***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>*

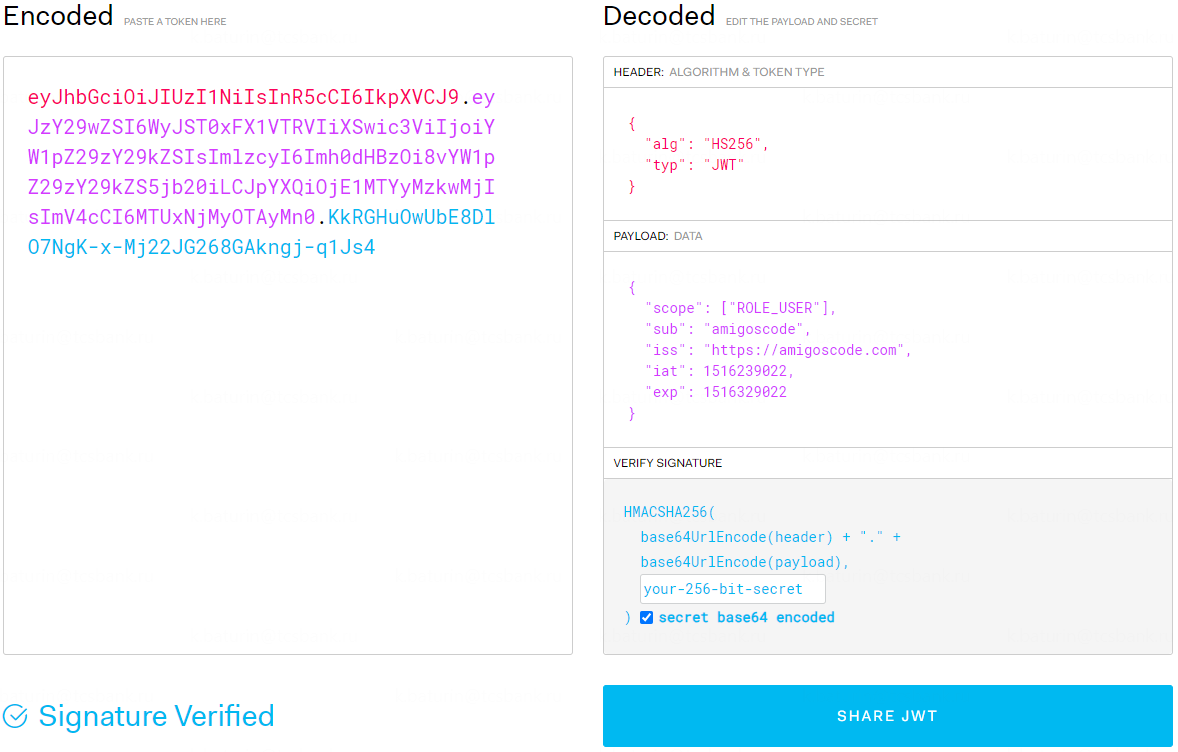
*<groupId>io.jsonwebtoken</groupId>*

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

*<version>0.12.3</version>*

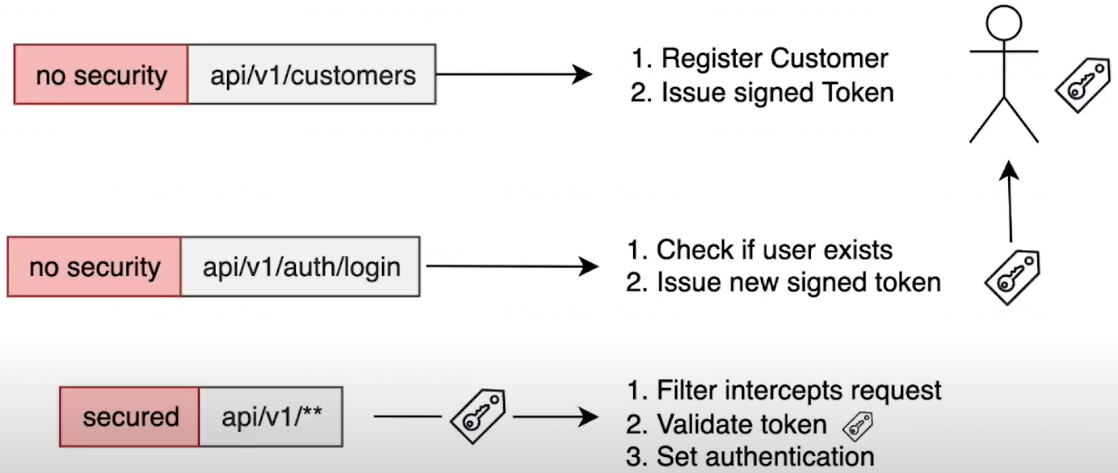
*<scope>runtime</scope>*

*</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 Key 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(); }*

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 CORS для HTML-форм*

*.csrf().disable()*

*// включаем ограничения по ролям и привилегиям*

*.authorizeHttpRequests(auth -> auth*

*// разрешить неавторизованные POST запросы на указанный эндпоинт*

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

*.permitAll()*

*// разрешить все запросы для авторизованных пользователей*

*.anyRequest()*

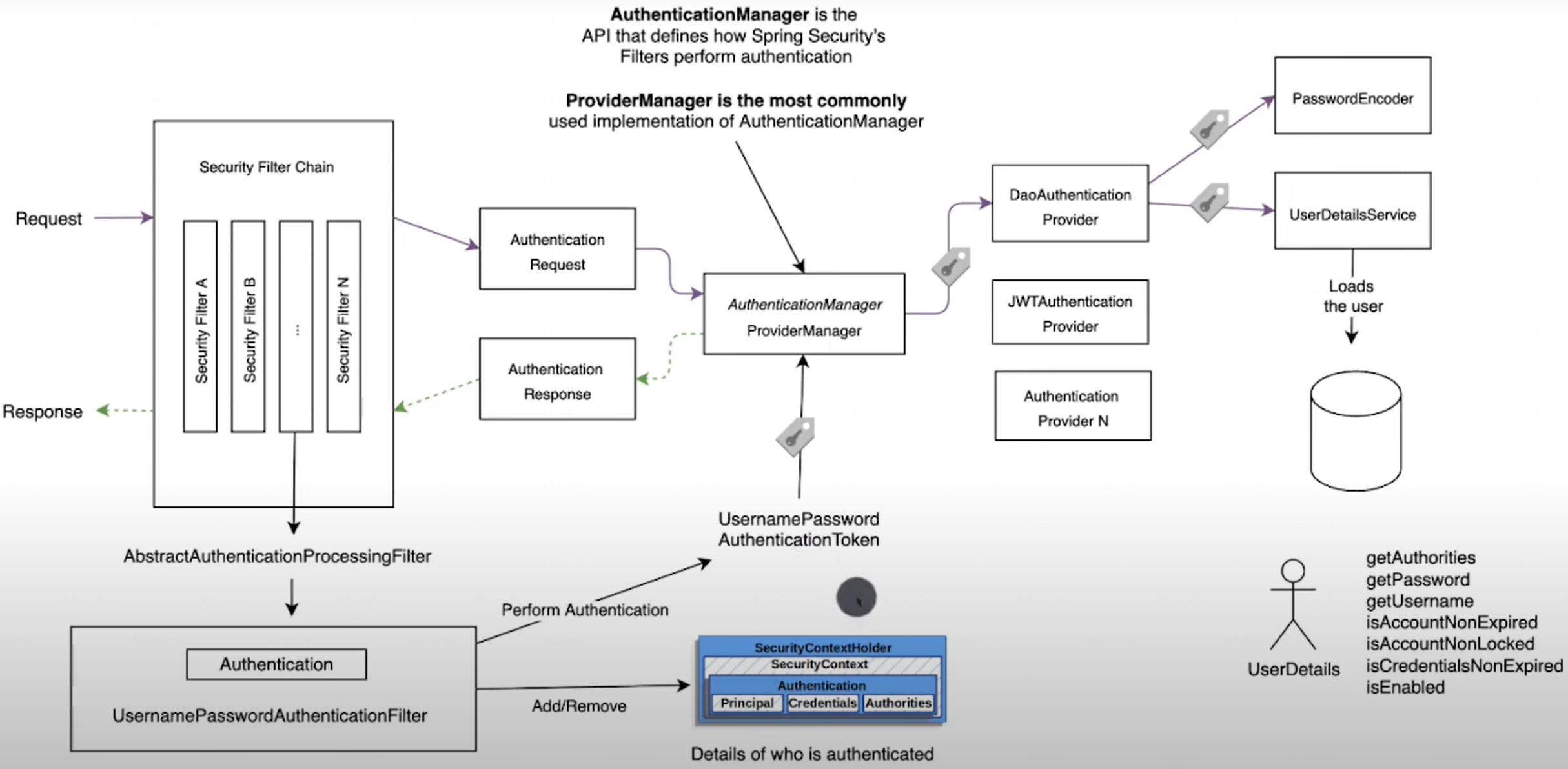
*.authenticated());*

*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.

***DaoAuthenticationProvider*** отвечает за получение имени пользователя, пароля и других атрибутов для аутентификации.

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")); } }*

Для включения шифрования передаваемого пароля нужно создать бин, возвращающий одну из реализаций ***PasswordEncoder***:

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

В результате, пароли будут храниться в БД в зашифрованном виде.

*@Configuration*

*public class SecurityConfig {*

*@Bean // включение шифрования паролей*

*public PasswordEncoder passwordEncoder() {*

*return new BCryptPasswordEncoder(); } }*

Создадим страницу и форму регистрации для полноценной работы с авторизацией через БД.

*@Controller*

*@RequestMapping("/register")*

*@RequiredArgsConstructor*

*public class RegistrationController { // контроллер регистрации*

*private final UserRepository userRepository;*

*private final PasswordEncoder passwordEncoder;*

*@GetMapping*

*public String registerForm() {*

*return "registration"; }*

*@PostMapping*

*public String processRegistration(RegistrationForm form) {*

*userRepository.save(form.toCustomUser(passwordEncoder));*

*return "redirect:/login"; } }*

*@Data*

*public class RegistrationForm { // класс с формой для регистрации*

*private String username;*

*private String password;*

*private String phone;*

*public CustomUser toCustomUser(PasswordEncoder passwordEncoder) {*

*return new User(username, passwordEncoder.encode(password), phone); } }*

*<!DOCTYPE html> // html-страница с формой для регистрации*

*<html xmlns=*[*http://www.w3.org/1999/xhtml*](http://www.w3.org/1999/xhtml)

*xmlns:th="http://www.thymeleaf.org">*

*<head>*

*<title>Taco Cloud</title>*

*</head>*

*<body>*

*<h1>Register</h1>*

*<img th:src="@{/images/TacoCloud.png}"/>*

*<form method="POST" th:action="@{/register}" id="registerForm">*

*<label for="username">Username: </label>*

*<input type="text" name="username"/><br/>*

*<label for="password">Password: </label>*

*<input type="password" name="password"/><br/>*

*<label for="confirm">Confirm password: </label>*

*<input type="password" name="confirm"/><br/>*

*<label for="phone">Phone: </label>*

*<input type="text" name="phone"/><br/>*

*<input type="submit" value="Register"/>*

*</form>*

*</body>*

*</html>*

Cоздадим страницу логина.

*<!DOCTYPE html>*

*<html xmlns=*[*http://www.w3.org/1999/xhtml*](http://www.w3.org/1999/xhtml)

*xmlns:th="http://www.thymeleaf.org">*

*<head>*

*<meta charset="UTF-8">*

*<title>Taco Cloud</title>*

*</head>*

*<body>*

*<h1>Login</h1>*

*<img th:src="@{/images/TacoCloud.png}"/>*

*<div th:if="${error}">*

*Unable to login. Check your username and password.*

*</div>*

*<p>New here? Click*

*<a th:href="@{/register}">here</a> to register.</p>*

*<form method="POST" th:action="@{/login}" id="loginForm">*

*<label for="username">Username: </label>*

*<input type="text" name="username" id="username" /><br/>*

*<label for="password">Password: </label>*

*<input type="password" name="password" id="password" /><br/>*

*<input type="submit" value="Login"/>*

*</form>*

*</body>*

*</html>*

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