Аутентификация

- Установка зависимостей
- Настройка процесса аутентификации
- Проверка наличия аутентификации
- Извлечение текущего пользователя

Аутентификация — это проверка подлинности. Например, программа может проверить, действительно ли пользователь является тем, за кого себя выдает. Мы все участвуем в этом процессе, когда заполняем формы логинов и паролей. Spring Boot отвечает за реализацию этого процесса со стороны бэкенда.

В отличие от всех остальных эндпоинтов API, аутентификация в Spring Boot работает не сама по себе. Чтобы работать с ней, нам придется использовать достаточно навороченный пакет Spring Security. Этот пакет предоставляет множество готовых компонентов, но требует тонкой настройки. Это большая и сложная тема, по которой пишутся целые книги. Глубокое изучение Spring Security — это слишком сложно, и в этом курсе мы не будем погружаться в эту тему. Поэтому большую часть кода в этом уроке мы рассмотрим без подробного объяснения.

Все события во время аутентификации можно разделить на два уровня:

- **Пользователь и его функциональность**. Сюда входит все от регистрации до проверки доступов. Чтобы работать с этим уровнем, нужно взять представленный в Spring Boot набор интерфейсов и реализовать его — тогда часть нужной функциональности начнет работать автоматически
- **Конкретный способ аутентификации**. В зависимости от способа аутентификации мы можем использовать разные механизмы и сторонние пакеты — например, OAuth или JWT-токены

В этом уроке мы последовательно пройдем по всем частям системы и настроим их. Для аутентификации мы будем использовать JWT-токены. Во время логина система будет формировать JWT-токен, который вернется клиенту. Клиент будет пользоваться этим токеном для последующих запросов, иначе ему будет отказано в доступе.

Установка зависимостей

Помимо Spring Security, нам понадобятся пакет для тестирования и пакет *oauth2-resource-server*, который выполняет большую часть логики по проверке доступа внутри себя:

```
implementation("org.springframework.boot:spring-boot-starter-security")
implementation("org.springframework.boot:spring-boot-starter-oauth2-resource-server")
testImplementation("org.springframework.security:spring-security-test")
```

Настройка процесса аутентификации

Для аутентификации пользователя понадобится два поля:

- *email* — это самый частый логин, кроме него иногда используют номер телефона или никнейм
- *passwordDigest* — это специальный хэш, связанный с паролем. Мы храним именно хэш, а не сам пароль, потому что с точки зрения безопасности, пароли хранить в базе данных нельзя. Чтобы не запутаться, мы назвали это поле *passwordDigest*, а не *password*

Внутри себя Spring Security работает с интерфейсом `UserDetails`, в который входят методы для работы с никнеймом, паролем и выдачей доступов. Сейчас мы реализуем только аутентификацию, поэтому нас интересует только часть методов. Остальные методы мы тоже реализуем, но после базовой функциональности:

```
package io.hexlet.spring.model;

import java.util.ArrayList;
import java.util.Collection;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Table;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import lombok.Getter;
import lombok.Setter;

@Entity
```

```java
@Getter
@Setter
@Table(name = "users")
public class User implements UserDetails, BaseEntity {
    // Остальные поля

    @Column(unique = true)
    @Email
    private String email;

    @NotBlank
    private String passwordDigest;

    @Override
    public String getPassword() {
        return passwordDigest;
    }

    @Override
    public String getUsername() {
        return email;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return new ArrayList<GrantedAuthority>();
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }
```

```
    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }
}
```

Обычно Spring Security работает с `username`, но под ним может скрываться что-то другое. Например, в нашем случае геттер возвращает `email`. Кроме того, вместо пароля мы получаем `passwordDigest`. Здесь все тоже корректно, потому что сравнение будет происходить с хэшем, а не с введенным пользователем паролем.

Далее мы создадим сервис, реализующий интерфейс `UserDetailsManager`. Через него Spring Boot будет выполнять CRUD-операции над пользователем. Для аутентификации реализуем метод `loadUserByUsername()`:

```
// src/main/java/io/hexlet/spring/service/CustomUserDetailsService.java
package io.hexlet.spring.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.provisioning.UserDetailsManager;
import org.springframework.stereotype.Service;

import io.hexlet.spring.model.User;
import io.hexlet.spring.repository.UserRepository;

@Service
public class CustomUserDetailsService implements UserDetailsManager {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        // Нужно добавить в репозиторий findByEmail
        var user = userRepository.findByEmail(email)
                .orElseThrow(() -> new UsernameNotFoundException("User not found"));
        return user;
    }
```

```
    @Override
    public void createUser(UserDetails userData) {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method 'createUser'");
    }

    @Override
    public void updateUser(UserDetails user) {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method 'updateUser'");
    }

    @Override
    public void deleteUser(String username) {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method 'deleteUser'");
    }

    @Override
    public void changePassword(String oldPassword, String newPassword) {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method 'changePassword'");
    }

    @Override
    public boolean userExists(String username) {
        // TODO Auto-generated method stub
        throw new UnsupportedOperationException("Unimplemented method 'userExists'");
    }
}
```

Дальше нам понадобится механизм хеширования пароля. Чтобы работать с ним, создадим бин:

```
// src/main/java/io/hexlet/spring/config/EncodersConfig.java
package io.hexlet.spring.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
```

```
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
public class EncodersConfig {
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

Теперь собираем все вместе. Указываем, что Spring Security должен использовать наши компоненты:

```
package io.hexlet.spring.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.annotation.authentication.builders.Authenticati
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.crypto.password.PasswordEncoder;

import io.hexlet.spring.service.CustomUserDetailsService;

@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Autowired
    private PasswordEncoder passwordEncoder;

    @Autowired
    private CustomUserDetailsService userService;

    @Bean
    public AuthenticationManager authenticationManager(HttpSecurity http) throws Exception
        return http.getSharedObject(AuthenticationManagerBuilder.class)
                .build();
    }
```

```
    @Bean
    public AuthenticationProvider daoAuthProvider(AuthenticationManagerBuilder auth) {
        var provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userService);
        provider.setPasswordEncoder(passwordEncoder);
        return provider;
    }
}
```

Мы сделали почти всю подготовительную работу. Осталось создать утилиту для генерации JWT-токена, и мы будем готовы реализовать аутентификацию.

Генерация состоит из двух вещей — подготовки токена и его шифрования. Рассмотрим пример класса, который делает эти операции:

```
// src/main/java/io/hexlet/spring/util/JWTUtils.java
package io.hexlet.spring.util;

import java.time.Instant;
import java.time.temporal.ChronoUnit;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.oauth2.jwt.JwtClaimsSet;
import org.springframework.security.oauth2.jwt.JwtEncoder;
import org.springframework.security.oauth2.jwt.JwtEncoderParameters;
import org.springframework.stereotype.Component;

@Component
public class JWTUtils {

    @Autowired
    private JwtEncoder encoder;

    public String generateToken(String username) {
        Instant now = Instant.now();
        JwtClaimsSet claims = JwtClaimsSet.builder()
                .issuer("self")
                .issuedAt(now)
                .expiresAt(now.plus(1, ChronoUnit.HOURS))
                .subject(username)
                .build();
```

```
        return this.encoder.encode(JwtEncoderParameters.from(claims)).getTokenValue();
    }
}
```

В коде выше используется `JwtEncoder`. Он добавляется в класс `EncodersConfig`, в который мы уже добавили `PasswordEncoder`:

```
package io.hexlet.spring.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.oauth2.jwt.JwtDecoder;
import org.springframework.security.oauth2.jwt.JwtEncoder;
import org.springframework.security.oauth2.jwt.NimbusJwtDecoder;
import org.springframework.security.oauth2.jwt.NimbusJwtEncoder;

import com.nimbusds.jose.jwk.JWK;
import com.nimbusds.jose.jwk.JWKSet;
import com.nimbusds.jose.jwk.RSAKey;
import com.nimbusds.jose.jwk.source.ImmutableJWKSet;
import com.nimbusds.jose.jwk.source.JWKSource;
import com.nimbusds.jose.proc.SecurityContext;

import io.hexlet.spring.component.RsaKeyProperties;

@Configuration
public class EncodersConfig {

    @Autowired
    // Создается ниже
    private RsaKeyProperties rsaKeys;

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    JwtEncoder jwtEncoder() {
```

```
        JWK jwk = new RSAKey.Builder(rsaKeys.getPublicKey()).privateKey(rsaKeys.getPrivate
        JWKSource<SecurityContext> jwks = new ImmutableJWKSet<>(new JWKSet(jwk));
        return new NimbusJwtEncoder(jwks);
    }

    @Bean
    JwtDecoder jwtDecoder() {
        return NimbusJwtDecoder.withPublicKey(rsaKeys.getPublicKey()).build();
    }
}
```

Для работы `JwtEncoder`нужны RSA-ключи — их можно сгенерировать, выполнив в терминале команды:

```
openssl genpkey -out private.pem -algorithm RSA -pkeyopt rsa_keygen_bits:2048
openssl rsa -in private.pem -pubout -out public.pem
```

Затем мы должны зайти в *application.yml* и указать путь к ключам. Например, вот так:

```
# src/main/resources/certs/
rsa:
  private-key: classpath:certs/private.pem
  public-key: classpath:certs/public.pem
```

Дальше мы создаем компонент, который прочитает эти ключи и сделает их доступными в коде:

```
package io.hexlet.spring.component;

import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

import lombok.Getter;
import lombok.Setter;

@Component
@ConfigurationProperties(prefix = "rsa")
```

```
@Setter
@Getter
public class RsaKeyProperties {
    private RSAPublicKey publicKey;
    private RSAPrivateKey privateKey;
}
```

Не забудьте, что в реальных проектах хранить приватный ключ в репозитории нельзя. Доступ к приватному ключу должен быть ограничен.

В нашем случае аутентификация — это обычный POST-запрос, в котором передаются электронная почта и пароль. Для унификации с предыдущими настройками, мы будем использовать имя `username` вместо почты `email`. В итоге у нас получится такой DTO:

```
package io.hexlet.spring.dto;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class AuthRequest {
    private String username;
    private String password;
}
```

Наконец, перейдем к контроллеру:

```
package io.hexlet.spring.controller.api;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import io.hexlet.spring.dto.AuthRequest;
import io.hexlet.spring.util.JWTUtils;

@RestController
```

```java
@RequestMapping("/api")
public class AuthenticationController {
    @Autowired
    private JWTUtils jwtUtils;

    @Autowired
    private AuthenticationManager authenticationManager;

    @PostMapping("/login")
    public String create(@RequestBody AuthRequest authRequest) {
        var authentication = new UsernamePasswordAuthenticationToken(
            authRequest.getUsername(), authRequest.getPassword());

        authenticationManager.authenticate(authentication);

        var token = jwtUtils.generateToken(authRequest.getUsername());
        return token;
    }
}
```

Все проделанные шаги свелись к строчке `authenticationManager.authenticate(authentication)`. В итоге внутри кода выполняется поиск пользователя в базе данных, хэширование пароля и сравнение. Если аутентификация прошла, выполнение продолжается дальше, если нет — код выбрасывает исключение.

После аутентификации программа генерирует токен и возвращает его клиенту. С этого момента клиент сам следит за отправкой токена в последующих запросах к API. При этом мы должны убрать API из публичного доступа и включить проверку токена.

Проверка наличия аутентификации

Проверка аутентификации настраивается в конфигурации, помеченной аннотацией `@EnableWebSecurity`. Выше мы уже создали такой класс, добавив туда конфигурацию провайдера и менеджера аутентификации. Теперь добавим туда фильтр, который применяется ко всем входящим запросам:

```java
package io.hexlet.spring.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
```

```java
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.authentication.builders.Authenticati
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.oauth2.jwt.JwtDecoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.web.servlet.handler.HandlerMappingIntrospector;

import io.hexlet.spring.service.CustomUserDetailsService;

@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Autowired
    private JwtDecoder jwtDecoder;

    @Autowired
    private PasswordEncoder passwordEncoder;

    @Autowired
    private CustomUserDetailsService userService;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http, HandlerMappingIntros
            throws Exception {
        // По умолчанию все запрещено
        return http
                .csrf(csrf -> csrf.disable())
                .authorizeHttpRequests(auth -> auth
                        // Разрешаем доступ только к /api/login, чтобы аутентифицироваться
                        .requestMatchers("/api/login").permitAll()
                        .anyRequest().authenticated())
                .sessionManagement(session -> session.sessionCreationPolicy(SessionCreatio
                .oauth2ResourceServer((rs) -> rs.jwt((jwt) -> jwt.decoder(jwtDecoder)))
                .httpBasic(Customizer.withDefaults())
                .build();
```

```
    }

    @Bean
    public AuthenticationManager authenticationManager(HttpSecurity http) throws Exception
        return http.getSharedObject(AuthenticationManagerBuilder.class)
                .build();
    }

    @Bean
    public AuthenticationProvider daoAuthProvider(AuthenticationManagerBuilder auth) {
        var provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userService);
        provider.setPasswordEncoder(passwordEncoder);
        return provider;
    }
}
```

Извлечение текущего пользователя

Кроме проверки доступа, в коде может понадобиться пользователь, выполняющий запрос. Например, такое бывает нужно, когда создается какая-то сущность, у которой есть автор. В таком случае автор почти всегда тот, кто выполняет запрос. Spring Security предоставляет возможность извлечь его из контекста:

```
package io.hexlet.spring.util;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Component;

import io.hexlet.spring.model.User;
import io.hexlet.spring.repository.UserRepository;

@Component
public class UserUtils {
    @Autowired
    private UserRepository userRepository;

    public User getCurrentUser() {
```

```
        var authentication = SecurityContextHolder.getContext().getAuthentication();
        if (authentication == null || !authentication.isAuthenticated()) {
            return null;
        }
        var email = authentication.getName();
        return userRepository.findByEmail(email).get();
    }
}
```

Этот класс позволяет получать пользователя в контроллере одним вызовом.

Дополнительные материалы

1. Пять простых шагов для понимания JWT
2. Securing web application, туториал

Далее →