Laboratorium 9. *Spring* i *JSON Web Token*

Cel zajęć

Wykonanie zadań z laboratorium pozwoli studentom poznać zaawansowane elementy konfiguracji *Spring Security*, konieczne do implementacji w aplikacjach uwierzytelniających klienta za pomocą tokena *JSON Web Token* (*JWT*) z wykorzystaniem ustalonych danych użytkownika.

Zakres tematyczny

- Przygotowanie *REST API* z autentykacją użytkownika o ustalonym loginie i haśle za pomocą tokena *JWT*.
- Konfiguracja *Spring Security* z autentykacją tokenem *JWT* (definicja klas konfiguracyjnych, generowanie i walidacja *JWT*).
- Testowanie *REST API* z autentykacją tokenem *JWT* w narzędziu *Postman*.

Proces konfiguracji *Spring Security*, generowania i walidacji tokena *JWT*, zrealizowano według przykładu prezentowanego na stronie [10]: https://www.techgeeknext.com/spring/spring-boot-security-token-authentication-jwt.
Definicje klas prezentowanych w przykładach niniejszego laboratorium, niezbędne do prawidłowej konfiguracji *Spring Security* do pracy z tokenem *JWT*, pochodzą w całości z pozycji [10]. W celu lepszego zrozumienia działania metod prezentowanych klas, komentarze w kodach, zostały przetłumaczone na język polski.

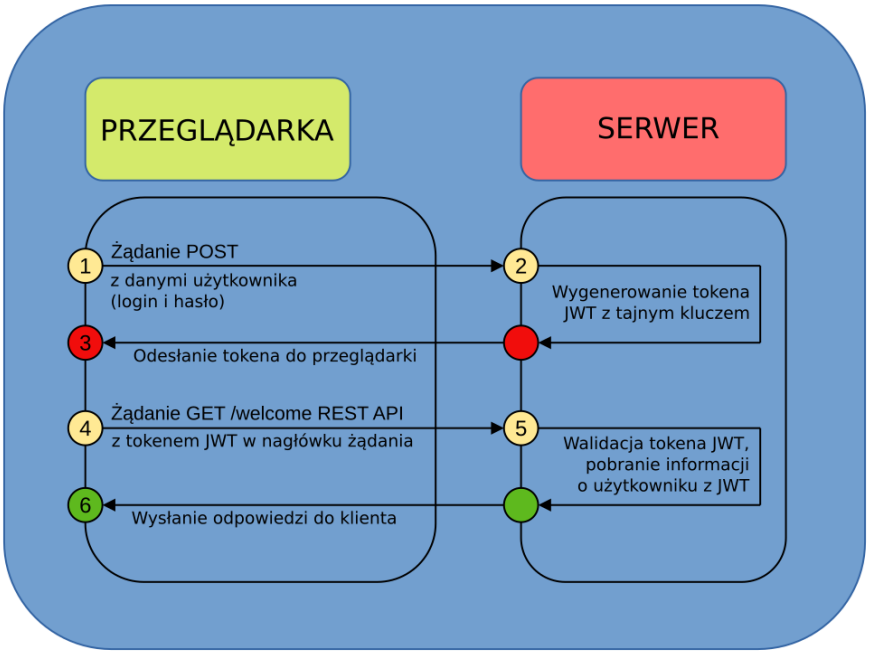
Wprowadzenie

Autentykacja użytkownika za pomocą *JWT* przebiega w następujących krokach:

- Użytkownik uwierzytelnia się wysyłając swoje dane (login i hasło).
- Po udanej autentykacji, serwer generuje token *JWT*, w którym są zaszyte dane użytkownika i informacja o jego uprawnieniach (ang. *credentials*) do korzystania z zasobów serwera oraz data ważności tokena.
- Serwer podpisuje (i jeśli trzeba koduje token *JWT*) oraz wysyła token do klienta w odpowiedzi na pierwsze żądanie.
- W oparciu o datę ważności ustanowioną po stronie serwera – klient odpowiednio długo przechowuje token *JWT* i przesyła go w nagłówku każdego kolejnego żądania.
- Klient identyfikuje użytkownika w oparciu o ten token, wobec czego nie trzeba za każdym kolejnym żądaniem przesyłać loginu i hasła w celu

uwierzytelnienia (robi się to tylko za pierwszym razem, a w odpowiedzi serwer wysyła token, który klient wykorzystuje do autentykacji kolejnych żądań).

Spring Boot REST Authentication z wykorzystaniem tokena *JWT* przedstawia rysunek 9.1.



Rys. 9.1. Uwierzytelnianie za pomocą tokena *JWT* [10]

Zadanie 9.1. *Spring Boot* i *JWT*

Wygeneruj nowy projekt o nazwie np. ***PAI_testjwt*** za pomocą narzędzia *Spring Initializr* z dołączoną zależnością *Web*. W pakiecie głównym projektu dodaj pakiet ***controller***, a w nim utwórz klasę ***TestController*** (Przykład 9.1).

Przykład 9.1. Klasa TestController

```
package bp.PAI_testjwt.controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class TestController {

    @RequestMapping({ "/hello" })
    public String welcomePage() {
```

```
        return "Welcome!";
    }
}
```

Uruchom i przetestuj aplikację z adresu ***http://localhost:8080/hello*** (metodą *GET*) a następnie do pliku ***pom.xml*** dołącz zależności dla *Spring Security* i *JWT*:

```
<dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
       <groupId>io.jsonwebtoken</groupId>
       <artifactId>jjwt</artifactId>
       <version>0.9.1</version>
</dependency>
```

Po ponownym zbudowaniu projektu z dodanymi zależnościami i uruchomieniu, aby zobaczyć efekt działania dla adresu ***/hello*** należy się zalogować jako użytkownik ***user*** za pomocą wygenerowanego przez serwer hasła. Jest to domyślna konfiguracja *Spring Security*, którą teraz należy zmodyfikować tak, aby autentykacja odbywała się za pomocą przekazanego tokena *JWT*.

9.1.1. Konfiguracja *Spring Security* z *JWT*

W celu konfiguracji *Spring Security* oraz wygenerowania i sprawdzenia poprawności tokena *JWT* należy obsłużyć dwie akcje:

- **Generowanie *JWT***: punkt końcowy (ang. *endpoint*) ***/authenticate*** zostanie wykorzystany w celu przekazania metodą *POST* nazwy i hasła użytkownika (***username***, ***password***) i wygenerowania tokena *JWT*.
- **Walidacja *JWT***: zostanie zrealizowana przez utworzoną już akcję kontrolera (*GET*) dla punktu końcowego ***/hello*** z przesłaniem otrzymanego w odpowiedzi z serwera tokena *JWT*.

Utwórz pakiet ***config*** a w nim zdefiniuj klasę ***JwtTokenUtil*** (Przykład 9.2) z metodami niezbędnymi do generowania i walidacji tokena *JWT*. Strukturę plików gotowego projektu pokazano na rysunku 9.2.

Przykład 9.2. Klasa JwtTokenUtil [10]

```
package bp.PAI_testjwt.config;

import java.io.Serializable;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;
```

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;

@Component
public class JwtTokenUtil implements Serializable {
    private static final long serialVersionUID = -2550185165626007488L;
    public static final long JWT_TOKEN_VALIDITY = 5 * 60 * 60;

    @Value("")
    private String secret;

    public String getUsernameFromToken(String token) {
        return getClaimFromToken(token, Claims::getSubject);
    }

    public Date getIssuedAtDateFromToken(String token) {
        return getClaimFromToken(token, Claims::getIssuedAt);
    }

    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) {
        return Jwts.parser()
                .setSigningKey(secret).parseClaimsJws(token)
                .getBody();
    }

    private Boolean isTokenExpired(String token) {
        final Date expiration = getExpirationDateFromToken(token);
        return expiration.before(new Date());
    }

    private Boolean ignoreTokenExpiration(String token) {
        // podaj tokeny, dla których wygaśnięcie jest ignorowane
        return false;
    }

    public String generateToken(UserDetails userDetails) {
        Map<String, Object> claims = new HashMap<>();
```

```
        return doGenerateToken(claims, userDetails.getUsername());
    }

    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))
          .signWith(SignatureAlgorithm.HS512, secret).compact();
    }

    public Boolean canTokenBeRefreshed(String token) {
        return (!isTokenExpired(token) || ignoreTokenExpiration(token));
    }

    public Boolean validateToken(String token, UserDetails userDetails){
        final String username = getUsernameFromToken(token);
        return (username.equals(userDetails.getUsername()) &&
                !isTokenExpired(token));
    }
}
```

9.1.2. Pobranie nazwy i hasła użytkownika

W pakiecie ***org.springframework.security.core.userdetails*** *Spring Security* dostarcza interfejs ***UserDetailsService***. Interfejs ten będzie wykorzystany, aby sprawdzić ***username***, ***password*** oraz uprawnienia danego użytkownika (ang. *GrantedAuthorities*).

Interfejs posiada tylko jedną metodę ***loadUserByUsername***. Menadżer autentykacji (***Authentication Manager***) wywołuje tę metodę, aby pobrać dane żytkownika z bazy danych w celu porównania ich z dostarczonymi danymi. W tym zadaniu, do przetestowania konfiguracji *JWT*, wykorzystany zostanie użytkownik z ustalonym, zahaszowym już hasłem (***BCrypt password***). Na kolejnym laboratorium 10 aplikacja zostanie rozbudowana tak, aby korzystała z danych użytkowników zapisanych w bazie *MySQL*.

Utwórz kolejny pakiet ***service*** z klasą ***JwtUserDetailsService*** (Przykład 9.3), która nadpisze metodę interfejsu ***UserDetailsService***.

Przykład 9.3. Klasa JwtUserDetailsService [10]

```
package bp.PAI_testjwt.service;

import java.util.ArrayList;
import org.springframework.security.core.userdetails.User;
```

```
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;
@Service
public class JwtUserDetailsService implements UserDetailsService {

  @Override
  public UserDetails loadUserByUsername(String username) throws
    UsernameNotFoundException {
      if ("pai".equals(username)) {
        return new User("pai",
         "$2a$10$slYQmyNdGzTn7ZLBXBChFOC9f6kFjAqPhccnP6DxlWXx2lPk1C3G6",
         new ArrayList<>());
      } else {
        throw new UsernameNotFoundException(
           "User not found with username: " + username);
      }
  }
}
```

W zadaniu wykorzystano ustalone dane użytkownika i przykładowe, zahaszowane już (bezpieczną funkcją haszującą *BCrypt* wykorzystaną przez serwer) hasło dla łańcucha "password":

```
username="pai"
password="$2a$10$slYQmyNdGzTn7ZLBXBChFOC9f6kFjAqPhccnP6DxlWXx2lP
k1C3G6"
```

9.1.3. Kontroler do autentykacji użytkownika

Do autentykacji użytkownika należy zdefiniować ***JwtAuthenticationController*** z akcją obsługującą żądanie (z danymi ***username*** i ***passsword***), przekazane metodą *POST* do *API*. Za pomocą menadżera autentykacji (ang. *Authentication Manager*) zostanie przeprowadzona autentykacja użytkownika i jego uprawnień (ang. *credentials*). Jeśli weryfikacja zakończy się sukcesem, ***JWTTokenUtil*** wygeneruje token *JWT*, który następnie zostanie przekazany do klienta.

W pakiecie ***controller*** zdefiniuj klasę kontrolera ***JwtAuthenticationController*** (Przykład 9.4).

Przykład 9.4. Klasa JwtAuthenticationController [10]

```
package bp.PAI_testjwt.controller;
import bp.PAI_testjwt.config.JwtTokenUtil;
import bp.PAI_testjwt.model.JwtRequest;
import bp.PAI_testjwt.model.JwtResponse;
import java.util.Objects;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import
org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.authentication.DisabledException;
import
org.springframework.security.authentication.UsernamePasswordAuthenticati
onToken;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
@CrossOrigin
public class JwtAuthenticationController {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private JwtTokenUtil jwtTokenUtil;

    @Autowired
    private UserDetailsService jwtInMemoryUserDetailsService;

    @RequestMapping(value = "/authenticate",
                    method = RequestMethod.POST)
    public ResponseEntity<?> generateAuthenticationToken(@RequestBody
      JwtRequest authenticationRequest) throws Exception {
        authenticate(authenticationRequest.getUsername(),
                     authenticationRequest.getPassword());
        final UserDetails userDetails = jwtInMemoryUserDetailsService
           .loadUserByUsername(authenticationRequest.getUsername());
        final String token = jwtTokenUtil.generateToken(userDetails);
        return ResponseEntity.ok(new JwtResponse(token));
    }

    private void authenticate(String username, String password)
          throws Exception {
        Objects.requireNonNull(username);
        Objects.requireNonNull(password);
        try {
            authenticationManager.authenticate(new
             UsernamePasswordAuthenticationToken(username, password));
        } catch (DisabledException e) {
```

```
            throw new Exception("USER_DISABLED", e);
        } catch (BadCredentialsException e) {
            throw new Exception("INVALID_CREDENTIALS", e);
        }
    }
}
```

Klasa tego kontrolera korzysta z obiektów ***JwtRequest*** i ***JwtResponse***, które zdefiniowane zostaną w kolejnych dwóch punktach. Obie klasy dodaj do kolejnego pakietu ***model*** (Rys. 9.2).

9.1.4. Klasa *JwtRequest*

Obiekt klasy ***JwtRequest*** (Przykład 9.5) jest wykorzystywany do pobrania danych użytkownika (***username*** i ***password***) z żądania wysłanego przez klienta.

Przykład 9.5. Klasa JwtRequest [10]

```
package bp.PAI_testjwt.model;
import java.io.Serializable;
public class JwtRequest implements Serializable {
    private static final long serialVersionUID = 5926468583005150707L;
    private String username;
    private String password;

    // domyślny konstruktor dla parsowania JSON
    public JwtRequest() {
    }

    public JwtRequest(String username, String password) {
        this.setUsername(username);
        this.setPassword(password);
    }

    public String getUsername() {
        return this.username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return this.password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

9.1.5. Klasa *JwtResponse*

Obiekt klasy ***JwtResponse*** (Przykład 9.6) jest wykorzystywany w celu utworzenia odpowiedzi z tokenem *JWT*, zwracanej do klienta.

Przykład 9.6. Klasa JwtResponse [10]

```
package bp.PAI_testjwt.model;
import java.io.Serializable;
public class JwtResponse implements Serializable {

    private static final long serialVersionUID = -8091879091924046844L;
    private final String jwttoken;

    public JwtResponse(String jwttoken) {
        this.jwttoken = jwttoken;
    }

    public String getToken() {
        return this.jwttoken;
    }
}
```

Pozostaje jeszcze do zdefiniowania najważniejsza klasa filtra ***JWT*** oraz konfiguracja ***Web Security***.

9.1.6. Filtr *JWT* w pakiecie *config*

Klasa ***JwtRequestFilter*** (Przykład 9.7) obsługuje wszystkie przychodzące żądania, waliduje tokeny *JWT* (przesyłane z żądaniem) i umieszcza w kontekście aplikacji informację, że zalogowany użytkownik jest uwierzytelniony.

Przykład 9.7. Klasa JwtRequestFilter [10]

```
package bp.PAI_testjwt.config;

import bp.PAI_testjwt.service.JwtUserDetailsService;
import java.io.IOException;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.security.authentication.UsernamePasswordAuthenticati
onToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
```

```
import
org.springframework.security.web.authentication.WebAuthenticationDetails
Source;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import io.jsonwebtoken.ExpiredJwtException;

@Component
public class JwtRequestFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUserDetailsService jwtUserDetailsService;

    @Autowired
    private JwtTokenUtil jwtTokenUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                    HttpServletResponse response, FilterChain chain)
            throws ServletException, IOException {
        final String requestTokenHeader =
                     request.getHeader("Authorization");
        String username = null;
        String jwtToken = null;
        // Token JWT ma postać "Bearer token". Należy usunąć słowo
        // Bearer i pobrać tylko właściwy token
        if (requestTokenHeader!=null &&
           requestTokenHeader.startsWith("Bearer ")) {
            jwtToken = requestTokenHeader.substring(7);
            try {
                username = jwtTokenUtil.getUsernameFromToken(jwtToken);
            } catch (IllegalArgumentException e) {
                System.out.println("Unable to get JWT Token");
            } catch (ExpiredJwtException e) {
                System.out.println("JWT Token has expired");
            }
        } else {
            logger.warn("JWT Token does not begin with Bearer String");
        }
        // Po uzyskaniu tokena należy go zweryfikować
        if (username != null &&
       SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails =
               this.jwtUserDetailsService.loadUserByUsername(username);
            // Jeśli token jest poprawny, będzie konfigurowane
            // uwierzytelnianie w Spring Security
            if (jwtTokenUtil.validateToken(jwtToken, userDetails)) {
                UsernamePasswordAuthenticationToken
                 usernamePasswordAuthenticationToken = new
                   UsernamePasswordAuthenticationToken(
```

```
                userDetails, null,
                userDetails.getAuthorities());
            usernamePasswordAuthenticationToken
                    .setDetails(new
            WebAuthenticationDetailsSource().buildDetails(request));
          // Po powyższych ustawieniach, bieżący użytkownik jest
          // traktowany jako uwierzytelniony, Konfiguracja
          // Spring Security - zakończona powodzeniem
          SecurityContextHolder.getContext()
            .setAuthentication(usernamePasswordAuthenticationToken);
         }
       }
       chain.doFilter(request, response);
   }
}
```

9.1.7. Klasa *JwtAuthenticationEntryPoint*

Do pakietu ***config*** dodaj klasę ***JwtAuthenticationEntryPoint*** (Przykład 9.8), która wysyła kod błędu 401 w odpowiedzi na brak autentykacji użytkownika.

Przykład 9.8. Klasa JwtAuthenticationEntryPoint [10]

```
package bp.PAI_testjwt.config;
import java.io.IOException;
import java.io.Serializable;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

@Component
public class JwtAuthenticationEntryPoint implements
AuthenticationEntryPoint, Serializable {

    private static final long serialVersionUID = -7858869558953243875L;

    @Override
    public void commence(HttpServletRequest request,
                         HttpServletResponse response,
                         AuthenticationException authException)
              throws IOException {
          response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
                             "Unauthorized");
    }
}
```

9.1.8. Konfiguracja *Web Security*

Kolejna klasa ***WebSecurityConfig*** w pakiecie ***config*** (Przykład 9.9) rozszerza ***WebSecurityConfigurerAdapter***, dzięki czemu można dowolnie skonfigurować ***WebSecurity i HTTPSecurity***.

Przykład 9.9. Klasa WebSecurityConfig [10]

```
package bp.PAI_testjwt.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.config.annotation.authentication.builders.A
uthenticationManagerBuilder;
import
org.springframework.security.config.annotation.method.configuration.Enab
leGlobalMethodSecurity;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity
;
import
org.springframework.security.config.annotation.web.configuration.EnableW
ebSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecu
rityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenti
cationFilter;

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;

    @Autowired
    private UserDetailsService jwtUserDetailsService;

    @Autowired
    private JwtRequestFilter jwtRequestFilter;
```

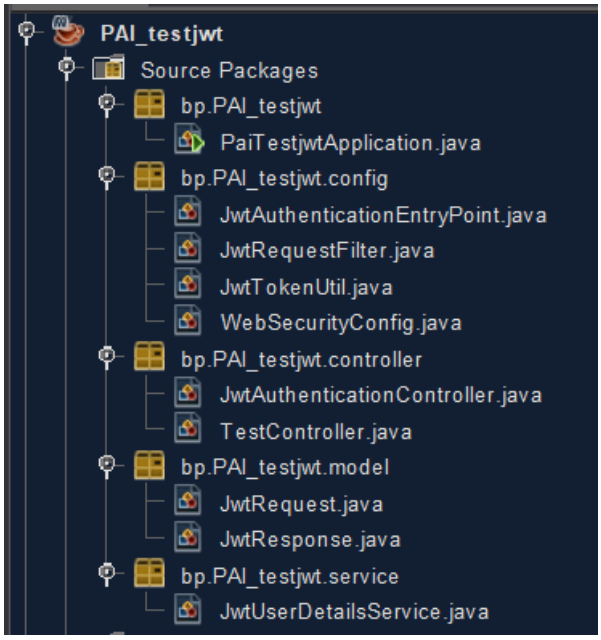
```
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth)
              throws Exception {
        // Konfiguracja menadżera AuthenticationManager tak, aby
        // wiedział skąd załadować użytkownika w celu dopasowania
        // danych uwierzytelniających
        // Zastosowano haszowanie hasła za pomocą BCryptPasswordEncoder
        auth.userDetailsService(jwtUserDetailsService).
                  passwordEncoder(passwordEncoder());
    }

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

    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean()
             throws Exception {
        return super.authenticationManagerBean();
    }

    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception
{
        // W tym przykładzie nie potrzebne jest zabezpieczenie CSRF
        httpSecurity.csrf().disable()
            // te żądania nie wymagają uwierzytelniania
            .authorizeRequests().antMatchers("/authenticate")
            .permitAll().
            // pozostałe żądania wymagają uwierzytelniania
            anyRequest().authenticated().and()
            // zastosowana sesja bezstanowa - sesja nie przechowuje
            // stanu użytkownika.
            .exceptionHandling()
            .authenticationEntryPoint(jwtAuthenticationEntryPoint).and()
            .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
            // Dodanie filtra do walidacji tokena przy każdym żądaniu
            httpSecurity.addFilterBefore(jwtRequestFilter,
                  UsernamePasswordAuthenticationFilter.class);
    }
}
```

Struktura całego projektu przedstawiona jest na rysunku 9.2.



Rys. 9.2. Struktura gotowego projektu

Zadanie 9.2. Test działania w narzędziu *Postman*

Do pliku ***application.properties*** dodaj wiersze:

```
spring.main.allow-circular-references=true
jwt.secret=Programowanie_ai2021
```

Korzystając z narzędzia *Postman*, wyślij żądanie typu POST na adres ***localhost:8080/authenticate*** z przekazaniem loginu i hasła (Rys. 9.3):

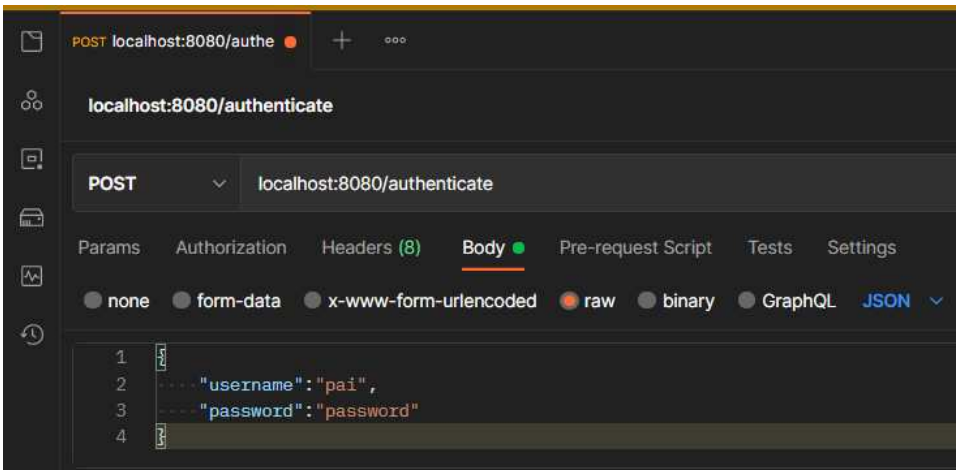
```
{   "username":"pai", "password":"password" }
```

Po sprawdzeniu danych użytkownika, jeżeli są prawidłowe, w odpowiedzi serwer odsyła token (Rys. 9.4).

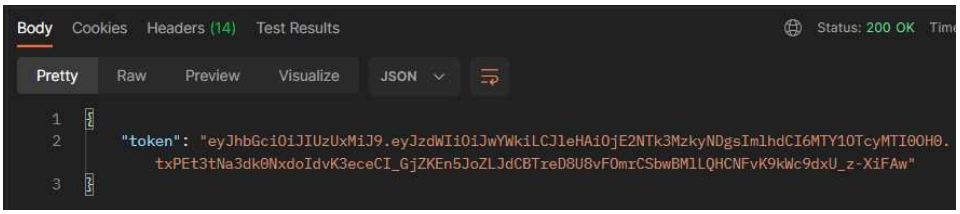
UWAGA! Jeśli korzystasz z wersji języka *Java >8*, może się pojawić wyjątek: *java.lang.NoClassDefFoundError: javax/xml/bind/JAXBException*

Rozwiązaniem tego problemu jest dodanie do pliku *pom.xml* zależności:

```
<dependency>
      <groupId>javax.xml.bind</groupId>
      <artifactId>jaxb-api</artifactId>
      <version>2.3.0</version>
</dependency>
```



Rys. 9.3. ***Postman*** **– żądanie z danymi użytkownika**

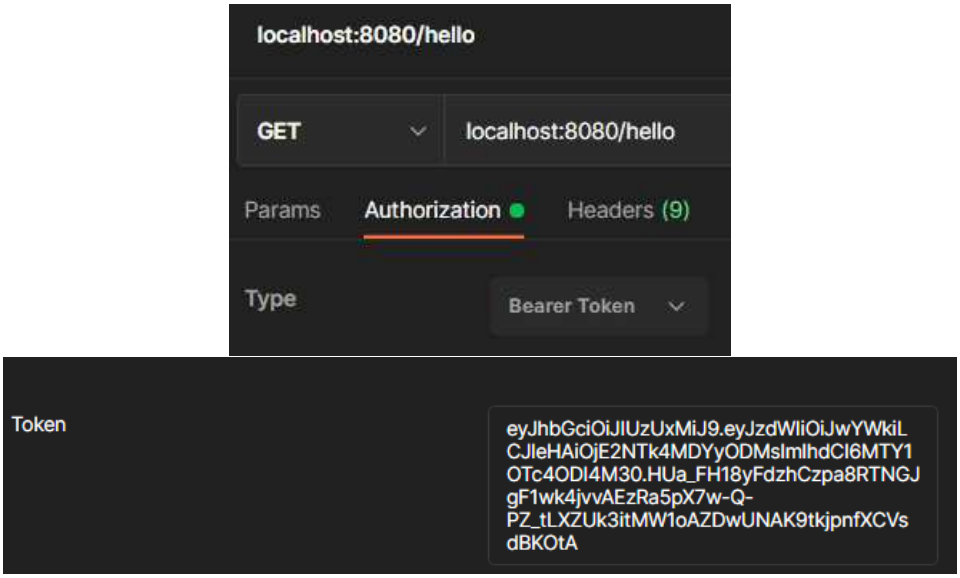


Rys. 9.4. ***Postman*** **– odpowiedź serwera z tokenem *JWT***

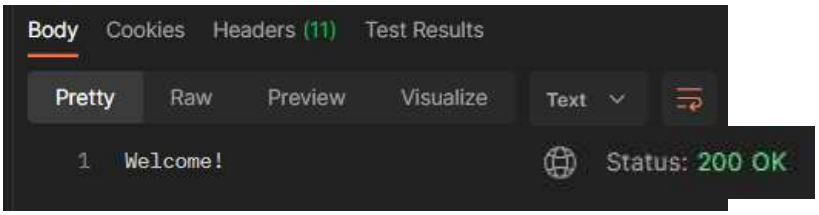
Skopiuj otrzymany token tak, aby można go było wykorzystać do uwierzytelnienia użytkownika w przypadku żądań wymagających autentykacji tokenem.

Przetestuj działanie autentykacji, wysyłając na adres ***localhost:8080/hello***, żądanie wymagające podania tokena do autentykacji. Token należy przekazać w nagłówku ***Authorization*** żądania (Rys. 9.5). Wybierz opcję ***Authorisation*** i wskaż typ tokena jako **Bearer token** oraz w odpowiednim polu tekstowym wklej token skopiowany (bez znaków cudzysłowia) z poprzedniego punktu i wyślij żądanie.

W odpowiedzi, po poprawnej weryfikacji tokena serwer prześle odpowiedź (***Status 200 OK***) postaci jak na rysunku 9.6.



Rys. 9.5. ***Postman*** **– żądanie z tokenem do autentykacji**



Rys. 9.6. ***Postman*** **– odpowiedź serwera po poprawnej weryfikacji tokena**

Próba dostępu do strony bez przesłania właściwego tokena zakończy się niepowodzeniem – serwer zwróci odpowiedź ***Status 401 Unauthorize*** (Rys. 9.7).

Rys. 9.7. ***Postman*** **– odpowiedź serwera po niepoprawnej weryfikacji tokena**

Taką samą odpowiedź jak na rysunku 9.7 serwer zwróci w przypadku próby przesłania w żądaniu *POST*, niepoprawnych danych użytkownika (sprawdź).

W niniejszym laboratorium test poprawności przechodzi tylko 1 użytkownik o podanych wcześniej parametrach, które zakodowano na sztywno w metodzie ***loadUserByUsername*** klasy ***JwtU serDetailsService***. W kolejnym laboratorium zostanie wykorzystana baza danych *MySQL* z danymi do uwierzytelnienia użytkownika.