст

Автоматически созданное описание

КОД добавления аутентификации доступен в ветке [tokenPlusAuthentication](https://github.com/natalyaKh/security-template/tree/tokenPlusAuthentication)

## Добавляем авторизацию

Делаем так, что бы только авторизованные пользователи могли получить доступ к приложению user-service. Используем токен для проверки авторизации

1. Пишем класс AuthorizationFilter

public class AuthorizationFilter extends BasicAuthenticationFilter {  
   
 public AuthorizationFilter(AuthenticationManager authenticationManager) {  
 super(authenticationManager);  
 }  
  
}

1. Каждый раз, когда происзодит обращение к любой конечной точке, которая требует авторизации, происходит вызов метода doFilterInternal

Метод имеет доступ к headers

@Override  
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,  
 FilterChain chain) throws IOException, ServletException {  
 String header = request.getHeader(SecurityConstants.*HEADER\_STRING*);  
 if(header == null || !header.startsWith(SecurityConstants.*TOKEN\_PREFIX*)){  
 chain.doFilter(request, response);  
 return;  
 }  
 UsernamePasswordAuthenticationToken authentication = getAuthentication(request);  
 SecurityContextHolder.*getContext*().setAuthentication(authentication);  
 chain.doFilter(request, response);  
}

1. Когда метод проверил что все ОК, и токен в принципе присутствует, он вызыввает метод getAuthentication который проверяет релевантность токена

private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {  
 String token = request.getHeader(SecurityConstants.*HEADER\_STRING*);  
  
 if (token != null) {  
 token = token.replace(SecurityConstants.*TOKEN\_PREFIX*, "");  
 String user = Jwts.*parser*()  
 .setSigningKey(SecurityConstants.*getTokenSecret*())  
 .parseClaimsJws(token)  
 .getBody()  
 .getSubject();  
 if (user != null) {  
 return new UsernamePasswordAuthenticationToken(user,  
 null, new ArrayList<>());  
 }  
 return null;  
 }  
 return null;  
}

1. Добавляем фильтр в конфигурацию WebSecurity

.and()  
 .addFilter(getAuthenticationFilter())  
 */\*\*  
 \* checking authorization  
 \*/* .addFilter(new AuthorizationFilter(authenticationManager()));  
  
*/\*\*  
 \* clean headers after all request*

1. Кроме этого, добавляем настройки CORS

*/\*\**

*\* add cors  
 \*/*@Bean  
public CorsConfigurationSource corsConfigurationSource() {  
 final CorsConfiguration configuration = new CorsConfiguration();  
  
 configuration.setAllowedOrigins(Arrays.*asList*("\*"));  
 configuration.setAllowedMethods(Arrays.*asList*("GET", "POST", "PUT", "DELETE", "OPTIONS"));  
 configuration.setAllowCredentials(true);  
 configuration.setAllowedHeaders(Arrays.*asList*("\*"));  
  
 final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  
 source.registerCorsConfiguration("/\*\*", configuration);  
  
 return source;  
}

Проверяем -> на методе test

Изображение выглядит как текст

Автоматически созданное описаниеРаботает!!! Код можно посмотреть в ветке [**authorization**](https://github.com/natalyaKh/security-template/tree/authorization)

# Настраиваем роли

## Роли

– всего будет создано 3 роли

* 1. ROLE\_USER
  2. ROLE\_ADMIN
  3. ROLE\_SUPERADMIN

## Права

– всего будет создано 3 права

* 1. READ
  2. WRITE
  3. DELETE

В Java вмеcто привилегий используется authority. По своей сути роль – это коллекция authority.

И Роли и Привилегии должны быть помещены в коллекцию <GrantedAuthory>

Текущий процесс аутентификации и авторизации пользователя в приложении – когда пользователь логинится происходит аутентикация. После аутентификации пользователь получает токен, и дальше происходит авторизация. То есть фильтры спринга проверяют, какие права есть у этого пользователя. И вот тут начинают работать роли и привелегии.

!!! У одной роли может быть много authority, и наоборот. Следовательно у нас связи - @ManyToMany

## Реализация

1. [Создаем у пользователей поле role](#_Создаем_у_пользователей)
2. [Создаем у пользователей authority](#_Создаем_у_пользователей_1)
3. [Добавляем в базу данных roles и authority](#_Добавляем_в_базу)
4. [Добавляем супер администратора](#_Добавляем_супер_администратора)

### Создаем у пользователей поле role

1. Добавляенм название таблички

@Table(name = "users")

1. Добавляем поле

@JoinTable(name = "users\_roles", joinColumns = @JoinColumn  
 (name = "users\_id", referencedColumnName = "id"))  
@ManyToMany(cascade = CascadeType.*PERSIST*, fetch = FetchType.*EAGER*)  
private Collection<RoleEntity> roles;

1. Создаем роли

@Table(name = "roles")  
@AllArgsConstructor  
@NoArgsConstructor  
@Getter  
@Setter  
@Builder  
@ToString  
@Entity  
public class RoleEntity extends BaseEntity{  
 @Column(nullable = false, length = 20)  
 String name;  
 @ManyToMany(mappedBy = "roles")  
 private Collection<User> users;  
}

После запуска, если пойдем в базу данных, то увидим, что там есть таблички USERS ROLES, но кроме этого создавать еще одна табличка – USERS\_ROLES которая содержит в себе поля user\_id и ole\_id



### Создаем у пользователей authority

1. Добавляем поле в RoleEntity

@ManyToMany(cascade= { CascadeType.*PERSIST* }, fetch = FetchType.*EAGER* )  
@JoinTable(name="roles\_authorities",  
 joinColumns=@JoinColumn(name="roles\_id",referencedColumnName="id"),  
 inverseJoinColumns=@JoinColumn(name="authorities\_id",referencedColumnName="id"))  
private Collection<AuthorityEntity> authorities;

1. Добавляем класс AuthorityEntity

@ManyToMany(mappedBy="authorities")  
private Collection<RoleEntity> roles;

1. Добавляем репозитории для обеих новых сущностей
2. Если сейчас запустить приложение, то у нас будут таблички-

Изображение выглядит как текст

Автоматически созданное описание

И в дальнейшем, мы будем проверять из таблицы users\_roles какие роли есть у пользователя, а потом по id роли из таблицы roles\_authorities получаем привилегии пользователя.

### Добавляем в базу данных roles и authority

Мы хотим добиться того, что бы при старте приложения в базе данных были роли и authorities

Для этого создаем класс InitialUsersSetup, который будет содержать в себе событие ApplicationReadyEvent

@Component  
public class InitialUsersSetup {  
}

В этом классе будет метод createAuthory, который будет возвращать AuthorityEntity

public InitialUsersSetup(AuthorityRepository authorityRepository) {  
 this.authorityRepository = authorityRepository;  
 }  
  
 private AuthorityEntity createAuthority(String name){  
 AuthorityEntity authorityEntity = authorityRepository.findByName(name);  
 if(authorityEntity == null){  
 authorityEntity = new AuthorityEntity(name);  
 authorityRepository.save(authorityEntity);  
   
 }  
 return authorityEntity;  
}

теперь физически создаем права – метод onApplicationEvent, над которым стоит аннотация, делающая его event

Изображение выглядит как текст

Автоматически созданное описание

Кроме этого мы создали ENUM класс

public enum Authorities {  
 *READ*, *WRITE*, *DELETE*}

Теперь если проверить базу – то увидим, что мы создали authorities

Изображение выглядит как стол

Автоматически созданное описание

Делаем то же самое для ROLES

private RoleEntity createRole(String name, Collection<AuthorityEntity> authorityEntities ){  
 RoleEntity roleEntity = roleRepository.findByName(name);  
 if(roleEntity == null){  
 roleEntity = new RoleEntity(name);  
 roleEntity.setAuthorities(authorityEntities);  
 roleRepository.save(roleEntity);  
 }  
 return roleEntity;  
}

добавляем код создания

*/\*\*  
 \* creating roles  
 \*/  
/\*\*  
 \* user can write his profile and read his information  
 \*/* RoleEntity userRole = createRole(Roles.*USER*.name(), Arrays.*asList*(readAuthority,  
 writeAuthority));  
*/\*\*  
 \* admin can write his profile and read all information and delete all profiles (not admin profile)  
 \*/* RoleEntity adminRole = createRole(Roles.*ADMIN*.name(), Arrays.*asList*(readAuthority,  
 writeAuthority, deleteAuthority));  
*/\*\*  
 \* super admin can do everything  
 \*/* RoleEntity superAdmin = createRole(Roles.*SUPER\_ADMIN*.name(), Arrays.*asList*(readAuthority,  
 writeAuthority, deleteAuthority));  
 }

Проверяем БД ->

Изображение выглядит как стол

Автоматически созданное описание

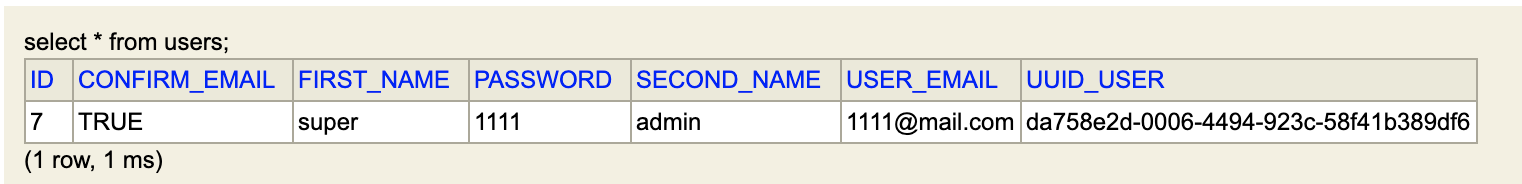
Видим что все работает. Мы не просто создали 3 роли и 3 права доступа – мы еще рассказали, у какой роли какие права (roles\_authorities таблица)

### Добавляем супер администратора

Так как мы не хотим давать при регистрации право на супер администрирование – мы хардкорно пропишем одного суперадминистратора

*/\*\*  
 \* create superAdministrator  
 \*/* if (superAdmin == null) return;  
 User superAdministrator = User.*builder*()  
 .uuidUser(UUID.*randomUUID*().toString())  
 .firstName("super")  
 .secondName("admin")  
 .password(bCryptPasswordEncoder.encode("1111"))  
 .userEmail("1111@mail.com")  
 .confirmEmail(true)  
 .roles(Arrays.*asList*(superAdmin))  
 .build();  
 userRepository.save(superAdministrator);  
 }

Проверяем в базе



Супер Администратор создан!

# Реализация фильтров для проверки роли и прав

Теперь когда пользователь заходит в систему, нам нужно проверять какая у него роль и права, и установить их для этого пользователя. По сути, теперь наш токен должен содержать еще и роль пользователя. Заодно добавим в header имя пользователя, его email и роль.

Сейчас во время логина у нас срабатывает Authentication filtr. Этот метод вызывает loadUserByName(), в котором возвращает пользователя по заданным параметрам. Теперь нужно, что бы этот метод добавлял еще и authorities возвращаемому пользователю.

1. [Добавляем authorities пользователю во время проверки](#_Добавляем_authorities_пользователю)
2. [Изменение authentication filter](#_Изменение_authentication_filter)

## Добавляем authorities пользователю во время проверки

@Override  
 public UserPrincipal loadUserByUsername(String email) throws UsernameNotFoundException {  
 User user = userRepository.findByUserEmail(email);  
 if (user == null) {  
// *TODO тут будет exception* }  
 return new UserPrincipal(user);  
// new org.springframework.security.core.userdetails.User(user.getUserEmail(),  
// user.getPassword(), true,  
// true, true,  
// true, new ArrayList<>());  
  
 }

Меняем метод так, что бы он возвращал Principal объект.

1. Добавляем класс UserPrincipal

public class UserPrincipal implements UserDetails {  
 User user;  
 public UserPrincipal(User user) {  
 this.user = user;  
 }

В этом классе будет несколько методов6,которые легко реализовать,

@Override  
public String getPassword() {  
 return this.user.getPassword();  
}  
  
@Override  
public String getUsername() {  
 return this.user.getFirstName() + this.user.getSecondName();  
}  
  
@Override  
public boolean isAccountNonExpired() {  
 return true;  
}  
  
@Override  
public boolean isAccountNonLocked() {  
 return true;  
}  
  
@Override  
public boolean isCredentialsNonExpired() {  
 return true;  
}  
  
@Override  
public boolean isEnabled() {  
 return this.user.getConfirmEmail();  
}  
  
public UserPrincipal() {  
}

и один важный метод, ради которого мы создали этот класс ->

public Collection<? extends GrantedAuthority> getAuthorities()

именно в этом методе мы добавляем GrantedAuthory – тот объект, которого ждет spring security для определения ролей.

@Override  
public Collection<? extends GrantedAuthority> getAuthorities() {  
 List<GrantedAuthority> authorities = new ArrayList<>();  
 List<AuthorityEntity> authorityEntities = new ArrayList<>();  
 Collection<RoleEntity> roles = user.getRoles();  
 if(roles == null) return authorities;  
 roles.forEach((role)-> {  
 authorities.add(new SimpleGrantedAuthority(role.getName()));  
 authorityEntities.addAll(role.getAuthorities());  
 });  
 authorityEntities.forEach(authorityEntity -> {  
 authorities.add(new SimpleGrantedAuthority(authorityEntity.getName()));  
 });  
 return authorities;  
}

## Изменение authentication filter

Теперь пришло время рассказать AuthenticationFilter, что мы используем новый класс, вмеcто User

1. Вместо

String userName = ((User) authResult.getPrincipal()).getUsername();

1. Ставим

String userName = ((UserPrincipal) authResult.getPrincipal()).getUsername();

1. Раньше мы ставили пустой массив при создании. Теперь нам нужно вернуть туда список прав пользователя. Меняем

private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {  
 String token = request.getHeader(SecurityConstants.*HEADER\_STRING*);  
  
 if (token != null) {  
 token = token.replace(SecurityConstants.*TOKEN\_PREFIX*, "");  
 String user = Jwts.*parser*()  
 .setSigningKey(SecurityConstants.*getTokenSecret*())  
 .parseClaimsJws(token)  
 .getBody()  
 .getSubject();  
 if (user != null) {  
 User userEntity = userRepository.findByUserEmail(user);  
 UserPrincipal userPrincipal = new UserPrincipal(userEntity);  
 return new UsernamePasswordAuthenticationToken(user,  
 null, userPrincipal.getAuthorities());  
 }  
 return null;  
 }  
 return null;  
}

1. Кроме этого, добавили

private final AuthenticationManager authenticationManager;  
private final UserRepository userRepository;  
  
public AuthenticationFilter(AuthenticationManager authenticationManager, UserRepository userRepository) {  
 this.authenticationManager = authenticationManager;  
 this.userRepository = userRepository;  
}

1. Ещё

public class AuthorizationFilter extends BasicAuthenticationFilter {  
 final UserRepository userRepository;  
 public AuthorizationFilter(AuthenticationManager authenticationManager, UserRepository userRepository) {  
 super(authenticationManager);  
 this.userRepository = userRepository;  
 }

1. Еще

public WebSecurity(UserService userService, BCryptPasswordEncoder bCryptPasswordEncoder, UserRepository userRepository) {  
 this.userService = userService;  
 this.bCryptPasswordEncoder = bCryptPasswordEncoder;  
 this.userRepository = userRepository;  
}

1. И репозиторий

*/\*\*  
 \* login user now by link .../users/login  
 \*/*protected AuthenticationFilter getAuthenticationFilter() throws Exception {  
 final AuthenticationFilter filter = new AuthenticationFilter(authenticationManager(), userRepository);  
 filter.setFilterProcessesUrl("/users/login");  
 return filter;  
}

# Создаем остальные методы контроллера

Для того, что бы проверить как работает доступ по правам – нужно написать методы, к которым этот доступ будет применяться.

В нашем случае нужно добавить методы

* /register - Регистрация пользователя с ролью ADMIN, /register - Регистрация пользователя с ролью USER - DONE
* /login – любой пользователь может залогиниться, если он до этого зарегистрировался. - DONE
* /getAllUsers – только для администратора
* /getUser - администратор и суперадминистратор имеет доступ кол всем записям, пользователь – только к своей записи
* /updateUser – изменить пользователя может только сам пользователь, независимо от того, какая у него роль. Кроме этого, супер администратор может менять любую запись
* /deleteUser - администратор имеет доступ кол всем записям, пользователь – только к своей записи!!! Если пользователь, которого хотят удалить имеет роль ADMIN, то удалить его может только SUPER\_ADMINISTRATOR

Пока добавляем их по стандартной схеме.

1. [Обновляем контроллер](#_Обновляем_контроллер)
2. [Обновляем сервис](#_Обновляем_сервис)
3. [Закрываем удаленным пользователям доступ к login](#_Закрываем_удаленным_пользователям)
4. [Добавили pingController](#_Добавили_pingController)

## Обновляем контроллер

@GetMapping("/{userEmail}")  
public ResponseEntity getUserByUserUuid(@PathVariable String userEmail){  
 return new ResponseEntity(  
 userService.getUserByUserUuid(userEmail), HttpStatus.*OK* );  
}  
  
@PutMapping("put/{userEmail}/{firstName}/{lastName}")  
public ResponseEntity updateUserByUserUuid(@PathVariable String userEmail,  
 @PathVariable String firstName,  
 @PathVariable String lastName){  
 return new ResponseEntity(  
 userService.updateUserByUserUuid(userEmail, firstName, lastName), HttpStatus.*OK* );  
}

@DeleteMapping("/{userEmail}")  
public ResponseEntity deleteUserByUserUuid(@PathVariable String userEmail){  
 boolean deleted = userService.deleteUserByUserUuid(userEmail);  
 if(deleted){  
 return new ResponseEntity(  
 Status.*SUCCESSFUL*, HttpStatus.*OK* );  
 }  
 return new ResponseEntity(  
 Status.*ERROR*, HttpStatus.*GONE* );  
}

## Обновляем сервис

private User getUserFromDto(UserDto userDto) {  
 return modelMapper.map(userDto, User.class);  
 }  
  
 private UserDto getDtoFromUser(User user) {  
 return modelMapper.map(user, UserDto.class);  
 }  
  
 @Override  
 public List<UserDto> getAllUsers() {  
 List<User> userList = userRepository.findAll();  
 if (userList.size() == 0) return new ArrayList<>();  
 List<UserDto> userDtoList = userList.stream().map(this::getDtoFromUser).collect(Collectors.*toList*());  
 return userDtoList;  
 }  
  
 @Override  
 public UserDto getUserByUserUuid(String userEmail) {  
 User user = userRepository.findByUserEmail(userEmail);  
 if (user == null) {  
// *TODO тут будет exception* }  
 return getDtoFromUser(user);  
 }  
  
 @Override  
 public UserDto updateUserByUserUuid(String userEmail, String firstName, String lastName) {  
 User user = userRepository.findByUserEmailAndDeleted(userEmail, false);  
 if (user == null) {  
// *TODO тут будет exception* }  
 if (firstName != null) user.setFirstName(firstName);  
 if (lastName != null) user.setSecondName(lastName);  
 userRepository.save(user);  
 return getDtoFromUser(user);  
 }

@Override  
 @Transactional  
 public Boolean deleteUserByUserUuid(String userEmail) {  
 Integer rez = userRepository.deleteUserByUserEmail(userEmail);  
// if (user == null) {  
// return false;  
// }  
// userRepository.delete(user);  
 return rez==1?true:false;  
 }  
  
 @Override  
 public UserPrincipal loadUserByUsername(String email) throws UsernameNotFoundException {  
 User user = userRepository.findByUserEmail(email);  
 if (user == null) {  
// *TODO тут будет exception* }  
 return new UserPrincipal(user);  
// new org.springframework.security.core.userdetails.User(user.getUserEmail(),  
// user.getPassword(), true,  
// true, true,  
// true, new ArrayList<>());  
  
 }

Кроме этого, добавляем поле deleted пользователю, что бы была возможность его удалить.

* Users

Boolean deleted;  
@Column(nullable = false)

* UserDto

private Boolean deleted = false;

Добавили класс для ответов у удаленного пользователя

public enum Status {  
 *SUCCESSFUL*, *ERROR*}

В userRepository нужно добавить метод, который удалит пользователя сразу, бкд проверки. И если такого пользователя нет – вернет ошибку

@Modifying  
@Query(value = "update users set deleted = true where user\_email=:userEmail",nativeQuery = true)  
Integer deleteUserByUserEmail(String userEmail);  
  
User findByUserEmailAndDeleted(String userEmail, boolean b);

## Закрываем удаленным пользователям доступ к login

Теперь, когда у нас есть удаленные пользователи, необходимо закрыть им доступ к аутентификации.

Делать это нужно в классе UserPrincipal

@Override  
public boolean isAccountNonExpired() {  
 return !this.user.getDeleted();  
}

Мы предполагаем, что удаленные пользователи это те же пользователи, у которых просрочен аккаунт

И последнее – тому супер администратору, которого мы дефолтно создаем необходимо добавить поле

.deleted(false)

Теперь осталось проверить как новые методы работают для зарегистрированных пользователей. Пока все еще без ролей.

## Добавили pingController

@RestController  
@RequestMapping("/ping/v1")  
public class PingController {  
 @GetMapping()  
 public String ping(){  
 return "I am work. Access only for SuperAdministrator";  
 }  
}

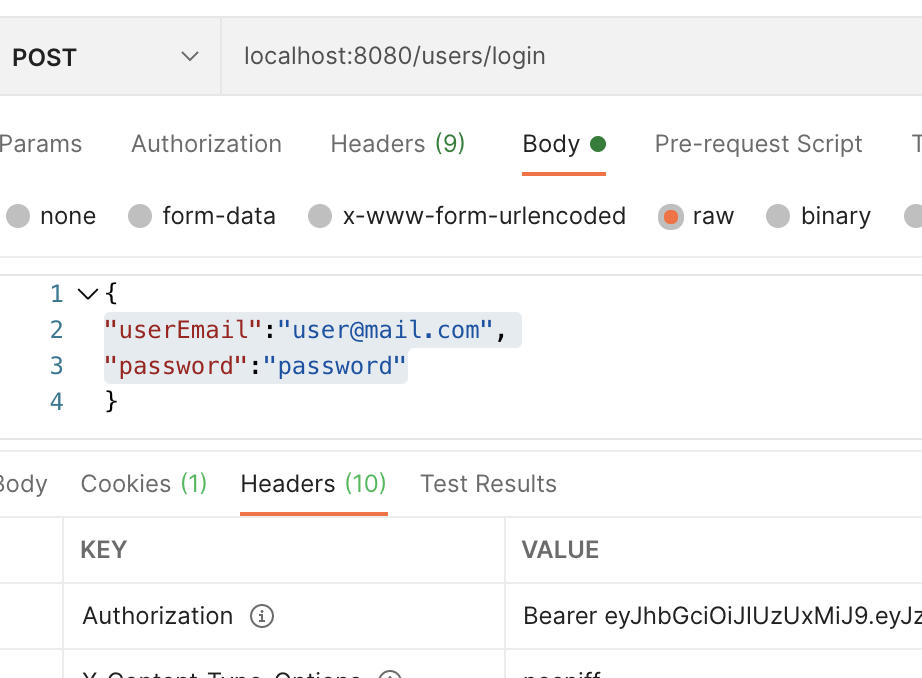
# Проверка методов контроллера

## Добавляем пользователя

Изображение выглядит как текст

Автоматически созданное описание

## Логиним пользователя



## Получаем всех пользователей

Изображение выглядит как текст

Автоматически созданное описание

## Получаем пользователя по e-mail

Изображение выглядит как текст

Автоматически созданное описание

## Обновление пользователя

Изображение выглядит как текст

Автоматически созданное описание

## Удаление пользователяИзображение выглядит как текст Автоматически созданное описание

!!! если удалить везде токен – везде будет приходить ответ 403-FORBIDDEN

# Добавляем роль при регистрации пользователя

В методе createUser класс UserServiceImpl добавляем роль пользователю.   
Логика – если пользователь зашел со специальным кодом – это ADMIN если нет – USER

## Добавляем код

public UserDto createUser(UserDto userDto) {  
 AuthorityEntity readAuthority = initialUsersSetup.createAuthority(Authorities.*READ*.name());  
 AuthorityEntity writeAuthority = initialUsersSetup.createAuthority(Authorities.*WRITE*.name());  
 AuthorityEntity deleteAuthority = initialUsersSetup.createAuthority(Authorities.*DELETE*.name());  
 User restoreUser = userRepository.findByUserEmail(userDto.getUserEmail());  
 if (restoreUser != null) {  
// *TODO тут будет exception* }  
 User user = getUserFromDto(userDto);  
 user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));  
 String defaultCode = SecurityConstants.*getAdminCode*();  
 if (userDto.getCode()!= "" & userDto.getCode().equals(defaultCode)) {  
 RoleEntity adminRole = initialUsersSetup.createRole(Roles.*ROLE\_ADMIN*.name(), Arrays.*asList*(readAuthority,  
 writeAuthority, deleteAuthority));  
 user.setRoles(Arrays.*asList*(adminRole));  
 } else {  
 RoleEntity userRole = initialUsersSetup.createRole(Roles.*ROLE\_USER*.name(), Arrays.*asList*(readAuthority,  
 writeAuthority));  
 user.setRoles(Arrays.*asList*(userRole));  
 }  
 user.setUuidUser(UUID.*randomUUID*().toString());  
 userRepository.save(user);  
 return modelMapper.map(user, UserDto.class);  
 }

## Проверяем

Создаем пользователя с ролью USER

Изображение выглядит как текст

Автоматически созданное описание

Создаем пользователя с ролью ADMIN

Изображение выглядит как текст

Автоматически созданное описание

Проверяем в БД

В таблице users есть 3 пользователя

Изображение выглядит как текст

Автоматически созданное описание

Проверяем роли

Изображение выглядит как стол

Автоматически созданное описаниеИзображение выглядит как стол

Автоматически созданное описание

Смотрим на пользователя с ролью SUPER\_ADMIN – его id = 7. D в таблице USERS\_ROLES видим, что у пользователя с id=7 роль с id = 6. Проверяем в таблице ROLES видим, что роль с id = 6 это роль супер администратора!

Так же проверяем все остальные роли остальных пользователей

В таблице ROLES\_AUTHORITIES можно проверить права пользователей исходя из роли.

Изображение выглядит как стол

Автоматически созданное описание

# Пишем проверку прав на уровне метода

1. [Проверяем метод удаления пользователя.](#_Проверяем_метод_удаления)
2. [Проверяем метод получения всех пользователей](#_Проверяем_метод_получения)
3. [Проверяем получение пользователя по email](#_Проверяем_получение_пользователя)
4. [Проверяем метод обновления пользователя](#_Проверяем_метод_обновления)

## Проверяем метод удаления пользователя.

Мы помним, что обычный пользователь не может удалять ничего, админ может удалять свой профиль, супер администратор может удалить любой профиль.

То есть мы разрешаем удаление только супер администратору, или администратору с таки же id как тот, который он хочет удалить

Имплементируем, исправляя по дороге БАГИ

1. Класс webSecurity – добавляем аннотацию

@EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled = true)

1. В контроллере добавляем над методом аннотацию, где разрешаем удалять только суперадминистратору, или администратору, но только свой профиль. Пользователь удалять ничего не может

@PreAuthorize("hasRole('SUPER\_ADMIN') or hasRole('ADMIN') and #userEmail == principal.id")  
@DeleteMapping("/{userEmail}")  
public ResponseEntity deleteUserByUserUuid(@PathVariable String userEmail){  
 boolean deleted = userService.deleteUserByUserUuid(userEmail);  
 if(deleted){  
 return new ResponseEntity(  
 Status.*SUCCESSFUL*, HttpStatus.*OK* );  
 }  
 return new ResponseEntity(  
 Status.*ERROR*, HttpStatus.*GONE* );  
}

1. Фильтры – вместо

// Этот метод проверяем верный ли токен  
 private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {  
 String token = request.getHeader(SecurityConstants.*HEADER\_STRING*);  
  
 if (token != null) {  
 token = token.replace(SecurityConstants.*TOKEN\_PREFIX*, "");  
 String user = Jwts.*parser*()  
 .setSigningKey(SecurityConstants.*getTokenSecret*())  
 .parseClaimsJws(token)  
 .getBody()  
 .getSubject();  
 if (user != null) {  
 User userEntity = userRepository.findByUserEmail(user);  
 UserPrincipal userPrincipal = new UserPrincipal(userEntity);  
 return new UsernamePasswordAuthenticationToken(user,  
 null, userPrincipal.getAuthorities());  
 }  
 return null;  
 }  
 return null;  
 }

1. Возвращаем principalUser вместо userEntity

// Этот метод проверяем верный ли токен  
private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {  
 String token = request.getHeader(SecurityConstants.*HEADER\_STRING*);  
  
 if (token != null) {  
 token = token.replace(SecurityConstants.*TOKEN\_PREFIX*, "");  
 String user = Jwts.*parser*()  
 .setSigningKey(SecurityConstants.*getTokenSecret*())  
 .parseClaimsJws(token)  
 .getBody()  
 .getSubject();  
 if (user != null) {  
 User userEntity = userRepository.findByUserEmail(user);  
 UserPrincipal userPrincipal = new UserPrincipal(userEntity);  
 return new UsernamePasswordAuthenticationToken(userPrincipal,  
 null, userPrincipal.getAuthorities());  
 }  
 return null;  
 }  
 return null;  
}

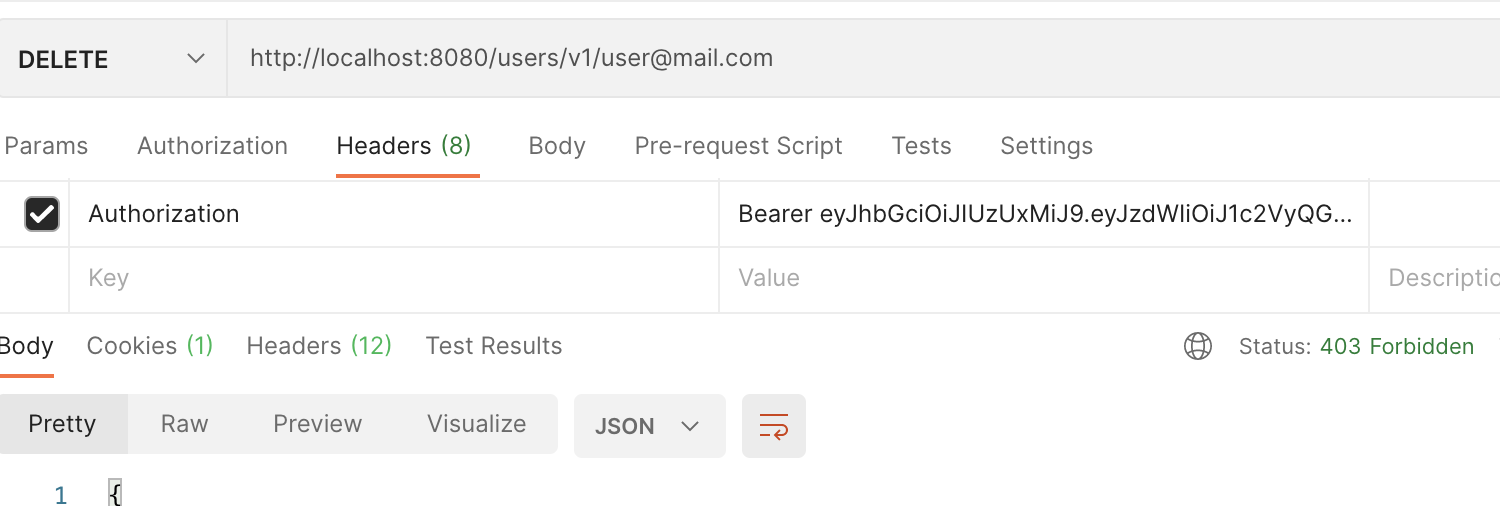
1. PrincipalUser добавляем + getter/setter

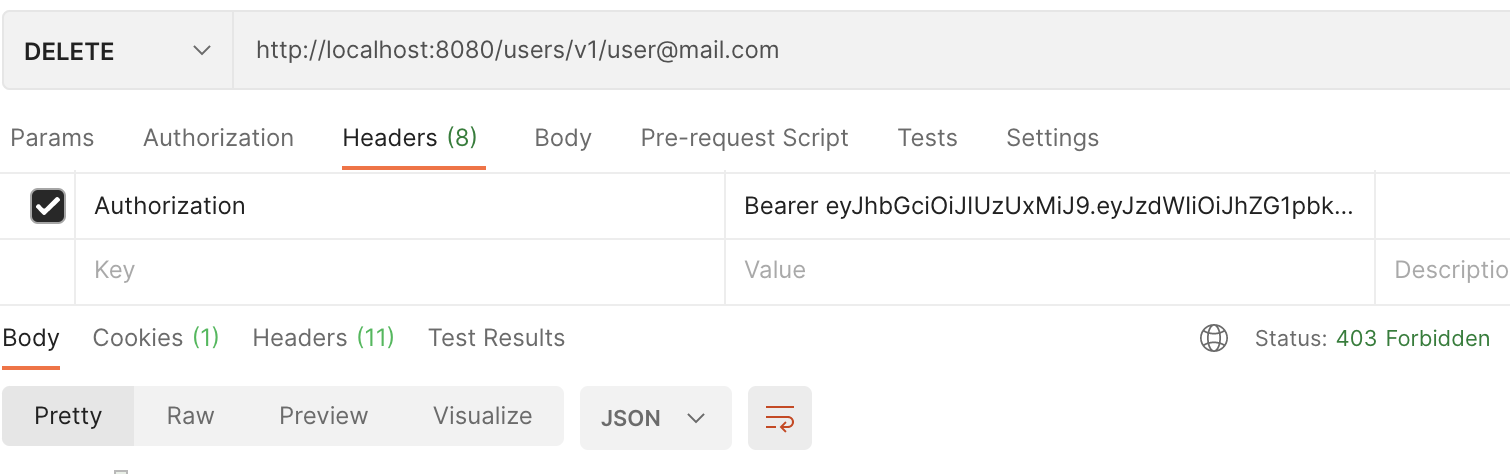
public class UserPrincipal implements UserDetails {  
 private User user;  
 private String id;  
 public UserPrincipal(User user) {  
 this.user = user;  
 this.id = user.getUserEmail();  
 }

1. Проверяем

Создаем 2 пользователя – администратора и пользователя

Логинимся под USER и пробуем удалить себя



Логинимся под администартором и пробуем удалить пользователя пробуем удалить свой профиль

Изображение выглядит как текст

Автоматически созданное описание

Еще раз создаем пользователей и пробуем удалить их под логином суперадминистратора

Изображение выглядит как текст

Автоматически созданное описаниеудаляем пользователя Изображение выглядит как текст

Автоматически созданное описание

Удаляет обоих!!!

## Проверяем метод получения всех пользователей

Логика метода – получать информацию про всех пользователей может администратор или супер адмнистратор

@PreAuthorize("hasAnyRole('SUPER\_ADMIN', 'ADMIN')")  
@GetMapping()  
public ResponseEntity getAllUsers(){  
 return new ResponseEntity(  
 userService.getAllUsers(), HttpStatus.*OK* );  
}

## Проверяем получение пользователя по email

Логика метода – пользователь может посмотреть свой профиль, администратор и суперадминистратор может посмотреть любой профиль

@PreAuthorize("hasAnyRole('SUPER\_ADMIN', 'ADMIN') or #userEmail == principal.id")  
@GetMapping("/{userEmail}")  
public ResponseEntity getUserByUserUuid(@PathVariable String userEmail){  
 return new ResponseEntity(  
 userService.getUserByUserUuid(userEmail), HttpStatus.*OK* );  
}

## Проверяем метод обновления пользователя

По логике метода, суперадминистратор может менять любой профиль, пользователь и администратор – только свой

@PreAuthorize("hasRole('SUPER\_ADMIN') or hasAnyRole('ADMIN', 'USER') and #userEmail == principal.id")  
@PutMapping("put/{userEmail}/{firstName}/{lastName}")  
public ResponseEntity updateUserByUserUuid(@PathVariable String userEmail,  
 @PathVariable String firstName,  
 @PathVariable String lastName) {  
 return new ResponseEntity(  
 userService.updateUserByUserUuid(userEmail, firstName, lastName), HttpStatus.*OK* );  
}

Проверяем контроллер ping

Изображение выглядит как текст

Автоматически созданное описание

Меняем контроллер– цель закрыть доступ к контроллеру всем кроме суперадминистратора

@RestController  
@RequestMapping("/ping/v1")  
@PreAuthorize("hasRole('SUPER\_ADMIN')")  
public class PingController {  
  
  
  
 @GetMapping()  
 public String pingAdmin(){  
 return "I am work on port: " + SecurityConstants.*getPort*();  
 }  
}

Теперь к методу pingAdmin доступ только у суперадминистратора,

Как и во всем методам, которые будут расположены в контроллере pingController