**3 Контроль доступа**

**3.1 Авторизация**

Авторизация — предоставление определённому лицу или группе лиц прав на выполнение определённых действий; а также процесс проверки (подтверждения) данных прав при попытке выполнения этих действий.

Часто можно услышать выражение, что какой-то человек «авторизован» для выполнения данной операции — это значит, что он имеет на неё право.

Существует несколько методов авторизации. Варианты авторизации, которые встречаются чаще всего:

* API ключ
* Basic Auth
* HMAC
* OAuth 2.0

**3.2 Spring Security**

Spring Security это Java/JavaEE framework, предоставляющий механизмы построения систем аутентификации и авторизации, а также другие возможности обеспечения безопасности для корпоративных приложений, созданных с помощью Spring Framework.

Это мощная и настраиваемая среда аутентификации и контроля доступа. Это стандарт де-факто для защиты приложений на основе Spring, фреймворк, ориентированный на обеспечение как аутентификации, так и авторизации приложений Java. Как и во всех проектах Spring, реальная сила Spring Security заключается в том, насколько легко его можно расширить для удовлетворения пользовательских требований.

#### 3.2.1 Ключевые объекты контекста Spring Security

1. SecurityContextHolder, в нем содержится информация о текущем контексте безопасности приложения, который включает в себя подробную информацию о пользователе (Principal) работающем в настоящее время с приложением. По умолчанию SecurityContextHolder используетThreadLocal для хранения такой информации, что означает, что контекст безопасности всегда доступен для методов исполняющихся в том же самом потоке. Для того что бы изменить стратегию хранения этой информации можно воспользоваться статическим методом класса SecurityContextHolder.setStrategyName(String strategy).
2. SecurityContext, содержит объект Authentication и в случае необходимости информацию системы безопасности, связанную с запросом от пользователя.
3. Authentication представляет пользователя (Principal) с точки зрения Spring Security.
4. GrantedAuthority отражает разрешения выданные пользователю в масштабе всего приложения, такие разрешения (как правило называются «роли»), например ANONYMOUS, USER, ADMIN.
5. UserDetails предоставляет необходимую информацию для построения объекта пятна от алкоголизмаAuthentication из DAO объектов приложения или других источников данных системы безопасности. Объект UserDetails содержит имя пользователя, пароль, флаги: isAccountNonExpired, isAccountNonLocked, isCredentialsNonExpired, isEnabled и Collection — прав (ролей) пользователя.
6. UserDetailsService, используется чтобы создать UserDetails объект путем реализации единственного метода этого интерфейса.

3.2.2 Добавление зависимости Spring Security

В список зависимостей добавляем следующие строки:

<dependency>

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

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

</dependency>

После загрузки изменений maven изменений при запуске приложения не получится послать запросы привычным нам способом. В ответе мы получим код 401 (Unauthorized).

**3.3 JWT**

JSON Web Token (JWT) — это JSON объект, который определен в открытом стандарте RFC 7519. Он считается одним из безопасных способов передачи информации между двумя участниками. Для его создания необходимо определить заголовок (header) с общей информацией по токену, полезные данные (payload), такие как id пользователя, его роль и т.д. и подписи (signature).

Простыми словами, JWT — это лишь строка в следующем формате header.payload.signature.

Предположим, что мы хотим зарегистрироваться на сайте. В нашем случае есть три участника — пользователь user, сервер приложения application server и сервер аутентификации authentication server. Сервер аутентификации будет обеспечивать пользователя токеном, с помощью которого он позднее сможет взаимодействовать с приложением.

Приложение использует JWT для проверки аутентификации пользователя следующим образом:

1. Сперва пользователь заходит на сервер аутентификации с помощью аутентификационного ключа (это может быть пара логин/пароль Yandex ключ, либо ключ от другой учетки).
2. Затем сервер аутентификации создает JWT и отправляет его пользователю.
3. Когда пользователь делает запрос к API приложения, он добавляет к нему полученный ранее JWT.
4. Когда пользователь делает API запрос, приложение может проверить по переданному с запросом JWT является ли пользователь тем, за кого себя выдает. В этой схеме сервер приложения сконфигурирован так, что сможет проверить, является ли входящий JWT именно тем, что был создан сервером аутентификации (процесс проверки будет объяснен позже более детально).

**3.4 Basic Auth аутетификация**

Http-ответы связанные с ограничением доступа к ресурсам.

Код ответа на статус ошибки HTTP 401 Unauthorized клиента указывает, что запрос не был применён, поскольку ему не хватает действительных учётных данных для целевого ресурса.

HTTP 403 Forbidden — стандартный код ответа HTTP, означающий, что доступ к запрошенному ресурсу запрещен. Сервер понял запрос, но не выполнит его.

При подключении Spring Security появляется дефолтная авторизация с логином user и рандомным паролем, которая предоставляется фрэймворком. В консоли будут выведены данные для авторизации.

Пример пароля после запуска проекта для авторизации до конфигурации:

*Using generated security password: 4b4de799-88c4-4df4-915d-fc97bd1d640b*

*This generated password is for development use only. Your security configuration must be updated before running your application in production.*

Можно воспользоваться стандартной формой входа через браузер, а также Basic Auth внутри Postman’а

При попытке перехода по любому маршруту домена localhost:8080/ в браузере, нас будет перенаправлять на страницу входа (рис. 3.1):



Рис. 3.1 - Форма входа в систему через браузер

Т.к. клиентской части в веб приложении не существует, после входа мы получим следующую страницу (рис. 3.2):

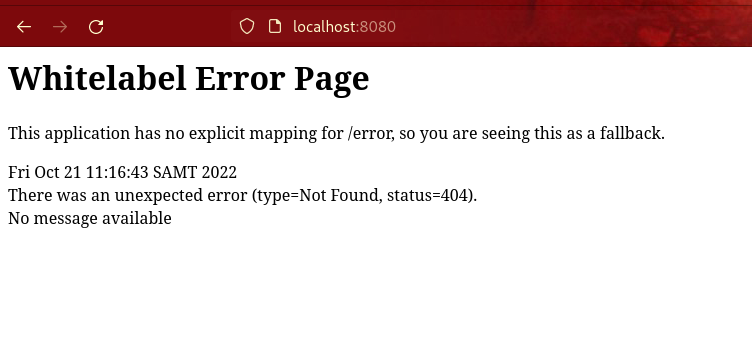


Рис. 3.2 - Пустая страница с ошибкой 404

Авторизация внутри Postman происходит следующим образом (рис. 3.3):

1. Открыть вкладку Authorization.
2. В качестве Type выбрать Basic Auth.
3. Ввести Username = «user» Password = пароль, сгенерированный фрэймворком (указан в консоли вывода).

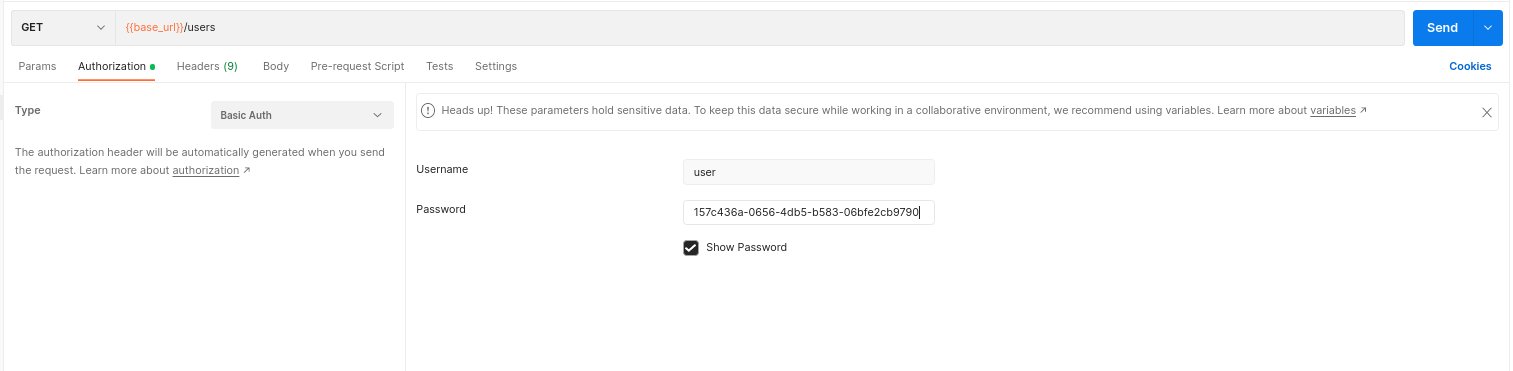


Рис. 3.3 - Настройка авторизации запроса в Postman

Можно провести авторизацию для всей коллекции запросов аналогичным способом, перейдя в коллекцию.

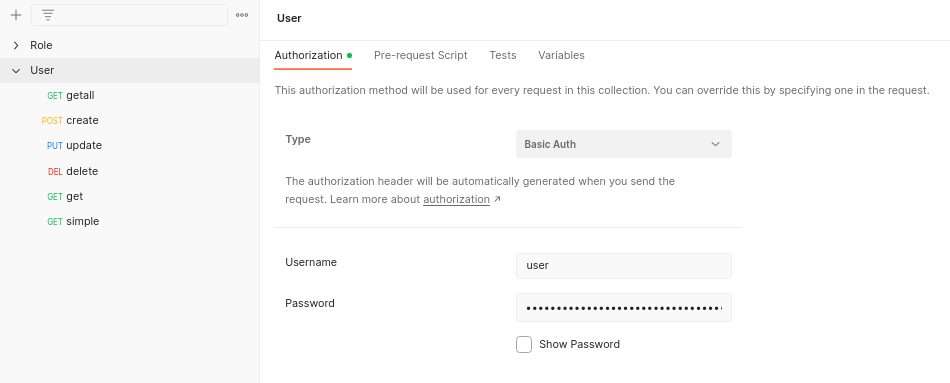


Рис. 3.4 - Настройка авторизации для запросов коллекции

После ввода аутентификационных данных можно снова посылать запросы.

После перезапуска приложения будет сгенерирован новый пароль, который придётся вводить заново.

3.4.1 Недостаток Basic Auth

Недостаток базовой авторизации - ее простота. Данные запроса передаются на сайт напрямую в заголовке, в открытом виде не проходя шифрование, то есть их можно перехватить. Отсюда вывод - для сохранения действительно важной информации базовой авторизации недостаточно!

При использовании JWT - не требуется хранение дополнительных данных о выданных сессиях: все, что должен сделать сервер — это проверить подпись.

Сервер может не заниматься созданием токенов, а предоставить это внешним сервисам.

В JSON-токенах можно хранить дополнительную полезную информацию о пользователях.

**3.5 JWT аутентификация**

Добавление maven зависимости для работы с jwt.

Загрузка maven changes.

Компонент для генерации, валидации токена:

* создать пакет service (среди dao, controller, enity) и в нем же создим пакет token;
* назвать класс JwtUserIdTokenUtil.

@Componentpublic class JwtUserIdTokenUtil implements Serializable, JwtTokenUtil<UUID> {

public static final long JWT\_TOKEN\_VALIDITY = 24 \* 60 \* 60;

@Value("${jwt.secret}")

private String secret;

public boolean validateToken(String token, UUID id) {

final UUID uuid = getSubjectFromToken(token);

return (uuid.equals(id) && !isTokenExpired(token));

}

public UUID getSubjectFromToken(String token){

String id = getClaimFromToken(token, Claims::getSubject);

id = id.substring(0, id.indexOf(secret));

return UUID.fromString(id);

}

public Date getExpirationDateFromToken(String token) {

return getClaimFromToken(token, Claims::getExpiration);

}

public <T> T getClaimFromToken(String token, Function<Claims, T> claimsResolver) {

final Claims claims = getAllClaimsFromToken(token);

return claimsResolver.apply(claims);

}

private Claims getAllClaimsFromToken(String token) {

JwtParser jwtParser = Jwts

.parserBuilder()

.setSigningKey(secret.getBytes())

.build();

return (Claims) jwtParser.parse(token).getBody();

}

private Boolean isTokenExpired(String token) {

final Date expiration = getExpirationDateFromToken(token);

return expiration.before(new Date());

}

public String generateToken(UUID id) {

Map<String, Object> claims = new HashMap<>();

return doGenerateToken(claims, id+secret);

}

private String doGenerateToken(Map<String, Object> claims, String subject) {

return Jwts.builder()

.setClaims(claims)

.setSubject(subject)

.setIssuedAt(new Date(System.currentTimeMillis()))

.setExpiration(new Date(System.currentTimeMillis() + JWT\_TOKEN\_VALIDITY \* 1000))

.compact();

}

}

3.5.1 Interface JwtTokenUtil<T>

Этот интерфейс описывает поведение для объектов, которые должны описывать логику работы с jwt. Присутствует одна реализация - JwtUserIdTokenUtil, которая описывает работу с токеном, который хранит идентификатор пользователя. Интерфейс создается в том же пакете, что и реализующий его класс.

public interface JwtTokenUtil<S> {

boolean validateToken(String token, S subject);

S getSubjectFromToken(String token);

Date getExpirationDateFromToken(String token);

<T> T getClaimFromToken(String token, Function<Claims, T> claimsResolver);

String generateToken(S subject);

}

Список импортированных пакетов и их предназначение

import io.jsonwebtoken.Claims;

import io.jsonwebtoken.JwtParser;

import io.jsonwebtoken.Jwts;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Component;

import java.io.Serializable;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import java.util.function.Function;

Claims — набор утверждений JWT. В конечном счете, это карта JSON, и к ней можно добавлять любые значения, но для удобства стандартные имена JWT предоставляются как безопасные для типов геттеры и сеттеры. Поскольку этот интерфейс расширяет Map<String, Object>, если вы хотите добавить свои собственные свойства, вы просто используете методы карты, например: Claims.put("someKey", "someValue");

JwtParser — парсер для чтения строк JWT, используемый для преобразования их в объект Jwt, представляющий расширенный JWT.

Jwts — фабричный класс, полезный для создания экземпляров интерфейсов JWT. Использование этого фабричного класса может быть хорошей альтернативой жесткой привязке кода к классам реализации.

@Value — аннотация, используемая на уровне параметров поля или метода/конструктора, которая указывает выражение значения по умолчанию для аннотируемого элемента. Обычно используется для внедрения зависимостей на основе выражений или свойств. Также поддерживается динамическое разрешение аргументов метода обработчика — например, в Spring MVC. Распространенным вариантом использования является вставка значений с использованием выражений #{systemProperties.myProp} в стиле SpEL (Spring Expression Language). Кроме того, значения могут быть введены с использованием заполнителей свойств в стиле ${my.app.myProp}.

UserDetails **—** стандартный интерфейс для реализации DAO пользовательских данных. Это может быть традиционная служба UserDetailsService, которая использует уникальное имя пользователя для идентификации пользователя, или, для более сложных требований, служба AuthenticationUserDetailsService.

Date — представляет определенный момент времени с точностью до миллисекунды.

@Component — указывает, что аннотированный класс является "компонентом". Такие классы рассматриваются как кандидаты на автоматическое обнаружение при использовании конфигурации на основе аннотаций и сканирования путей к классам.

Поля и методы:

1. JWT\_TOKEN\_VALIDITY — время жизни токена. По умолчанию установлено 15 минут.
2. secret — соль для генерации токена.
3. validateToken — метод, проверяющий токен.
4. getSubjectFromToken — возвращает зашифрованные данные, извлеченные из токена (в нашем случае — id пользователя)
5. getExpirationDateFromToken — возвращает дату истечения токена.
6. getClaimFromToken — возвращает утверждение, тип которого указан в аргументе по токену.
7. getAllClaimsFromToken — возвращает все утверждения по токену.
8. isTokenExpired — указывает истек ли срок токена.
9. generateToken — генерирует токен из пользовательских данных.
10. doGenerateToken — генерирует токен из строки subject.

**3.6 Класс UserDetailsService**

Основной интерфейс, который загружает пользовательские данные.

* Создать класс UserDetailsServiceImpl в пакете security

@Service

public class UserDetailsServiceImpl implements UserDetailsService {

private final UserCrudService userCrudService;

public UserDetailsServiceImpl(UserCrudService userCrudService){

this.userCrudService = userCrudService;

}

public UserDetails loadUserById(UUID id){

return fromEntity(userCrudService.getById(id));

}

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

return fromEntity(userCrudService.getByUsername(username));

}

UserDetails fromEntity(User user){

boolean enabled = user.isEnabled();

return new org.springframework.security.core.userdetails.User(

user.getEmail(),

user.getPassword(),

enabled,enabled,enabled,enabled,

grantedAuthorities(user)

);

}

public static List<GrantedAuthority> grantedAuthorities(User user){

return Collections.singletonList(new SimpleGrantedAuthority(

"ROLE\_" + user.getRole().getName())

);

}

}

Интерфейс описывает только один метод loadUserByUsername — возвращение данных пользователя в объекте, реализующем UserDetails. В данном случае мы извлекаем пользовательскую информацию из базы данных с помощью UserRepository, инициализируем объект User из пакета org.springframework.security.core.userdetails.

Добавляем собственный метод для получения пользователя по его идентификатору — loadByUserId(UUID id);

Интерфейс GrantedAuthority — представляет полномочия, предоставленные Authentication объекту.

Для поиска пользователя по email нужно добавить соответствующий метод в UserRepository:

public interface UserRepository extends JpaRepository<User, UUID> {

Optional<User> findUserByEmail(String email);

}

**3.7 Класс JwtRequestFilter**

Давайте сначала разберемся, как работают фильтры. Фильтр можно вызывать либо до, либо после выполнения сервлета [*.*](https://www-baeldung-com.translate.goog/spring-boot-add-filter?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) Когда запрос отправляется сервлету, RequestDispatcher может перенаправить его другому сервлету. Есть вероятность, что другой сервлет также имеет такой же фильтр. В таких сценариях один и тот же фильтр вызывается несколько раз.

Но мы можем захотеть, чтобы конкретный фильтр вызывался только один раз для каждого запроса. Обычный вариант использования — при работе с Spring Security. Когда запрос проходит через цепочку фильтров, мы можем захотеть, чтобы некоторые действия аутентификации выполнялись только один раз для запроса.

В таких ситуациях мы можем расширить OncePerRequestFilter. Spring гарантирует, что OncePerRequestFilter выполняется только один раз для данного запроса.

@Component

public class JwtRequestFilter extends OncePerRequestFilter {

private final TokenParserFromRequest tokenParserFromRequest;

private final JwtTokenService jwtTokenService;

private final UserDetailsServiceImpl userDetailsService;

@Autowired

public JwtRequestFilter(TokenParserFromRequest tokenParserFromRequest,

JwtTokenService jwtTokenService, UserDetailsServiceImpl userDetailsService) {

this.tokenParserFromRequest = tokenParserFromRequest;

this.jwtTokenService = jwtTokenService;

this.userDetailsService = userDetailsService;

}

@Override

protected void doFilterInternal(HttpServletRequest request,

HttpServletResponse response,

FilterChain chain)

throws ServletException, IOException {

String jwtToken = tryGetToken(request);

try {

tryAuthenticate(request,jwtToken);

} catch (IllegalArgumentException e) {

System.out.println("Unable to get JWT Token");

} catch (ExpiredJwtException e) {

System.out.println("JWT Token has expired");

System.out.println("Refreshing token");

tryAuthenticate(request,

jwtTokenService

.refreshToken(jwtToken)

.getValue());

}

chain.doFilter(request, response);

}

private void tryAuthenticate(HttpServletRequest request, String jwtToken){

UUID userId = jwtTokenService.getIdFromToken(jwtToken);

if (SecurityContextHolder.getContext().getAuthentication() == null

&& jwtTokenService.tokenExists(jwtToken)) {

if (jwtTokenService.validateToken(jwtToken, userId)) {

authenticate(

userDetailsService.loadUserById(userId),

request

);

System.out.println("in filter: " + SecurityContextHolder.getContext().getAuthentication());

}

}

}

private String tryGetToken(HttpServletRequest request){

String jwtToken;

try {

jwtToken = tokenParserFromRequest.parse(request);

}

catch (IllegalArgumentException e){

jwtToken = null;

}

return jwtToken;

}

private void authenticate(UserDetails user, HttpServletRequest request){

UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken =

new UsernamePasswordAuthenticationToken(

user,

null,

user.getAuthorities()

);

usernamePasswordAuthenticationToken.setDetails(

new WebAuthenticationDetailsSource().buildDetails(request)

);

SecurityContextHolder.getContext()

.setAuthentication(usernamePasswordAuthenticationToken);

}

}

HttpServletRequest — объект, с помощью которого можно получить переданные в запросе данные. В данном случае мы получаем содержимое заголовка «Authorization», в котором находятся ключевое слово Bearerс токеном, разделенные пробелом.

Если пользователь не вошел в систему, то извлеченный username из токена используется для получения UserDetails из соответствующего сервисного объекта.

Если токен проходит валидацию, то мы создаем токен авторизации и инициализируем аутентификацию в контексте Spring Security.

Иначе — пользователь не проходит аутентификацию и получает код 401.

**3.8 TokenParserFromRequest**

Утилитарный класс для получения токена в виде строки из хедера запроса.

@Service

public class TokenParserFromRequest {

private final JwtUserIdTokenUtil jwtUserIdTokenUtil;

@Autowired

public TokenParserFromRequest(JwtUserIdTokenUtil jwtUserIdTokenUtil) {

this.jwtUserIdTokenUtil = jwtUserIdTokenUtil;

}

public String parse(HttpServletRequest request){

final String requestTokenHeader = request.getHeader("Authorization");

if (requestTokenHeader == null || !requestTokenHeader.startsWith("Bearer ")) {

throw new IllegalArgumentException("Authorization token not found");

}

return requestTokenHeader.split(" ")[1];

}

}

**3.9 JwtAuthenticationEntryPoint**

AuthenticationEntryPoint — модифицирует ответ, чтобы дать понять клиенту что необходима аутентификация (заголовки, редирект на страницу логина, т.п.)

@Component

public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint, Serializable {

@Override

public void commence(HttpServletRequest request, HttpServletResponse response,

AuthenticationException authException) throws IOException {

response.sendError(HttpServletResponse.SC\_UNAUTHORIZED, "Unauthorized");

}

}

При неверных данных аутентификации в ответ на запрос будет отправлен код ошибки 401.

**3.10 WebSecurityConfig**

Объект конфигурации защиты ресурсов, выхода из системы, обработки неавторизованных запросов.

@Configuration

public class WebSecurityConfig {

private final UserDetailsService userDetailsService;

private final JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;

private final JwtRequestFilter jwtRequestFilter;

@Autowired

public WebSecurityConfig(UserDetailsService userDetailsService,

JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint,

JwtRequestFilter jwtRequestFilter) {

this.userDetailsService = userDetailsService;

this.jwtAuthenticationEntryPoint = jwtAuthenticationEntryPoint;

this.jwtRequestFilter = jwtRequestFilter;

}

@Bean

SecurityFilterChain web(HttpSecurity http, SuccessLogoutHandlerImpl successLogoutHandler) throws Exception {

http

.csrf()

.disable()

.authorizeHttpRequests(

(authorize) -> authorize

.antMatchers("/auth/login", "/auth/reg")

.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()

.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);

return http.build();

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder(12);

}

@Bean

AuthenticationManager authenticationManagerBean(AuthenticationConfiguration authenticationConfiguration)

throws Exception {

return authenticationConfiguration.getAuthenticationManager();

}

}

CSRF — «межсайтовая подделка запроса», также известна как XSRF— вид атак на посетителей веб-сайтов, использующий недостатки протокола HTTP. Если жертва заходит на сайт, созданный злоумышленником, от её лица тайно отправляется запрос на другой сервер (например, на сервер платёжной системы), осуществляющий некую вредоносную операцию (например, перевод денег на счёт злоумышленника). Для осуществления данной атаки жертва должна быть аутентифицирована на том сервере, на который отправляется запрос, и этот запрос не должен требовать какого-либо подтверждения со стороны пользователя, которое не может быть проигнорировано или подделано атакующим скриптом.

*.csrf().disable()* - отключение защиты, пока что она нам не пригодится.

*.authorizeHttpRequests* — метод конфигурации доступа к маршрутам.

Для всех дочерних маршрутов маршрута «*/auth/reg», «/auth/login»* указан доступ для любого клиента. Остальные запросы доступны только после авторизации.

*.logout* — метод конфигурации выхода из системы.

**3.11 SuccessLogoutHandlerImpl**

Описываем поведение после удачной попытки выйти из системы и инжектим этот класс в аргументы метода

web(HttpSecurity http, SuccessLogoutHandlerImpl successLogoutHandler)

@Component

public class SuccessLogoutHandlerImpl implements LogoutSuccessHandler {

private final LogoutService logoutService;

@Autowired

public SuccessLogoutHandlerImpl(LogoutService logoutService) {

this.logoutService = logoutService;

}

@Override

public void onLogoutSuccess(HttpServletRequest request,

HttpServletResponse response,

Authentication authentication){

logoutService.logout(request);

}

}

**3.12 PasswordEncoderImpl**

Для дополнительной защиты добавляем в конец пароля соль.

@Component

public class PasswordEncoderImpl implements PasswordEncoder {

private final BCryptPasswordEncoder bCryptPasswordEncoder;

@Value("${jwt.secret}")

private String salt;

public PasswordEncoderImpl() {

this.bCryptPasswordEncoder = new BCryptPasswordEncoder(12);

}

@Override

public String encode(CharSequence rawPassword) {

return bCryptPasswordEncoder.encode(rawPassword+salt);

}

@Override

public boolean matches(CharSequence rawPassword, String encodedPassword) {

return bCryptPasswordEncoder.matches(rawPassword+salt,encodedPassword);

}

}

**3.13 DTO**

Data Transfer Object **-** один из шаблонов проектирования, используется для передачи данных между подсистемами приложения.

DTO для передачи данных пользователя при авторизации:

@Getter

@Setter

@NoArgsConstructor

@AllArgsConstructor

public class AuthDto implements Serializable {

private String email;

private String password;

}

DTO для передачи данных при регистрации:

@Getter

@Setter

@NoArgsConstructor

@AllArgsConstructor

public class RegDto extends AuthDto {

private String firstname;

private String lastname;

@JsonProperty(access = JsonProperty.Access.READ\_ONLY)

private UUID id;

}

DTO для передачи данных при обновлении:

@Getter

@Setter

public class UpdateDto implements Serializable {

private String firstname;

private String lastname;

private String email;

@JsonProperty(access = JsonProperty.Access.WRITE\_ONLY)

private String password;

}

DTO для передачи информации о пользователе:

@Getter

@Setter

@AllArgsConstructor

@NoArgsConstructor

public class UserInfo implements Serializable {

private UUID id;

private String firstname;

private String lastname;

private String email;

}

**3.14 Сущности**

14.1 User

Чтобы выполнить регистрацию/авторизацию, пользователь должен указать email и пароль. Т.к. у нашей сущности User отсутствуют соответствующие поля, нам придется изменить ее:

@Getter

@Setter

@Entity

@Table(name = "users")

public class User implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = "id", nullable = false)

private UUID id;

@Column(nullable = false, length = 25,name = "first\_name")

private String firstname;

@Column(nullable = false, length = 25, name = "last\_name")

private String lastname;

@JsonProperty(access = JsonProperty.Access.READ\_ONLY)

@ManyToOne

@JoinColumn(name = "role")

private Role role;

@Column(nullable = false, unique = true)

private String email;

@Column(nullable = false)

private String password;

@JsonIgnore

private boolean enabled = true;

}

*@JsonIgnore* — аннотация, скрывающее поле от изменения при передачи объекта User внутри JSON.

Поле *enabled* указывает на признак входа пользователя.

14.2 Token

Для отслеживания входа/выхода из системы токены придется сохранять в базу данных:

@Getter

@Setter

@Entity(name = "token")

@Table(name = "tokens")

public class Token implements Serializable {

@Column(nullable = false, unique = true)

private String value;

@JsonIgnore

private boolean killed;

@Id

@Column(name = "user\_id")

private UUID userId;

}

**3.15 TokenRepository**

Для взаимодействия с базой данных без репозитория не обойтись:

public interface TokenRepository extends JpaRepository<Token, UUID> {

Optional<Token> findByValue(String value);

Optional<Token> findTokenByUserId(UUID id);

}

**3.16 JwtTokenService**

Для гибкого контроля хранения, обновления, запроса токенов объявим сервисный объект:

@Service

public class JwtTokenService {

private final JwtTokenUtil<UUID> jwtUserIdTokenUtil;

private final TokenRepository tokenRepository;

@Autowired

public JwtTokenService(JwtTokenUtil<UUID> jwtUserIdTokenUtil,

TokenRepository tokenRepository) {

this.jwtUserIdTokenUtil = jwtUserIdTokenUtil;

this.tokenRepository = tokenRepository;

}

public Token createToken(UUID userId){

String token = jwtUserIdTokenUtil.generateToken(userId);

Token tokenEntity = new Token();

tokenEntity.setUserId(userId);

tokenEntity.setValue(token);

tokenEntity.setKilled(false);

return tokenRepository.save(tokenEntity);

}

public void kill(String tokenValue){

Token token = tokenByValue(tokenValue);

token.setKilled(true);

tokenRepository.save(token);

}

public void kill(Token token){

token.setKilled(true);

tokenRepository.save(token);

}

public boolean tokenIsKilled(String tokenValue){

return tokenByValue(tokenValue).isKilled();

}

public boolean tokenExists(String tokenValue){

AtomicBoolean exists = new AtomicBoolean(false);

tokenRepository.findByValue(tokenValue).ifPresent(

(t)-> exists.set(!t.isKilled())

);

return exists.get();

}

private Token tokenByValue(String val){

Token token = tokenRepository.findByValue(val).orElseThrow(()->{

throw new EntityExistsException("token with value: '" + val + "' doesn't exists");

});

return token;

}

public Optional<Token> findOptionalByUserId(UUID id){

return tokenRepository.findTokenByUserId(id);

}

public Token findByUserId(UUID id){

return tokenRepository.findTokenByUserId(id).orElseThrow(()->{

throw new EntityExistsException("token by user id: " + id + " doesn't exists");

});

}

public boolean validateToken(String token, UUID id){

if(!jwtUserIdTokenUtil.validateToken(token,id)){

if(new Date().before(jwtUserIdTokenUtil.getExpirationDateFromToken(token))){

kill(token);

}

return false;

}

return true;

}

public UUID getIdFromToken(String token){

return jwtUserIdTokenUtil.getSubjectFromToken(token);

}

public Token refreshToken(String tokenValue){

Token token = tokenByValue(tokenValue);

token.setValue(jwtUserIdTokenUtil.generateToken(token.getUserId()));

return tokenRepository.save(token);

}

}

**3.17 UserService**

Чтобы не описывать проверки существования сущностей используя методы репозитория, создадим декоративный объект, служащий оберткой и дополняющий поведением:

@Service

public class UserService implements UserCrudService{

private final UserRepository userRepository;

private final PasswordEncoder passwordEncoder;

private final TokenRepository tokenRepository;

@Autowired

public UserService(UserRepository userRepository,

PasswordEncoder passwordEncoder,

TokenRepository tokenRepository) {

this.userRepository = userRepository;

this.passwordEncoder = passwordEncoder;

this.tokenRepository = tokenRepository;

}

public User create(User user){

user.setPassword(passwordEncoder.encode(user.getPassword()));

return userRepository.save(user);

}

public User update(User user, UUID id){

if(!userRepository.existsById(id)) {

throw new EntityExistsException("User with id:'"+ user.getId() +"' doesn't exists");

}

user.setPassword(passwordEncoder.encode(user.getPassword()));

return userRepository.save(user);

}

@Override

public void removeById(UUID id) {

userRepository.deleteById(id);

}

@Override

public void remove(User user) {

tokenRepository.findTokenByUserId(user.getId()).ifPresent(tokenRepository::delete);

userRepository.delete(user);

}

@Override

public void removeByUsername(String username) {

userRepository.removeByEmail(username);

}

@Override

public List<User> getAll() {

return userRepository.findAll();

}

@Override

public User getById(UUID id) {

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

throw new EntityExistsException("user with id: " + id + " doesn't exists");

});

}

@Override

public User getByUsername(String username) {

return userRepository.findUserByEmail(username).orElseThrow(()->{

throw new UsernameNotFoundException("user with username: " + username

+ " doesn't exists");

});

}

@Override

public boolean existsById(UUID id) {

return userRepository.existsById(id);

}

@Override

public boolean existsByUsername(String username) {

return userRepository.findUserByEmail(username).isPresent();

}

}

Интерфейс описывающий поведение сервисного объекта для произведения CRUD операций:

public interface UserCrudService {

User create(User user);

User update(User user, UUID id);

void removeById(UUID id);

void remove(User user);

void removeByUsername(String username);

List<User> getAll();

User getById(UUID id);

User getByUsername(String username);

boolean existsById(UUID id);

boolean existsByUsername(String username);

}

Для реализации интерфейса в репозиторий следует добавить метод удаления пользователя идентифицируя его по email:

public interface UserRepository extends JpaRepository<User, UUID> {

Optional<User> findUserByEmail(String email);

void removeByEmail(String email);

}

**3.18 UserMeService**

Сервис для получения персональных данных с помощью имени пользователя, который находится в объекте текущей аутентификации:

@Service

public class UserMeService{

private final UserCrudService userCrudService;

@Autowired

public UserMeService(UserCrudService userCrudService) {

this.userCrudService = userCrudService;

}

public UserInfo update(UpdateDto user) {

User userEntity = fromUpdateDto(user);

userEntity = userCrudService.update(userEntity, userEntity.getId());

return toUserInfo(userEntity);

}

public void remove(){

User user = getAuthenticatedUser();

userCrudService.remove(user);

SecurityContextHolder.getContext().setAuthentication(null);

}

public UserInfo get(){

User user = getAuthenticatedUser();

return toUserInfo(user);

}

private User getAuthenticatedUser(){

return userCrudService.getByUsername(SecurityContextHolder

.getContext()

.getAuthentication()

.getName()

);

}

private UserInfo toUserInfo(User user){

return new UserInfo(user.getId(),

user.getFirstname(),

user.getLastname(),

user.getEmail()

);

}

private User fromUpdateDto(UpdateDto updateDto){

User userEntity = new User();

userEntity.setEmail(updateDto.getEmail());

userEntity.setLastname(updateDto.getLastname());

userEntity.setFirstname(updateDto.getFirstname());

userEntity.setPassword(updateDto.getPassword());

userEntity.setId(getAuthenticatedUser().getId());

userEntity.setRole(getAuthenticatedUser().getRole());

userEntity.setEnabled(getAuthenticatedUser().isEnabled());

return userEntity;

}

}

**3.19 Сервисы авторизации, регистрации, выхода**

Сервис регистрации (UserRegService) — получает данные для регистрации на вход и возвращает токен авторизации на выход:

@Service

public class UserRegService {

private final UserCrudService userCrudService;

private final RoleRepository roleRepository;

private final UserAuthService userAuthService;

public UserRegService(UserCrudService userCrudService,

RoleRepository roleRepository,

UserAuthService userAuthService) {

this.userCrudService = userCrudService;

this.roleRepository = roleRepository;

this.userAuthService = userAuthService;

}

public Token registration(RegDto regDto) throws Exception {

if (userCrudService.existsByUsername(regDto.getEmail())) {

throw new EntityExistsException("User with email: " + regDto.getEmail() + " already exists");

}

User user = fromRegDto(regDto);

user.setRole(roleRepository.findById("USER").get());

userCrudService.create(user);

Token token = userAuthService.authorization(

new AuthDto(user.getEmail(), regDto.getPassword())

);

return token;

}

private User fromRegDto(RegDto regDto){

User user = new User();

user.setEnabled(true);

user.setEmail(regDto.getEmail());

user.setFirstname(regDto.getFirstname());

user.setLastname(regDto.getLastname());

user.setPassword(regDto.getPassword());

return user;

}

}

Сервис авторизации (UserAuthService) — принимает на вход данные для авторизации, на выход возвращает токен авторизации:

@Service

public class UserAuthService {

private final UserCrudService userCrudService;

private final AuthenticationManager authenticationManager;

private final UserDetailsService userDetailsService;

private final JwtTokenService jwtTokenService;

@Autowired

public UserAuthService(UserCrudService userCrudService,

AuthenticationManager authenticationManager,

UserDetailsService userDetailsService,

JwtTokenService jwtTokenService) {

this.userCrudService = userCrudService;

this.authenticationManager = authenticationManager;

this.userDetailsService = userDetailsService;

this.jwtTokenService = jwtTokenService;

}

public Token authorization(AuthDto authDto) throws Exception {

authenticate(authDto);

return getToken(authDto.getEmail());

}

private void authenticate(AuthDto data) throws Exception {

try {

UserDetails user = userDetailsService

.loadUserByUsername(data.getEmail());

authenticationManager.authenticate(

new UsernamePasswordAuthenticationToken(

data.getEmail(),

data.getPassword(),

user.getAuthorities()

)

);

System.out.println(SecurityContextHolder.getContext().getAuthentication());

} catch (DisabledException e) {

throw new Exception("USER\_DISABLED", e);

} catch (BadCredentialsException e) {

throw new Exception("INVALID\_CREDENTIALS", e);

}

}

private Token getToken(String username) {

UUID userId = userCrudService.getByUsername(username).getId();

AtomicReference<Token> token = new AtomicReference<>();

jwtTokenService.findOptionalByUserId(userId).ifPresentOrElse(

(t) -> {

if (t.isKilled()) {

token.set(jwtTokenService.createToken(userId));

} else {

token.set(t);

}

},

() -> token.set(jwtTokenService.createToken(userId))

);

return token.get();

}

}

Сервис выхода из системы (LogoutService) — помечает токен авторизации как «убитый», получая либо объект запроса, либо само значение токена в виде строки:

@Service

public class LogoutService {

private final TokenParserFromRequest tokenParserFromRequest;

private final JwtTokenService jwtTokenService;

@Autowired

public LogoutService(TokenParserFromRequest tokenParserFromRequest,

JwtTokenService jwtTokenService) {

this.tokenParserFromRequest = tokenParserFromRequest;

this.jwtTokenService = jwtTokenService;

}

public void logout(HttpServletRequest request){

String token = tokenParserFromRequest.parse(request);

jwtTokenService.kill(token);

}

public void logout(String token){

jwtTokenService.kill(token);

}

}

**3.20 Controllers**

Контроллер авторизации/регистрации:

@RestController

@RequestMapping("/auth")

public class AuthRestController {

private final UserAuthService userAuthService;

private final UserRegService userRegService;

public AuthRestController(UserAuthService userAuthService,

UserRegService userRegService) {

this.userAuthService = userAuthService;

this.userRegService = userRegService;

}

@PostMapping(path = "/login")

@RequestMapping(value = "/login")

public Token createAuthenticationToken(@RequestBody AuthDto authenticationRequest)

throws Exception {

return userAuthService.authorization(authenticationRequest);

}

@PostMapping

@RequestMapping("/reg")

public Token createAccount(@RequestBody RegDto user) throws Exception {

return userRegService.registration(user);

}

}

Контроллер CRUD операций для сущность User:

@RequestMapping("/users")

@RestController

public class UserRestController {

private final UserService userService;

@Autowired

public UserRestController(UserService userService){

this.userService = userService;

}

@GetMapping

public List<User> getAll(){

return userService.getAll();

}

@GetMapping("/{id}")

public User getById(@PathVariable("id") UUID id){

return userService.getById(id);

}

@PutMapping("/{id}")

public User update(User user, @PathVariable("id") UUID id){

return userService.update(user, id);

}

@PostMapping

public User create(User user){

return userService.create(user);

}

@DeleteMapping("/{id}")

public void remove(@PathVariable("id") UUID id){

userService.removeById(id);

}

}

Контроллер для редактирования и просмотра данных текущей авторизации:

@RestController

@RequestMapping("/me")

public class MeRestController {

private final UserMeService userMeService;

@Autowired

public MeRestController(UserMeService userMeService) {

this.userMeService = userMeService;

}

@GetMapping

public UserInfo get(){

return userMeService.get();

}

@PutMapping

public UserInfo update(@RequestBody UpdateDto user){

return userMeService.update(user);

}

@DeleteMapping

public void remove(){

userMeService.remove();

}

}

**3.21 Миграция базы данных**

Добавление скриптов (рис. 3.5).

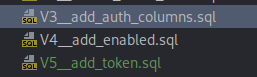


Рис. 3.5 - Скрипты миграции добавления полей аутентификации, enabled; добавления таблицы токенов

Добавление полей аутентификации:

ALTER TABLE users

ADD email varchar(25) NOT NULL UNIQUE,

ADD password varchar(120) NOT NULL;

Добавление поля enabled:

ALTER TABLE users

ADD enabled boolean default true;

Добавление таблицы токенов:

create table if not exists tokens(

user\_id uuid not null,

value varchar(255) not null,

killed bool,

primary key(user\_id),

unique (user\_id),

unique (value)

);

alter table tokens add foreign key (user\_id) references users(id)

**3.22 Salt хэширования**

**Salt (соль, модификатор входа хэш-функции)** — строка данных, которая передаётся хеш-функции вместе с входным массивом данных (прообразом) для вычисления хэша (образа).

Используется для усложнения определения прообраза хэш-функции методом перебора по словарю возможных входных значений (прообразов), включая атаки с использованием радужных таблиц. Позволяет скрыть факт использования одинаковых прообразов при использовании для них разной соли.

Различают статическую соль (одна и та же для всех входных значений) и динамическую (генерируется для каждого входного значения персонально).

3.22.1 Определение строки с Salt

В application.properties добавляем новую строку, смысл которой в хранении «соли» для хэширования пароля и JWT токена: jwt.secret=simple.

Для получения значения переменной нужно:

* описать поле строкового типа, которое будет хранить это значение;
* установить аннотацию @Value со следующим значением (рис. 3.6 и рис. 3.7).

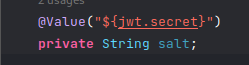


Рис. 3.6 — Пример установки значения переменной из application.properties

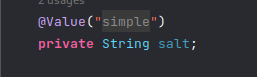


Рис. 3.7 - Пример подсказки значения в переменной после установки аннотации

**3.23 Тестирование маршрутов в Postman**

Во всех запросах нам понадобится один и тот же домен: http://localhost:8080.

Для того, чтобы не писать адрес при создании нового запроса, создадим глобальную переменную в Postman’e (рис. 3.8-3.11).

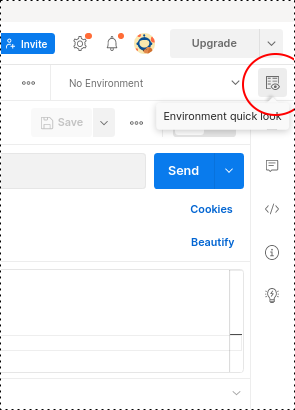


Рис. 3.8 - Вкладка настройки сред (правый верхний угол)

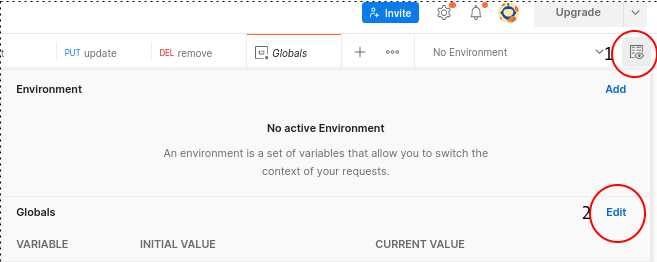


Рис. 3.9 - Вкладка добавления среды и редактирования глобальной

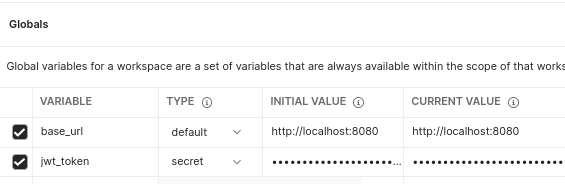


Рис. 3.10 - Создание переменных глобальной среды

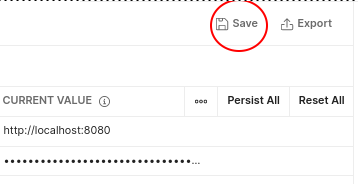


Рис. 3.11 - Сохранение переменных глобальной среды

Получить значение переменной можно указав ее имя, обрамленное двумя фигурными скобками: {{base\_url}}.

3.24.1 Тестирование маршрутов аутентификации

Для визуального удобства можно создать папку в коллекции User и назвать ее «auth» для хранения запросов связанных с аутентификацией (рис. 3.12).

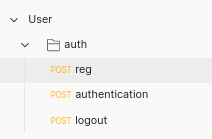


Рис. 3.12 — Создание папки для запросов аутентификации

3.24.2 Регистрация

Для регистрации сервер требует JSON объект в теле запроса (рис. 3.13):

{

"email":"superman@mail.ru",

"firstname":"Roman",

"lastname":"Zhul",

"password":"password"

}

На выход получаем JSON объект токена:

{

"value": "eyJhbGciOiJub25lIn0.eyJzdWIiOiI5NjRlYjRlZC03Y2NjLTQ4NzYtOTE1OS05NWY4ODU5YmEyNjRzaW1wbGUiLCJpYXQiOjE2NjY4MjY0MjMsImV4cCI6MTY2NjkxMjgyM30.",

"userId": "964eb4ed-7ccc-4876-9159-95f8859ba264"

}

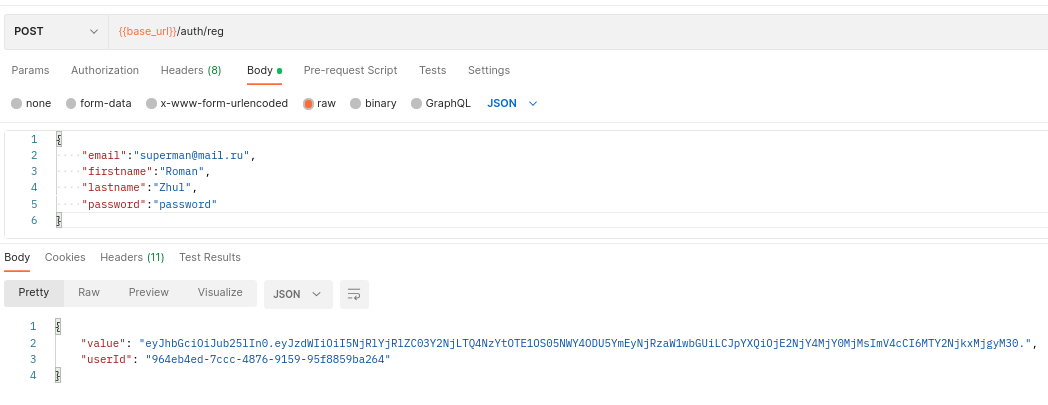


Рис. 3.13 - Регистрация пользователя с получением токена авторизации

3.24.3 Авторизация

При авторизации нужно указать только логин и пароль внутри тела запроса:

{

"email":"superman@mail.ru",

"password":"password"

}

При правильно введенных данных пользователя получим токен и id пользователя как при регистрации (рис. 3.14 и рис. 3.15).

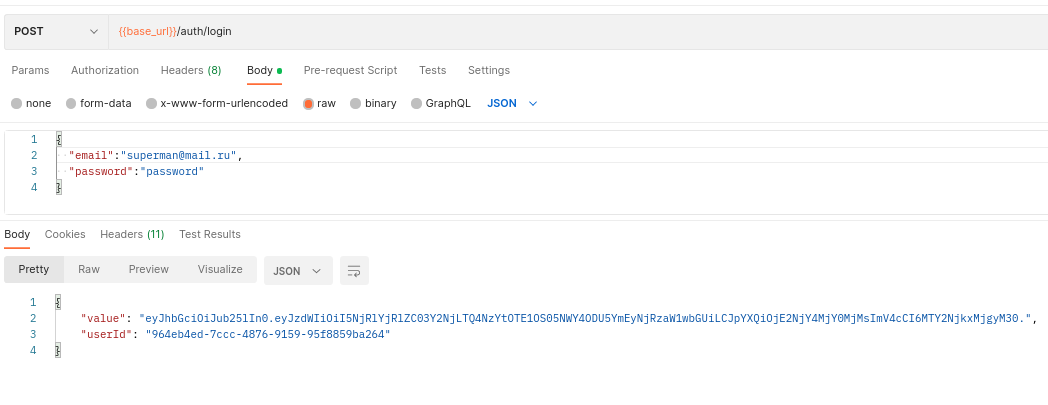


Рис. 3.14 - Ответ на попытку авторизации с корректными данными

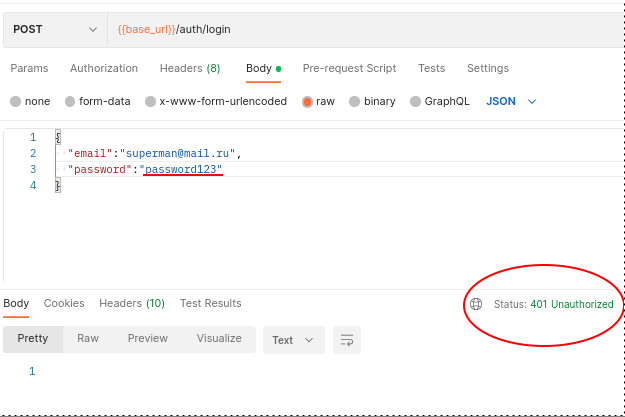


Рис. 3.15 - попытка авторизации с неправильным паролем

3.24.4 Выход из системы

Выход из системы осуществляется простым post запросом по маршруту */auth/logout*

При успешном выходе получаем в ответ код 200, статус ОК (рис. 3.16).

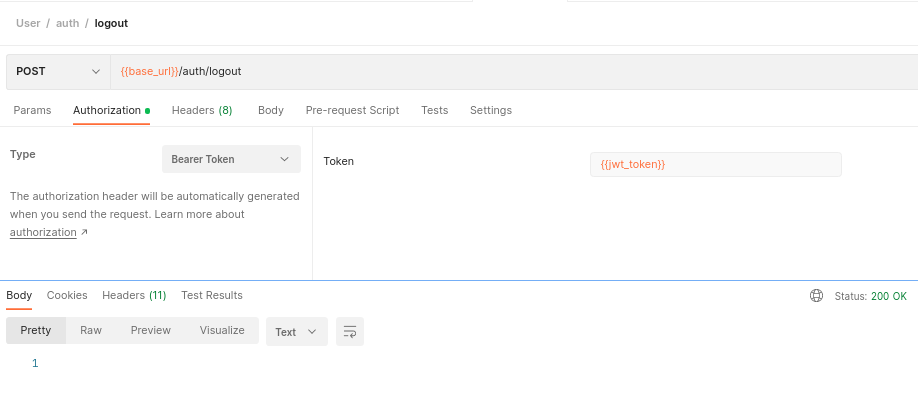


Рис. 3.16 - выход из системы с указанной переменной в методе авторизации

Остальные запросы по методу доступа будут схожи с выходом из системы: потребуется указать токен в методе авторизации.

Можно указать авторизацию на уровне запроса/папки/коллекции, что облегчает создание новых запросов, поскольку не придется дублировать информацию о запрашиваемом домене.

**3.25 Лабораторная работа - Авторизация. Spring Security**

Цель лабораторной работы: ознакомиться с принципом работы Spring Security и контролем доступа.

Задание:

1. Добавьте требующиеся maven зависимости
2. Создайте сущность Token и отредактируйте сущность User как показано на примере выше
3. Добавьте скрипты миграции
4. Опишите сервисные классы
5. Дополните репозиторий пользователя новым методом
6. Опишите репозиторий для сущности Token
7. Опишите Data Transfer Object классы
8. Опишите фильтр авторизации
9. Опишите AuthenticationEntryPoint
10. Опишите класс успешного выхода из системы — SuccessLogoutHandlerImpl
11. Опишите объект хэширования пароля
12. Опишите конфигурацию WebSecurityConfig
13. Отредактируйте контроллер для маршрута /users
14. Добавьте контроллер для маршрута /me, описывающий получения данных авторизованного пользователя
15. Добавьте контроллер авторизации/регистрации/выхода из системы
16. Создайте соответствующие запросы в Postman для тестирования приложения

Задание для индивидуального выполнения:

* Установите собственное значение соли в application.properties
  + обратите внимание, что при изменении соли при имеющихся аутентификационных данных в базе вы потеряете доступ к этим данным. То есть, если вы регистрировались , а затем изменили соль, вы не сможете войти в этот аккаунт.
* Создайте запросы для просмотра и редактирования информации авторизованного пользователя
* Установите следующее разграничение прав доступа для методов UserRestController:
  + getAll, getById — доступны для всех;
  + update, create, remove - только для модераторов и администраторов

Контрольные вопросы:

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