Отчет по лабораторной работе 3 - Авторизация

Цель работы

Основной целью работы была реализация системы авторизации пользователей с использованием JWT-токенов, а также обеспечение безопасности API. В рамках работы были выполнены задачи по настройке аутентификации, защиты эндпоинтов, управления пользователями и тестирования API.

Реализация механизма аутентификации

Для аутентификации пользователей был реализован следующий функционал:

- **Регистрация нового пользователя**:
  Эндпоинт `/auth/register` принимает данные пользователя (email и пароль), сохраняет их в базу данных, предварительно хешируя пароль с помощью `BCryptPasswordEncoder`. Это обеспечивает безопасное хранение паролей.
- **Вход в систему**:
  Эндпоинт `/auth/login` проверяет переданные email и пароль, используя `AuthenticationManager`. После успешной аутентификации генерируются два JWT-токена: access-токен (срок действия 30 минут) и refresh-токен (срок действия 7 дней). Access-токен используется для доступа к защищенным ресурсам, а refresh-токен — для обновления access-токена без повторного ввода пароля.

```
@Service
@RequiredArgsConstructor
public class AuthService implements UserDetailsService {

  private final UserRepository userRepository;
  private final TokenBlacklistService tokenBlacklistService;
  private final JwtService jwtService;

  @Override
  public UserDetails loadUserByUsername(String username)
    throws UsernameNotFoundException {
    Optional<User> userDetail = userRepository.findByEmail(username);

      .map(UserInfoDetails::new)
      .orElseThrow(() ->
```

```
        new UsernameNotFoundException("User not found: " + username)
      );
  }

  public User getCurrentUser() {
    Object principal = SecurityContextHolder.getContext()
      .getAuthentication()
      .getPrincipal();

    if (principal instanceof UserInfoDetails) {
      UserInfoDetails userInfoDetails = (UserInfoDetails) principal;
      return userRepository
        .findById(userInfoDetails.getId())
        .orElseThrow(() -> new IllegalArgumentException("User not found"));
    } else if (principal instanceof String) {
      // Если principal - это строка (email), загружаем пользователя из базы
      return userRepository
        .findByEmail((String) principal)
        .orElseThrow(() -> new IllegalArgumentException("User not found"));
    }
    throw new IllegalStateException("Unexpected principal type");
  }

  public Map<String, String> refreshTokens(String refreshToken) {
    validateRefreshToken(refreshToken);

    String email = jwtService.extractUsername(refreshToken);

    User user = userRepository
      .findByEmail(email)
      .orElseThrow(() -> new UsernameNotFoundException("User not found"));

    String newAccessToken = jwtService.generateAccessToken(email);
    String newRefreshToken = jwtService.generateRefreshToken(email);

    tokenBlacklistService.addTokenToBlacklist(
      refreshToken,
      jwtService.extractExpiration(refreshToken).getTime()
    );

    return Map.of(
      "accessToken",
      newAccessToken,
      "refreshToken",
      newRefreshToken
    );
```

```
  }

  private void validateRefreshToken(String refreshToken) {
    if (refreshToken == null || refreshToken.isBlank()) {
      throw new IllegalArgumentException("Refresh token is empty");
    }

    if (!"REFRESH".equals(jwtService.extractTokenType(refreshToken))) {
      throw new UnauthorizedException("Invalid token type");
    }

    if (!jwtService.validateToken(refreshToken)) {
      throw new UnauthorizedException("Invalid or expired refresh token");
    }

    if (tokenBlacklistService.isTokenBlacklisted(refreshToken)) {
      throw new UnauthorizedException("Refresh token was revoked");
    }
  }

  public void logout(String token) {
    if (token != null && !token.isBlank()) {

      try {
        String username = jwtService.extractUsername(token);

        long expirationTime = jwtService.extractExpiration(token).getTime();
        tokenBlacklistService.addTokenToBlacklist(token, expirationTime);
      } catch (Exception e) {

      }
    }

    SecurityContextHolder.clearContext();
  }
}
```

Настройка middleware для защиты API

Для защиты API была настроена Spring Security:

- **JWT-фильтр (`JwtAuthFilter`)**
  Этот фильтр проверяет наличие и валидность JWT-токена в заголовке `Authorization`

каждого запроса. Если токен валиден, пользователь аутентифицируется, и его данные сохраняются в контексте безопасности. Фильтр также проверяет, не находится ли токен в черном списке (blacklist), что позволяет отзывать токены при выходе из системы.

- **Настройка доступа к эндпоинтам**
  В `WebSecurityConfig` определены правила доступа:
  - Публичные эндпоинты (`/auth/register` , `/auth/login`) доступны без аутентификации.
  - Доступ к большенству эндпоинтов настраиваться отдельно через `@PreAuthorize("hasAuthority('admin')"` .
  - Все остальные эндпоинты открыты (`.anyRequest().permitAll()`).

```
@Component
@RequiredArgsConstructor
public class JwtAuthFilter extends OncePerRequestFilter {

  @Lazy
  private final UserDetailsService userDetailsService;

  private final TokenBlacklistService tokenBlacklistService;
  private final JwtService jwtService;

  @Override
  protected void doFilterInternal(
    HttpServletRequest request,
    HttpServletResponse response,
    FilterChain filterChain
  ) throws ServletException, IOException {
    String authHeader = request.getHeader("Authorization");
    String token = null;
    String username = null;
    String tokenType = null;

    if (authHeader != null && authHeader.startsWith("Bearer ")) {
      token = authHeader.substring(7);
      username = jwtService.extractUsername(token);
      tokenType = jwtService.extractTokenType(token);
    }

    if (
      username != null &&
      SecurityContextHolder.getContext().getAuthentication() == null &&
      tokenType.equals("accessToken") &&
```

```
      !tokenBlacklistService.isTokenBlacklisted(token)
    ) {
      UserDetails userDetails =
userDetailsService.loadUserByUsername(username);
      if (jwtService.validateToken(token, userDetails.getUsername())) {
        UsernamePasswordAuthenticationToken authToken =
          new UsernamePasswordAuthenticationToken(
            userDetails, // Важно передать userDetails, а не username
            null,
            userDetails.getAuthorities()
          );
        authToken.setDetails(
          new WebAuthenticationDetailsSource().buildDetails(request)
        );
        SecurityContextHolder.getContext().setAuthentication(authToken);
      }
    }
    filterChain.doFilter(request, response);
  }
}
```

Реализация управления пользователями

- **Выход из системы (logout)**
  При выходе из системы access-токен добавляется в черный список (`TokenBlacklistService`), что предотвращает его дальнейшее использование. Черный список регулярно очищается от истекших токенов для оптимизации памяти.
- **Обновление токенов**
  Эндпоинт `/auth/refresh` принимает refresh-токен, проверяет его валидность и выдает новую пару access- и refresh-токенов. Старый refresh-токен также добавляется в черный список.

Тестирование API

Тестирование проводилось с помощью Postman. Были проверены следующие сценарии:

1. **Регистрация и вход**:
   - Успешная регистрация нового пользователя.
   - Получение токенов после входа.

2. **Доступ к защищенным эндпоинтам**:
   - Доступ к `/users` с ролью `admin`.
   - Отказ в доступе при отсутствии токена или невалидном токене.
3. **Обновление токенов**:
   - Успешное обновление токенов с использованием refresh-токена.
   - Отказ при использовании истекшего или невалидного refresh-токена.
4. **Выход из системы**:
   - Проверка, что токен после выхода больше недействителен.

Примеры HTTP-ответов:

- `200 OK` — успешная аутентификация или доступ к ресурсу.
- `401 Unauthorized` — невалидный или отсутствующий токен.
- `403 Forbidden` — недостаточно прав для доступа (например, отсутствие роли `admin`).

JWT-аутентификации

- **JWT-аутентификация**:
  Токены содержат следующие данные:
  - `email` пользователя (в качестве subject).
  - `tokenType` (различает access- и refresh-токены).
  - Срок действия (expiration time).

Токены подписываются с использованием секретного ключа, что предотвращает их подделку.

```
@Component
public class JwtService {

  private static final long ACCESS_TOKEN_EXPIRATION = 1000 * 60 * 30;
  private static final long REFRESH_TOKEN_EXPIRATION = 1000 * 60 * 60 * 24 *
7;

  private final String secret;

  public JwtService(@Value("${jwt.secret}") String secret) {
    this.secret = secret;
  }
```

```
  public String generateAccessToken(String email) {
    Map<String, Object> claims = new HashMap<>();
    claims.put("tokenType", "accessToken");
    return createToken(claims, email, ACCESS_TOKEN_EXPIRATION);
  }

  public String generateRefreshToken(String email) {
    Map<String, Object> claims = new HashMap<>();
    claims.put("tokenType", "refreshToken");
    return createToken(claims, email, REFRESH_TOKEN_EXPIRATION);
  }

  private String createToken(
    Map<String, Object> claims,
    String email,
    long expiration
  ) {
    return Jwts.builder()
      .setClaims(claims)
      .setSubject(email)
      .setIssuedAt(new Date())
      .setExpiration(new Date(System.currentTimeMillis() + expiration))
      .signWith(getSignKey(), SignatureAlgorithm.HS256)
      .compact();
  }

  private Key getSignKey() {
    byte[] keyBytes = Decoders.BASE64.decode(secret);
    return Keys.hmacShaKeyFor(keyBytes);
  }

  public String extractUsername(String token) {
    return extractClaim(token, Claims::getSubject);
  }

  public String extractTokenType(String token) {
    return extractClaim(token, claims ->
claims.get("tokenType").toString());
  }

  public Date extractExpiration(String token) {
    return extractClaim(token, Claims::getExpiration);
  }

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

```
    final Claims claims = extractAllClaims(token);
    return claimsResolver.apply(claims);
  }

  private Claims extractAllClaims(String token) {
    return Jwts.parserBuilder()
      .setSigningKey(getSignKey())
      .build()
      .parseClaimsJws(token)
      .getBody();
  }

  private Boolean isTokenExpired(String token) {
    return extractExpiration(token).before(new Date());
  }

  public Boolean validateToken(String token, String username) {
    final String usernameToken = extractUsername(token);
    return (usernameToken.equals(username) && !isTokenExpired(token));
  }

  public Boolean validateToken(String token) {
    return !isTokenExpired(token);
  }
}
```