***Spring Security*** – модуль *Spring*, использующийся для аутентификации, авторизации и обеспечения безопасности приложения. Документацию можно найти [тут](https://spring.io/projects/spring-security#learn).

**Аутентификация** – это процесс проверки личности того, кто пытается получить доступ к ресурсу.

**Авторизация** – это процесс проверки прав доступа к запрашиваемому ресурсу.

Для подключения модулей *Spring Security* достаточно добавить его стартер ***spring-boot-starter-security***.

*<dependency>*

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

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

*</dependency>*

*<dependency>*

*<groupId>org.springframework.security</groupId>*

*<artifactId>spring-security-test</artifactId> // для тестирования*

*</dependency>*

Сразу после этого, при доступе к любой странице будет требоваться аутентификация на базовой странице входа. По умолчанию имя пользователя – *user*, а пароль будет сгенерирован и выведен в консоль приложения. Никаких особых ролей или полномочий не предусматривается.

*Spring Security* имеет встроенную защиту от подделки межсайтовых запросов (***Cross-Site Request Forgery***). Для включения *CORS* для всего проекта, нужно создать класс конфигурации с аннотацией ***@Configuration***, реализующий интерфейс *WebMvcConfigurer* и переопределить метод *addCorsMappings*:

*@Configuration*

*public class WebMvcConfig implements WebMvcConfigurer {*

*@Override*

*public void addCorsMappings(CorsRegistry registry) {*

*registry.addMapping("/api/\*\*") // разрешить обращения по пути*

*.allowedOrigins("\*") // разрешить доступ со всех сайтов*

*.allowedMethods("\*"); } } // разрешить все http методы*

***JSON Web Token (JWT)*** – это специальный формат токена, который позволяет безопасно передавать данные между клиентом и сервером. *JWT*-токен состоит из трех частей, которые разделены точкой:

* ***Header*** – заголовок с информацией о токене, типе токена и алгоритме шифрования
* ***Payload*** – полезные данные в формате *JSON*, которые мы хотим передать в токене. Например, имя пользователя, его роль, даты выпуска и истечения токена. Присутствуют зарезервированные имена.
* ***Signature*** – подпись токена, которая позволяет проверить, что токен не был изменен.

Для представления токена в виде компактной строки используется ***Base64Url*** кодирование.

Подробнее можно почитать [тут](https://jwt.io/). Также там можно найти библиотеки с реализацией *JWT* для разных языков. Рекомендовано использовать <https://github.com/jwtk/jjwt>

*<dependency>*

*<groupId>io.jsonwebtoken</groupId>*

*<artifactId>jjwt-api</artifactId>*

*<version>0.12.3</version>*

*</dependency>*

*<dependency>*

*<groupId>io.jsonwebtoken</groupId>*

*<artifactId>jjwt-impl</artifactId>*

*<version>0.12.3</version>*

*<scope>runtime</scope>*

*</dependency>*

*<dependency>*

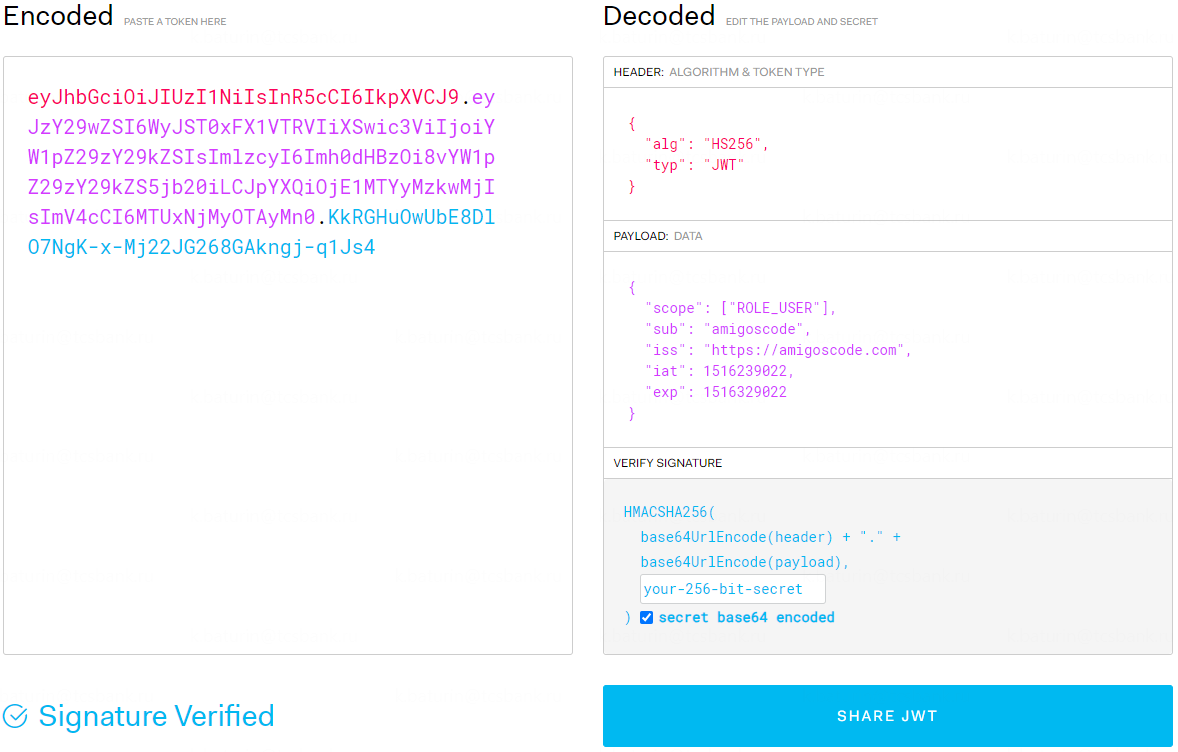
*<groupId>io.jsonwebtoken</groupId>*

*<artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson if Gson is preferred -->*

*<version>0.12.3</version>*

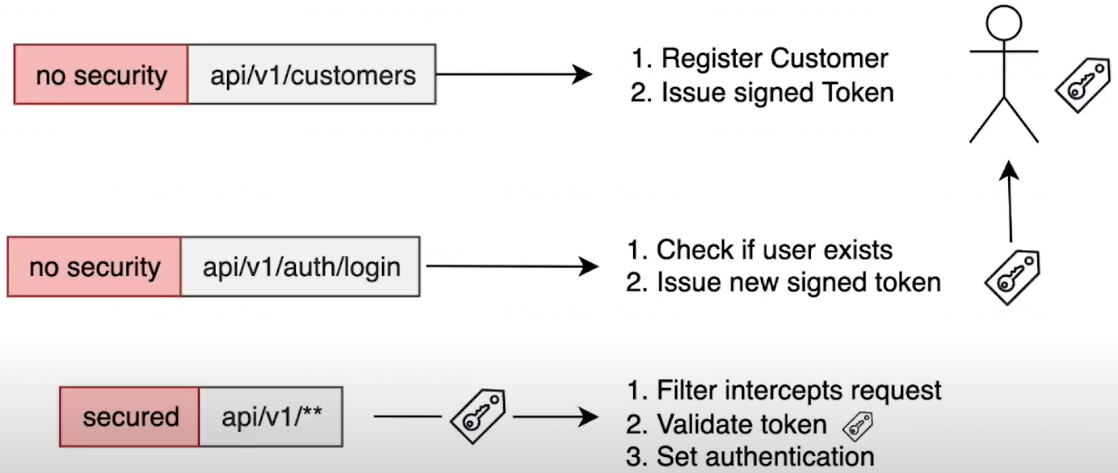
*<scope>runtime</scope>*

*</dependency>*



Логика работы с *JWT* токеном:

* для пользователей без токена доступны только эндпоинт регистрации нового пользователя и эндпоинт авторизации, которые могут выпускать токен
* остальные ресурсы недоступны без корректного токена



Рассмотрим пример создания *JWT* токена.

*@Service*

*public class JWTUtil {*

*// секретная фраза*

*private static final String SECRET\_KEY = "foobar\_123456789 ";*

*// переводим фразу в байт-код и создаем ключ зашифрованный алгоритмом HMAC-SHA*

*private SecretKey getSigningKey() {*

*return Keys.hmacShaKeyFor(SECRET\_KEY.getBytes()); } }*

*// на основании полученных данных выпускаем и отдаем токен*

*public String issueToken(String subject) {*

*return issueToken(subject, Map.of()); }*

*public String issueToken(String subject, String ...scopes) {*

*return issueToken(subject, Map.of("scopes", scopes)); }*

*public String issueToken(String subject, Map<String, Object> claims) {*

*return Jwts.builder()*

*.claims(claims)*

*.subject(subject)*

*.issuer("http://straigt.com")*

*.issuedAt(Date.from(Instant.now()))*

*.expiration(Date.from(Instant.now().plus(15, ChronoUnit.DAYS)))*

*// алгоритм сигнатуры подписи определится автоматически в зависимости от длины ключа*

*.signWith(getSigningKey())*

*.compact();*

*private Claims getClaims(String token) { // декордирование токена и извлечение данных*

*return Jwts.parser()*

*.verifyWith(getSigningKey())*

*.build()*

*.parseSignedClaims(token)*

*.getPayload(); }*

*public String getSubject(String token) {*

*return getClaims(token).getSubject(); }*

*private boolean isTokenExpired(String jwt) { // проверка срока годности*

*Date today = Date.from(Instant.now());*

*return getClaims(jwt).getExpiration().before(today); }*

*public boolean isTokenValid(String jwt, String username) { // валидация токена*

*String subject = getSubject(jwt);*

*return subject.equals(username) && !isTokenExpired(jwt); } }*

Создадим эндпоинт контроллера, который будет создавать пользователя и возвращать токен в заголовке.

*@RestController*

*@RequestMapping("api/v1/customers")*

*@RequiredArgsConstructor*

*public class CustomerController {*

*private final CustomerService customerService;*

*private final JWTUtil jwtUtil;*

*@PostMapping*

*public ResponseEntity<?> registerCustomer(@RequestBody CustomerRegistrationRequest request) {*

*customerService.addCustomer(request);*

*String jwtToken = jwtUtil.issueToken(request.email(), "ROLE\_USER"); // имя УЗ и её роли*

*return ResponseEntity.ok()*

*.header(HttpHeaders.AUTHORIZATION, jwtToken)*

*.build(); }*

Создадим реализацию интерфейса ***SecurityFilterChain*** для защиты эндпоинтов от несанкционированного доступа.

*@Configuration*

*@EnableWebSecurity // отключаем дефолтную конфигурацию webSecurity*

*public class SecurityFilterChainConfig {*

*@Bean*

*public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {*

*http*

*// отключаем проверку CSRF для HTML-форм*

*.csrf(csrf -> csrf.disable())*

*// включаем наш кастомный CORS*

*.cors(Customizer.withDefaults())*

*// включаем ограничения по ролям и привилегиям*

*.authorizeHttpRequests(auth -> auth*

*// разрешить неавторизованные POST запросы на указанные эндпоинты*

*.requestMatchers(HttpMethod.POST, "/api/v1/customers", "/api/v1/login").permitAll()*

*// разрешить неавторизованные GET запросы на указанные эндпоинты*

*.requestMatchers(HttpMethod.GET, "/ping", “/actuator/\*\*”).permitAll()*

*// любые другие запросы разрешены только для аутентифицированных пользователей*

*.anyRequest().authenticated());*

*// не запоминаем состояние для сессии (отключаем сеансы)*

*.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))*

*// внедряем свою реализацию провайдера*

*.authenticationProvider(authenticationProvider)*

*// внедряем свой фильтр проверки токена до фильтра авторизации по имени/паролю*

*.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);*

*// внедряем свой фильтр проброса исключений*

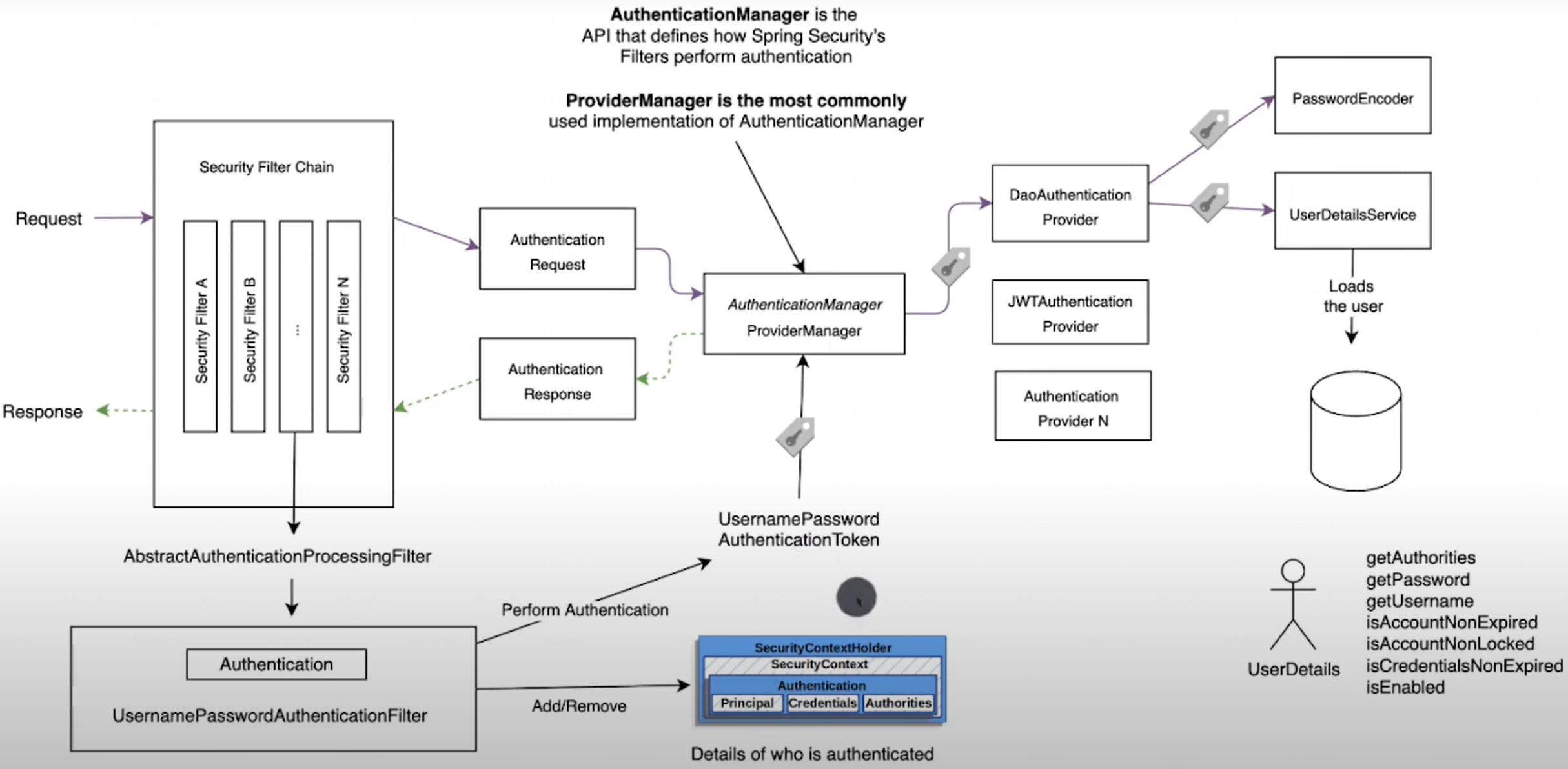
*.exceptionHandling(exception -> exception.authenticationEntryPoint(authenticationEntryPoint));*

*return http.build(); } }*

Билдер ***HttpSecurity*** позволяет:

* Потребовать выполнения определенных условий безопасности перед обслуживанием запроса.
* Отправить пользователю свою страницу входа.
* Предоставить пользователям возможность выйти из приложения.
* Настроить защиту от подделки межсайтовых запросов.

С помощью методов аналогичных ***authorizeRequests()*** можно задать пути и шаблоны *URL*, а также требования безопасности для этих путей. Правила безопасности, объявленные первыми, имеют приоритет над объявленными ниже. Они могут быть написаны также на языке *SpEL* с помощью метода ***access(String codeBySpel)***. По умолчанию после успешного входа пользователь будет перенаправляться на страницу, с которой был направлен на авторизацию.



В таблице выше демонстрируется логика работы менеджера аутентификации.

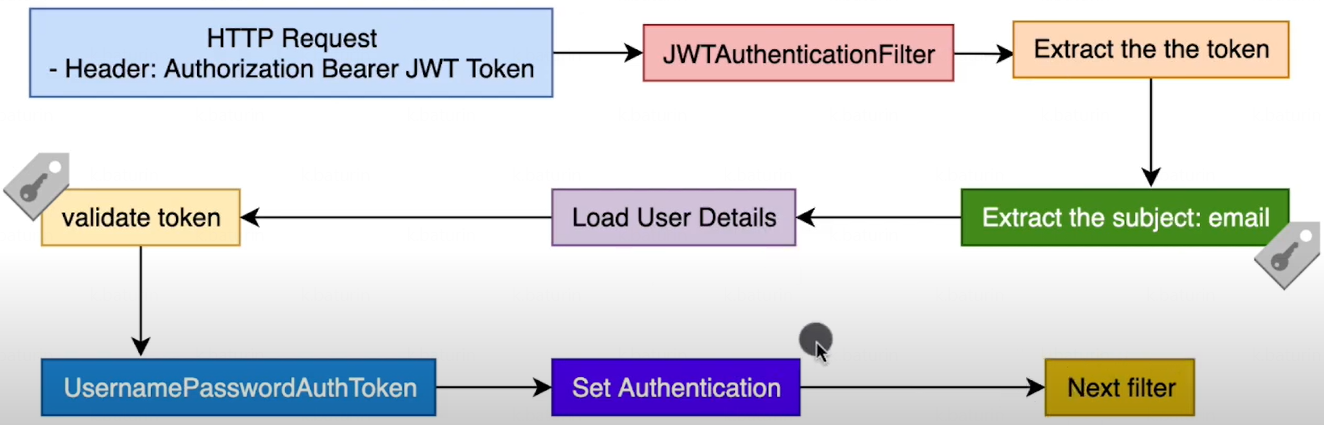
***AuthenticationManger*** – это интерфейс *API*, который определяет как ***Spring Security Filters*** проводят аутентификацию. Его реализацией по умолчанию является ***ProviderManager***.

В свою очередь, *ProviderManager* для работы использует реализации интерфейса ***AuthenticationProvier***. Например, ***DaoAuthenticationProvider***. Она отвечает за получение имени пользователя, пароля и других атрибутов для аутентификации.

***JWTAuthenticationFilter*** – это специальный фильтр, который мы внедряем в ***Secure Filter Chain***. Он будет перехватывать запрос и выполнять аутентификацию.

Логика работы аутентификации, следующая:

1. Запрос поступает в наше API.
2. Сначала запрос проходит через цепочку фильтров безопасности. Если запрос не пройдет хотя бы один из фильтров, то он отклоняется с ошибкой (например, 401 или 403).
3. Когда запрос попадает на наш фильтр *JWTAuthenticationFilter*, процесс аутентификации передается *AuthenticationManger*, а тот задействует реализацию *DaoAuthenticationProvider*. Она загружает данные из БД и сравнивает данные. В случае успеха фильтр добавляет пользователя в контекст безопасности.



Для построения объекта аутентификации из класса, он должен поддерживать интерфейс ***UserDetails***. Можно использовать уже готовую реализацию ***User*** для этого интерфейса или создать свою. Например:

*@Entity*

*@Data*

*@NoArgsConstructor*

*@AllArgsConstructor*

*public class Customer implements UserDetails {*

*@Id*

*@GeneratedValue(strategy=GenerationType.IDENTITY)*

*private Long id;*

*private final String username;*

*private final String password;*

*@Override // наделим правами с ролью USER*

*public Collection<? Extends GrantedAuthority> getAuthorities() {*

*return Arrays.asList(new SimpleGrantedAuthority(“ROLE\_USER”)); }*

*@Override // реализация-заглушка*

*public Boolean isAccountNonExpired() {*

*return true; }*

*@Override // реализация-заглушка*

*public Boolean isAccountNonLocked() {*

*return true; }*

*@Override // реализация-заглушка*

*public Boolean isCredentialsNonExpired() {*

*return true; }*

*@Override // реализация-заглушка*

*public Boolean isEnabled() {*

*return true; } }*

Для работы с объектами аутентификации *UserDetails*, нам потребуется класс реализующий интерфейс ***UserDetailsService***. Он может работать с хранилищами в памяти, БД, а также использовать сторонние сервисы *LDAP* или *OAuth2*. Например:

*@Service*

*@RequiredArgsConstructor*

*public class CustomerUserDetailsService implements UserDetailsService {*

*private final CustomerDao customerDao;*

*@Override*

*public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {*

*return customerDao.selectCustomerByEmail(username)*

*.orElseThrow(() -> new UsernameNotFoundException("Username " + username +" not found")); } }*

Поле *password* объекта *UserDetails* хранится в БД в виде хеша. В процессе аутентификации полученный пароль кодируется в хеш и сравнивается с хешем пароля из БД. Для включения шифрования передаваемого пароля нужно создать бин, возвращающий одну из реализаций ***PasswordEncoder***:

* *BCryptPasswordEncoder* – использовать шифрование *bcrypt*. По умолчанию сложность 10.
* *NoOpPasswordEncoder* – не применять шифрование.
* *Pbkdf2PasswordEncoder* – использовать шифрование *PBKDF2*.
* *SCryptPasswordEncoder* – использовать шифрование *Scrypt*.
* *StandardPasswordEncoder* – использовать шифрование *SHA-256*.

Создадим класс *SecurityConfig*, настроим шифрование, менеджер и провайдер аутентификации.

*@Configuration*

*public class SecurityConfig {*

*@Bean // создаем шифрование*

*public PasswordEncoder passwordEncoder() {*

*return new BCryptPasswordEncoder(); }*

*@Bean // создаем менеджер аутентификации*

*public AuthenticationManager authenticationManager(AuthenticationConfiguration configuration)*

*throws Exception {*

*return configuration.getAuthenticationManager(); }*

*@Bean // создаем провайдер аутентификации*

*public AuthenticationProvider authenticationProvider(*

*UserDetailsService userDetailsService, PasswordEncoder passwordEncoder) {*

*DaoAuthenticationProvider daoAuthenticationProvider = new DaoAuthenticationProvider();*

*daoAuthenticationProvider.setPasswordEncoder(passwordEncoder);*

*daoAuthenticationProvider.setUserDetailsService(userDetailsService);*

*return daoAuthenticationProvider; } }*

Создадим класс *JWTAuthenticationFilter* и настроим специальный фильтр цепочки безопасности.

*@Component*

*public class JWTAuthenticationFilter extends OncePerRequestFilter { // единожды для запроса*

*private final JWTUtil jwtUtil;*

*private final UserDetailsService userDetailsService;*

*public JWTAuthenticationFilter(JWTUtil jwtUtil, UserDetailsService userDetailsService) {*

*this.jwtUtil = jwtUtil;*

*this. userDetailsService = userDetailsService; }*

*@Override*

*protected void doFilterInternal(@NonNull HttpServletRequest request,*

*@NonNull HttpServletResponse response,*

*@NonNull FilterChain filterChain) throws ServletException, IOException {*

*// получаем токен из заголовка Authorization*

*String authHeader = request.getHeader("Authorization");*

*// если токена нет, то переходим к следующему фильтру*

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

*filterChain.doFilter(request, response);*

*return; }*

*// извлекаем информацию об инициаторе запроса из токена*

*String jwt = authHeader.substring(7);*

*String subject = jwtUtil.getSubject(jwt);*

*// если инициатор не пустой и не аутентифицирован*

*if (subject != null && SecurityContextHolder.getContext().getAuthentication() == null) {*

*// загружаем информацию о пользователе из БД*

*UserDetails userDetails = userDetailsService.loadUserByUsername(subject);*

*// сравниваем зашифрованный логин из jwt токена и из БД*

*if (jwtUtil.isTokenValid(jwt, userDetails.getUsername())) {*

*// создаем токен аутентификации и обновляем контекст безопасности для запроса*

*UsernamePasswordAuthenticationToken authenticationToken = new*

*UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());*

*authenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));*

*SecurityContextHolder.getContext().setAuthentication(authenticationToken); } }*

*// переходим к следующему фильтру*

*filterChain.doFilter(request, response); } }*

Для корректного проброса исключений через фильтр безопасности нужно создать реализацию интерфейса ***AuthenticationEntryPoint*** и добавить её в цепочку фильтров безопасности.

*@Component*

*public class DelegatedAuthEntryPoint implements AuthenticationEntryPoint {*

*private final HandlerExceptionResolver handlerExceptionResolver;*

*public DelegatedAuthEntryPoint(*

*@Qualifier("handlerExceptionResolver") HandlerExceptionResolver handlerExceptionResolver) {*

*this.handlerExceptionResolver = handlerExceptionResolver; }*

*@Override*

*public void commence(HttpServletRequest request, HttpServletResponse response,*

*AuthenticationException authException) throws IOException, ServletException {*

*// разрешаем проброс исключений*

*handlerExceptionResolver.resolveException(request, response, null, authException); } }*

Для централизованной обработки исключений и выдачи подробной информации об ошибке создадим класс с аннотациями ***@ControllerAdvice*** и ***@ExceptionHandler***.

*@ControllerAdvice*

*public class DefaultExceptionHandler {*

*// обрабатываем стандартную ошибку*

*@ExceptionHandler(ResourceNotFoundException.class)*

*public ResponseEntity<ApiError> handleException(ResourceNotFoundException e, HttpServletRequest request) {*

*ApiError apiError = new ApiError(request.getRequestURI(), e.getMessage(),*

*HttpStatus.NOT\_FOUND.value(), LocalDateTime.now());*

*return new ResponseEntity<>(apiError, HttpStatus.NOT\_FOUND); }*

*// обрабатываем ошибку авторизации*

*@ExceptionHandler(InsufficientAuthenticationException.class)*

*public ResponseEntity<ApiError> handleException(InsufficientAuthenticationException e,*

*HttpServletRequest request) {*

*ApiError apiError = new ApiError(request.getRequestURI(), e.getMessage(),*

*HttpStatus.FORBIDDEN.value(), LocalDateTime.now());*

*return new ResponseEntity<>(apiError, HttpStatus.FORBIDDEN); }*

*// обрабатываем все остальные ошибки*

*@ExceptionHandler(Exception.class)*

*public ResponseEntity<ApiError> handleException(Exception e, HttpServletRequest request) {*

*ApiError apiError = new ApiError(request.getRequestURI(), e.getMessage(),*

*HttpStatus.INTERNAL\_SERVER\_ERROR.value(), LocalDateTime.now());*

*return new ResponseEntity<>(apiError, HttpStatus.INTERNAL\_SERVER\_ERROR); } }*

Ранее для работы *CORS* мы создавали класс *WebMvcConfig* реализующий *WebMvcConfigurer*. После включения своей цепочки фильтров безопасности нужно создать свой фильтр ***CORS***, а старую настройку удалить.

*@Configuration*

*public class CorsConfig {*

*@Value("#{'${cors.allowed-origins}'.split(',')}")*

*private List<String> allowedOrigins;*

*@Value("#{'${cors.allowed-methods}'.split(',')}")*

*private List<String> allowedMethods;*

*@Value("#{'${cors.allowed-headers}'.split(',')}")*

*private List<String> allowedHeaders;*

*@Value("#{'${cors.exposed-headers}'.split(',')}")*

*private List<String> exposedHeaders;*

*@Bean*

*public CorsConfigurationSource corsConfigurationSource() {*

*CorsConfiguration configuration = new CorsConfiguration();*

*configuration.setAllowedOrigins(allowedOrigins);*

*configuration.setAllowedMethods(allowedMethods);*

*configuration.setAllowedHeaders(allowedHeaders); // разрешенные заголовки*

*configuration.setExposedHeaders(exposedHeaders); // ожидаемые заголовки*

*UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();*

*source.registerCorsConfiguration("/api/\*\*", configuration);*

*return source; } }*

Теперь, когда у нас уже есть служба входа по токену, создадим службу выдачи нового токена при аутентификации по логин/паролю ***AuthenticationService***.

*@Service*

*public class AuthenticationService {*

*private final AuthenticationManager authenticationManager;*

*private final CustomerDTOMapper customerDTOMapper;*

*private final JWTUtil jwtUtil;*

*public AuthenticationService(AuthenticationManager authenticationManager,*

*CustomerDTOMapper customerDTOMapper, JWTUtil jwtUtil) {*

*this.authenticationManager = authenticationManager;*

*this.customerDTOMapper = customerDTOMapper;*

*this.jwtUtil = jwtUtil; }*

*public AuthenticationResponse login(AuthenticationRequest request) {*

*// делегируем аутентификацию нашей реализации authenticationManager*

*// через провайдер DaoAuthenticationProvider*

*Authentication authenticate = authenticationManager.authenticate(*

*new UsernamePasswordAuthenticationToken(request.username(), request.password()) );*

*// получаем аутентифицированный объект из ответа*

*Customer principal = (Customer) authenticate.getPrincipal();*

*// преобразуем его в DTO сущность*

*CustomerDTO customerDTO = customerDTOMapper.apply(principal);*

*// создаем новый токен*

*String token = jwtUtil.issueToken(customerDTO.username(), customerDTO.roles());*

*return new AuthenticationResponse(token, customerDTO); } }*

Создадим эндпоинт контроллера для логина ***AuthenticationController***.

*@RestController*

*@RequestMapping("api/v1/auth")*

*public class AuthenticationController {*

*private final AuthenticationService authenticationService;*

*public AuthenticationController(AuthenticationService authenticationService) {*

*this.authenticationService = authenticationService; }*

*@PostMapping("login")*

*public ResponseEntity<?> login(@RequestBody AuthenticationRequest request) {*

*AuthenticationResponse response = authenticationService.login(request);*

*return ResponseEntity.ok()*

*.header(HttpHeaders.AUTHORIZATION, response.token())*

*.body(response); } }*

Для аутентификации через сторонний сайт, можно использовать спецификацию ***OAuth2***. Это потребует подключения стартера ***spring-boot-starter-oauth2-client***.

Для включения возможности защиты методов нужно включить глобальную защиту методов в классе конфигурации безопасности с помощью аннотации ***@EnableGlobalMethodSecurity***.

Аннотация ***@PreAuthorize()*** проверяет условие ***ДО*** вызова метода. В случае невыполнения условия, будет выброшено исключение *AccessDeniedException*, необязательное к обработке.

*@PreAuthorize(“hasRole(‘ADMIN’)”)*

*public void deleteAllOrders() {*

*orderRepository.deleteAll(); }*

Аннотация ***@PostAuthorize()*** проверяет условие ***ПОСЛЕ*** вызова метода. В случае невыполнения условия, будет выброшено исключение *AccessDeniedException*, необязательное к обработке.

*@PostAuthorize(“hasRole(‘ADMIN’) || returnObject.user.username == authentication.name”)*

*public TacoOrder getOrder(Long id) {*

*… }*

Для определения авторизованного пользователя можно использовать один из способов.

* Внедрить объект ***java.security.Principal*** в метод контроллера.

*@PostMapping*

*public String processOrder(@Valid TacoOrder order, Errors errors,*

*SessionStatus sessionStatus, Principal principal) {*

*User user = userRepository.findByUsername(principal.getName());*

*order.setUser(user);*

*… }*

* Внедрить объект ***org.springframework.security.core.Authentication*** в метод контроллера.

*@PostMapping*

*public String processOrder(@Valid TacoOrder order, Errors errors,*

*SessionStatus sessionStatus, Authentication authentication) {*

*User user = (User) authentication.getPrincipal();*

*order.setUser(user);*

*… }*

* Внедрить параметр метода с аннотацией ***@AuthenticationPrincipal***.

*@PostMapping*

*public String processOrder(@Valid TacoOrder order, Errors errors,*

*SessionStatus sessionStatus, @AuthenticationPrincipal User user) {*

*order.setUser(user);*

*… }*

* Использовать ***org.springframework.security.core.context.SecurityContextHolder***, чтобы получить контекст безопасности.

*@PostMapping*

*public String processOrder(@Valid TacoOrder order, Errors errors, SessionStatus sessionStatus) {*

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

*User user = (User) authentication.getPrincipal();*

*order.setUser(user);*

*… }*

стр.175