Bezbednost veb informacionih sistema

Autori:
Goran Savić
Milan Segedinac

1. Opšti aspekti bezbednosti

Kada govorimo o bezbednosti veb orijentisanog informacionog sistema, dva aspekta su od ključne važnosti. Prvi aspekt je kontrola pristupa u zavisnosti od identiteta korisnika koji pristupa aplikaciji. Utvrđivanje identiteta korisnika koji pristupa se naziva **autentikacija** (eng. *authentication*) i standardno se vrši zahtevanjem da se korisnik prekorišćenja sistema prijavi na sistem slanjem kredencijala (korisničko ime i lozinka). Korisnik koji koristi aplikaciju može imati različita prava pristupa u aplikaciji. Neke funkcionalnosti su dostupne samo određenim korisnicima. Pri pristupu funkcionalnostima neophodno je proveriti da li korisnik ima pravo da izvrši određenu funkcionalnost. Ovo se naziva **autorizacija** (eng. *authorization*). Obzirom da je komplikovano definisati kontrolu pristupa na nivou pojedinačnog korisnika, standardna realizacija je da se prava pristupa dodeljuju ulogama korisnika (npr. administrator, standardni korisnik, gost, ...). Svakom korisniku se dodeljuju uloge kojima pripada.

Drugi važan aspekt bezbednosti je kako se podaci u aplikaciji prenose. Kada govorimo o veb aplikacijama, podaci se preko mreže razmenjuju putem poruka između klijenta i servera. Postoji opasnost da neko presretne mrežnu komunikaciju i pristupi podacima koje poruka sadrži. Iz tog razloga, za sigurnu komunikaciju je neophodno izvršiti šifrovanje poruke pre slanja, odnosno dešifrovanje nakon prijema. U nastavku teksta biće reči o konkretnim tehnikama obezbeđivanja sigurne komunikacije kroz šifrovanje poruka.

2. Šifrovanje poruka

Osnovna ideja šifrovanja (eng. *encryption*) poruka je transformacija poruke u drugu poruku, koja se može dešifrovanjem (eng. *decryption*) transformisati u izvornu poruku. Šifrovanje se vrši zamenom svakog karaktera u poruci drugim karakterom prema nekoj šemi šifrovanja. Parametar koji određuje pravilo transformacije izvornog podatka u šifrovani podatak naziva se **ključ** (eng. *key*).

Na primer, ako je izvorna poruka *Srbija*, svaki karakter možemo predstaviti brojnom vrednošću, npr. korišćenjem ASCII kodova karaktera. Algoritam šifrovanja bi mogao da na vrednost svakog karaktera doda vrednost 15. Pri dešifrovanju, potrebno je oduzeti vrednost 15 da bi se dobio originalni karakter. U ovom primeru, broj 15 ima ulogu ključa šifrovanja. Ovakav algoritam šifrovanja je ilustrovan u tabeli.

Originalna slovna vrednost	Originalna brojna vrednost (ob)	Šifrovana brojna vrednost (sb = ob + 15)	Dešifrovana brojna vrednost (sb – 15)	Dešifrovana slovna vrednost
S	83	98	83	S
r	114	129	114	r
b	98	113	98	b
i	105	120	105	i
j	106	121	106	j
a	97	112	97	a

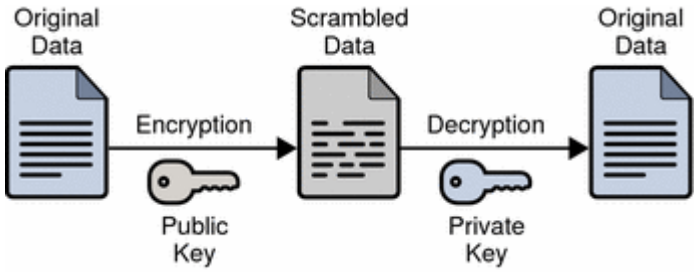
Algoritme šifrovanja možemo klasifikovati u dve grupe:

- **simetrični** algoritmi

- **asimetrični** algoritmi

Simetrični algoritmi koriste isti ključ i za šifrovanje i za dešifrovanje poruke. Algoritam iz prethodnog primera je simetričan.

Kod asimetričnih algoritama, postoje dva različita ključa. Jedan se koristi za šifrovanje poruke, a drugi za dešifrovanje. Ključ koji se koristi za šifrovanje poruke se naziva **javni ključ** (eng. *public key*) , jer je poznat i javno dostupan, tako da svako može da izvrši šifrovanje poruke. Ključ koji se koristi za dešifrovanje poruke je **privatni ključ** (eng. *private key*) i dostupan je samo vlasniku ključeva, koji jedini može da izvrši dešifrovanje poruke. Asimetrični algoritam šifrovanja je ilustrovan na slici.



* preuzeto sa https://docs.oracle.com

3. Načini autentikacije u veb aplikaciji

Kao što smo naveli, autentikacija se standardno vrši tako što klijent putem korisničkog imena i lozinke informiše server o svom identitetu. Za svaku akciju koju klijent sprovodi, server mora da ima informaciju o klijentovom identitetu. Kao što znamo, HTTP protokol je *stateless* protokol, što će reći da se svaki novi zahtev tretira nezavisno od prethodnih zahteva. Ako je klijent uputio zahtev za logovanjem u kojem je poslao svoje kredencijale, a nakon toga zahtev za izvršenje određene funkcionalnosti, ne postoji direktna veza između ta dva zahteva i server ne bi mogao da donese odluku da li korisnik ima pravo da drugi zahtev izvrši.

Osnovna autentikacija

Jedan način za rešenje pomenutog problema je takozvana **osnovna autentikacija** (eng. *basic authentication*). Kod ovog tipa autentikacije, klijent uz svaki zahtev u zaglavlju mora da dostavi i korisničko ime i lozinku korisnika koji zahtev izvršava. Time server za svaki zahtev jednostavno može da utvrdi koji korisnik ga izvršava. Bez dodatnih mehanizama zaštite, ovaj tip autentikacije je vrlo nesiguran obzirom da se presretanjem zahteva može doći do informacije o korisničkom imenu i lozinki korisnika.

Cookies

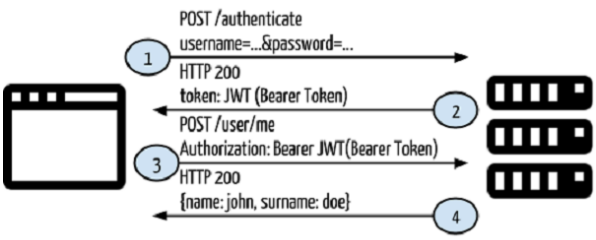
Drugi način rešavanja problema je da klijent serveru pošalje korisničko ime i lozinku samo jednom, pri prijavi na sistem. Za svaki naredni zahtev, server mora da utvrdi da li dolazi od prijavljenog korisnika. Jedan način da se ovo realizuje je korišćenje *cookies*. *Cookie* predstavlja string koji server izgeneriše pri prijavi korisnika i koji se klijentu vrati kao odgovor na prijavu. Internet *browseri* standardno podržavaju korišćenje *cookies* za

komunikaciju između klijenta i servera. *Cookie* dobijen od servera *browser* skladišti u lokalnom računaru i šalje ponovo serveru pri svakom zahtevu. Server mora da ima mehanizam da dobijeni *cookie* asocira sa podacima o korisniku koji su poslani pri prijavi. Ovo se najčešće rešava tako što server u memoriji sadrži mapu koja skladišti parove *cookie*-korisnik. Na ovaj način, kada preuzme *cookie* iz zahteva, server može da utvrdi na kojeg se korisnika odnosi i da sprovede kontrolu pristupa. *Cookie* obično ima ograničen rok važenja i kaže se da *cookie* identifikuje jednu sesiju korisnika. Slika ilustruje komunikaciju između klijenta i servera pri korišćenju *cookies*.

* preuzeto sa https://docs.microsoft.com

Autentikacija bazirana na tokenima

Mana korišćenja *cookies* je u tome što server mora da čuva informacije o sesiji korisnika time što skladišti informaciju o vezi između određenog *cookie* stringa i korisnika. Ovo je potencijalni problem ako aplikaciji pristupa veliki broj korisnika. Iz tog razloga se nastoji da u skladu sa *stateless* prirodom HTTP protokola i komunikacija bude potpuno *stateless*, tako da server ne čuva informacije o sesiji korisnika. Jedan način da se ovo realizuje je korišćenjem bezbednosnog tokena (eng. *security* token) umesto *cookies*. Token je, kao i *cookie*, string, ali umesto da identifikuje sesiju, token identifikuje direktno korisnika time što u sebi skladišti informacije o samom korisniku. Iz tog razloga, server ne mora da skladišti dodatne informacije o sesiji korisnika, tako da je komunikacija potpuno *stateless*, što poboljšava performanse servera. Dodatna prednost pri korišćenju tokena je ta što su *cookies* automatski upravljani od strane *browsera*, što uvodi određena ograničenja pri upravljanju komunikacijom. Takođe, treba uzeti u obzir i da u modernim sistemima *browseri* nisu jedine klijentske aplikacije koje pristupaju serverskom API-ju (to može biti i mobilna aplikacija ili neki proizvoljan klijentski program). Autentikacija bazirana na tokenima je ilustrovana na slici.

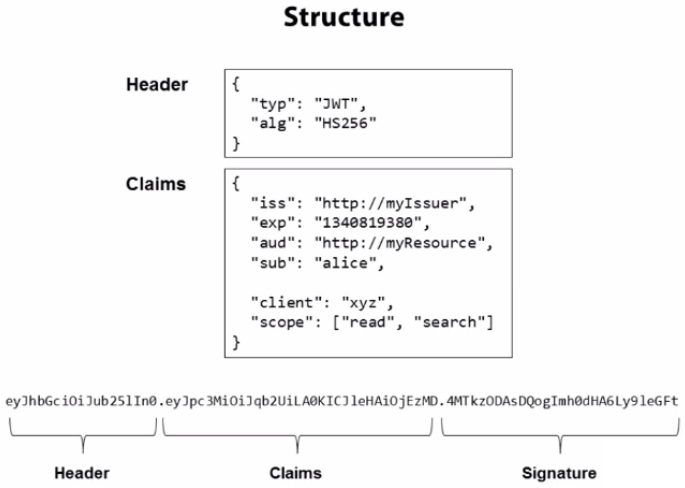


* preuzeto sa https://code.tutsplus.com

Trenutno dominantan format za reprezentaciju tokena kod autentikacije bazirane na tokenima su **JSON Web Tokens** (JWT). JWT je string koji se sastoji iz tri dela:

- zaglavlje - tip tokena i algoritam kriptovanja. Služi da server ima informaciju kako da obradi token
- glavni sadržaj - subjekat na koga se token odnosi, uloge subjekta, rok trajanja. Iz ovog dela tokena server dobija informaciju o identitetu korisnika i u skladu sa tim sprovodi kontrolu pristupa
- potpis - string generisan šifrovanjem zaglavlja, sadržaja i serverove tajne lozinke. Služi da onemogući da maliciozan korisnik sam izgeneriše token i tim tokenom proba da pristupi aplikaciji. Samo server može da izvrši proveru potpisa tokena, obzirom da je potpis kreiran korišćenjem serverove tajne lozinke. Server pri prijemu tokena vrši validaciju tokena pri kojoj utvrđuje da li je token kreiran od strane servera, kao i da li je rok važenja tokena prošao.

Struktura JWT je ilustrovana na sledećoj slici.



U gornjem delu slike vidimo da token u JSON notaciji evidentira zaglavlje i glavni sadržaj. Na dnu slike je sam token koji sadrži elemente tokena razdvojene tačkom. Vidimo da je od svih podataka formiran jedan string enkodovan prema Base64 šemi.

4. Spring Security

Spring Security je projekat koji pojednostavljuje implementaciju bezbednosti u Spring aplikacijama. Projekat pokriva autentikaciju i autorizaciju na nivou veb zahteva, ali i poziva pojedinačnih metoda. Sama implementacija bezbednosti se umnogome zasniva na tehnikama deklarativnog programiranja. Spring Security sadrži sledeće module:

- Modul za konfiguraciju - podrška za konfigurisanje bezbednosti u aplikaciji kroz XML ili Java anotacije
- Jezgro - biblioteka sa ključnim funkcijama za bezbednost
- Kriptografija - podrška za kriptovanje šifara
- Veb - podrška za bezbednost veb aplikacije kroz filtere za procesiranje zahteva

Spring Security implementira bezbednost u veb aplikaciji kroz presretanje veb zahteva putem filtera. Filteri su Spring komponente koje se mogu ugrađivati u standardni tok podataka u Spring aplikaciji kako bi izvršili dodatno procesiranje podataka. Spring sadrži generički filter *DelegatingFilterProxy* koji može da delegira posao specifičnom filteru. Spring Security tu može da injektuje svoj specifičan filter *springSecurityFilterChain* koji sprovodi kontrolu prava pristupa. Ovaj filter se automatski injektuje u Spring Boot aplikaciji kada postavimo anotaciju @EnableWebSecurity u konfiguracionu klasu.

Kontrola prava pristupa

U konfiguracionoj klasi se može definisati autorizacija u aplikaciji. Potrebno je da konfiguraciona klasa nasledi klasu *WebSecurityConfigurerAdapter* i da redefiniše metodu *configure*. U ovoj metodi se konfiguriše bezbednost kojom upravlja Spring Security. Između ostalog, može se definisati kojem URL-u u aplikaciji određeni korisnik može da pristupi. URL-ovi se definišu preko tzv. *ant matchers*, što su šabloni stringova kakve koristi Ant alat za izgradnju softvera. Kontrola pristupa se može definisati za proizvoljnog korisnika, samo ulogovanog korisnika ili korisnika koji pripada određenoj grupi korisnika. Pogledajmo primer konfiguracije autorizacije u Spring aplikaciji korišćenjem Spring Security.

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception {
        httpSecurity
          .authorizeRequests()
          .antMatchers("/api/login", "/api/register").permitAll()
          . antMatchers(HttpMethod.POST, "/api/**").hasAuthority("ROLE_ADMIN")
          .anyRequest().authenticated();
    }

}
```

Prikazana konfiguraciona klasa definiše da su URL-ovi `/api/login` i `/api/register` dostupni svim korisnicima. HTTP POST metode može da izvršava samo korisnik sa ulogom `ROLE_ADMIN`, dok svi ostali zahtevi omogućeni samo ulogovanim (autentikovanim) korisnicima.

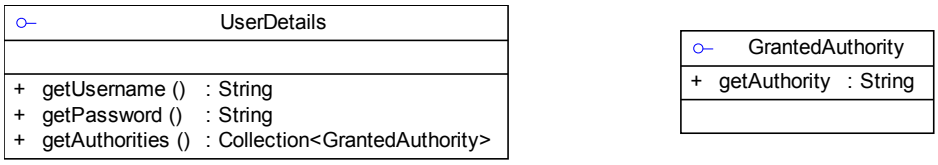
Drugi način definisanja autorizacije u aplikaciji je anotiranjem metoda koje obrađuju klijentske zahteve anotacijom @PreAuthorize. Ova anotacija specificira da će pre izvršavanja metode biti provereno da li korisnik koji je uputio zahtev ima odgovarajuća prava. Da bi Spring Security koristio ovu anotaciju, neophodno je u konfiguracionoj klasi dodati anotaciju `@EnableGlobalMethodSecurity(prePostEnabled = true)`. Dat je primer korišćenja anotacije @PreAuthorize.

```
@PreAuthorize("hasRole('ROLE_MANAGER')")
@RequestMapping(value = "api/payment/{id}", method = RequestMethod.DELETE,
      consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Void> delete(@PathVariable Long id) {
      ...
}
```

Prikazana metoda će biti izvršena samo ako Spring Security utvrdi da je klijent koji izvršava zahtev ranije ulogovan i da ima ulogu ROLE_MANAGER.

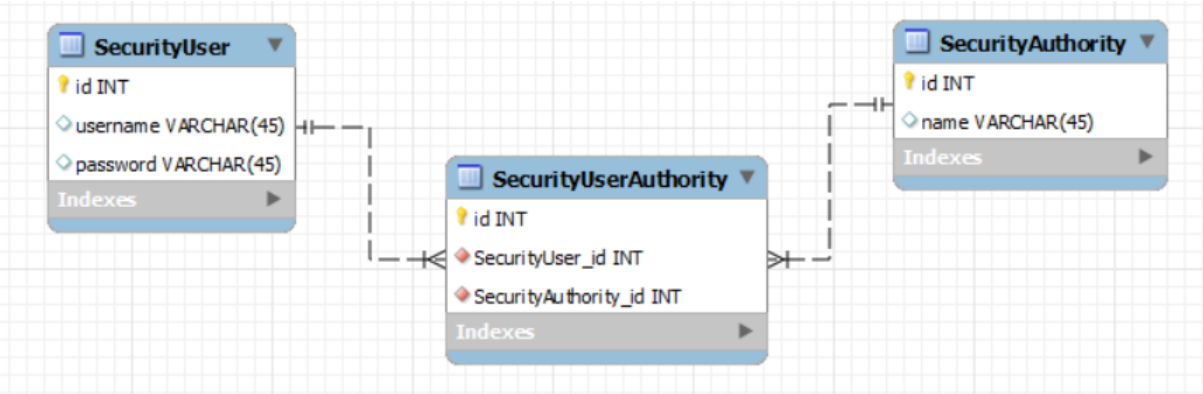
Model korisnika

Vidimo da autorizacija može da zavisi od toga koju ulogu korisnik ima u sistemu. Takođe, za autentikaciju korisnika je neophodno utvrditi ko je korisnik koji se prijavljuje. Iz tog razloga, u aplikaciji je neophodno uskladištiti informacije o korisnicima koji mogu aplikaciju da koriste, kao i o njihovim pravima pristupa. Spring Security predviđa sledeći model korisnika i njihovih prava pristupa kakav je ilustrovan na slici.



Vidimo da Spring Security vodi evidenciju o korisnicima (interfejs UserDetails) i njihovim mogućim ulogama (interfejs GrantedAuthority). Jedan korisnik može imati više uloga u aplikaciji. Za autentikaciju i autorizaciju, neophodno je da Spring Security modul sadrži objekat tipa UserDetails. Obzirom da je UserDetails samo interfejs, postoji klasa User koja implementira ovaj interfejs i koja se može koristiti za reprezentaciju korisnika. Ovaj objekat se instancira na osnovu podataka o korisnicima uskladištenim u sistemu. Pri logovanju, mora se utvrditi identitet korisnika i uloge koje može da obavlja. Sistem može na različite načine da vrši skladištenje podataka o korisnicima, kao i autentikaciju na osnovu uskladištenih podataka. Bez obzira na na način autentikacije, neophodno je instancirati objekat klase User iz Spring Security modula, da bi ovaj modul mogao da preuzme upravljanje bezbednošću aplikacije.

Pogledajmo primer modela koji se može koristiti za evidenciju korisnika, ako se koristi relaciona baza podataka. Naglašavamo da je ovo proizvoljan model i da nije u vezi sa samim Spring Security modulom, kao model prikazan na prethodnoj slici.



Korisnici su predstavljeni tabelom SecurityUser, a moguće uloge u sistemu tabelom SecurityAuthority. Obzirom da jedan korisnik može imati više uloga, kao i da u jednoj ulozi može biti više korisnika, veza između ove dve klase je N:N, pa se koristi vezna tabela SecurityUserAuthority da definiše koje sve uloge određeni korisnik ima.

Utvrđivanje identiteta korisnika

Pri prijavi u aplikaciju, neophodno je utvrditi da li se u tabeli SecurityUser nalazi korisnik sa definisanim kredencijalima. Kod za autentikaciju je predstavljen interfejsom

UserDetailsService iz Spring Security modula. Reč je o interfejsu, jer Spring Security omogućuje implementaciju proizvoljnog načina autentikacije u klasi koja implementira ovaj interfejs. Interfejs predviđa samo jednu metodu čija je deklaracija data u nastavku.

```
UserDetails loadUserByUsername(String username);
```

Pri redefiniciji ove metode, neophodno je učitati podatke o korisniku na osnovu korisničkog imena. Takođe, potrebno je učitati i uloge koje korisnik ima u sistemu kako bi se dalje mogla sprovesti autorizacija. Vidimo da kao rezultat, metoda mora da vrati objekat tipa UserDetails. Dat je primer redefinicije ove metode, ako aplikacija podatke o korisnicima skladišti u modelu kakav je prikazan na prethodnoj slici.

```
public class UserDetailsServiceImpl implements UserDetailsService {

  @Autowired
  private UserRepository userRepository;

  public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
   SecurityUser user = userRepository.findByUsername(username);

   if (user == null) {
      throw new UsernameNotFoundException(String.format("No user found with
            username '%s'.", username));
   } else {
      List<GrantedAuthority> grantedAuthorities = new
ArrayList<GrantedAuthority>();

      for (UserAuthority ua: user.getUserAuthorities()) {
            grantedAuthorities.add(new
                  SimpleGrantedAuthority(ua.getAuthority().getName()));
      }

      return new org.springframework.security.core.userdetails.User(
        user.getUsername(),
        user.getPassword(),
        grantedAuthorities);
   }

}
```

Metoda dobavlja podatke o korisniku korišćenjem *query* metode *findByUsername* iz Spring Data JPA repozitorijuma. Ako traženi korisnik postoji, metoda će kao rezultat vratiti objekat klase User koji će Spring Security da evidentira i dalje da koristi za sprovođenje kontrole pristupa. Vidimo da se u objekat ubacuje i informacija o ulogama koje korisnik ima (lista *grantedAuthorities*). Ova informacija je preuzeta iz modela korisnika iz ranije prikazane tabele SecurityUserAuthority.

Za korišćenje prikazanog koda za autentikaciju korisnika, neophodno je u konfiguracionoj klasi navesti da se za autentikaciju koristi klasa koja implementira UserDetailsService za autentikaciju. Takođe, potrebno je konfigurisati način način skladištenja lozinki u bazi podataka. Standardno se lozinke skladište enkodovane, tako da se uvidom u bazu podataka ne može pročitati sama lozinku. Za enkodovanje se koristi jednosmerna funkcija, tako da nije moguće od enkodovane lozinke dobiti originalnu vrednost lozinke. Provera da li je lozinka ispravna se vrši enkodovanjem unešene vrednosti i proverom da li se ona poklapa sa uskladištenom enkodovanom vrednošću za lozinku. Pogledajmo deo konfiguracione klase zadužen za specifikaciju načina autentikacije korisnika.

```
@Autowired
private UserDetailsService userDetailsService;

@Autowired
public void configureAuthentication(
      AuthenticationManagerBuilder authenticationManagerBuilder)
            throws Exception {

      authenticationManagerBuilder

            .userDetailsService(this.userDetailsService).passwordEncoder(
                  passwordEncoder());
      }

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

Vidimo da Spring injektuje ranije prikazanu klasu koja implementira UserDetailsService interfejs. Pri konfiguraciji autentikacije, definisano je da se enkodovanje lozinki vrši korišćenjem BCrypt algoritma, za šta postoji ugrađena podrška kroz BCryptPasswordEncoder klasu.

Prijava na sistem korišćenjem JWT

Do sada prikazani delovi koda omogućuju da Spring Security sprovede kontrolu prava pristupa ukoliko zna ko je korisnik koji upućuje veb zahtev. U nastavku ćemo videti kako sistem može da utvrdi od kojeg korisnika je zahtev stigao. Ranije smo objasnili da ima više načina da se ovo realizuje. Mi ćemo ovde predstaviti kako se korišćenjem Spring Security modula može obezbediti autentikacija koja koristi JWT.

Prvi korak pri ovakvom načinu autentikacije je prijava korisnika na sistem. Pri prijavi, neophodno je da server proveri da li su korisničko ime i lozinka ispravni i ako jesu da izgeneriše JWT koji će vratiti kao odgovor klijentu. Metoda Spring kontrolera koja ovo obavlja je data u nastavku.

```
@RequestMapping(value = "/api/login", method = RequestMethod.POST)
public ResponseEntity<String> login(@RequestBody LoginDTO loginDTO) {
   try {
      UsernamePasswordAuthenticationToken token = new
         UsernamePasswordAuthenticationToken(
       loginDTO.getUsername(), loginDTO.getPassword());

       Authentication authentication = authenticationManager.authenticate(token);
       SecurityContextHolder.getContext().setAuthentication(authentication);

       UserDetails details = userDetailsService.
             loadUserByUsername(loginDTO.getUsername());
       return new ResponseEntity<String>(tokenUtils.generateToken(details),
       HttpStatus.OK);
   } catch (Exception ex) {
       return new ResponseEntity<String>("Invalid login", HttpStatus.BAD_REQUEST);
   }
}
```

Metoda formira *Authentication* objekat na osnovu korisničkog imena i lozinke koje je klijent poslao, a koje je server deserijalizovao kao objekat klase LoginDTO. Ovaj objekat se postavlja u Spring Security kontekst čime ovaj modul dobija informaciju o korisniku. Na

osnovu ovih podataka, Spring Security će automatski pozvati metodu loadUserByUsername iz klase koja implementira UserDetailsService kako bi proverio da li korisnik postoji, kao i da li se njegova lozinka poklapa sa lozinkom postavljenom u *Authentication* objekat.

Ukoliko su podaci u prijavi ispravni, server može da izgeneriše JWT. JWT treba da sadrži i podatke o detaljima korisnika, pa se iz tog razloga još jednom eksplicitno poziva metoda loadUserByUsername, kako bi se ti podaci dobavili. Generisanje JWT je u datom primeru delegirano posebnom objektu pod nazivom *tokenUtils* koji je injektovan u kontroler. Ove nećemo ulaziti u detalje implementacije ove klase.

Utvrđivanje pošiljaoca zahteva korišćenjem JWT

Kao što smo ranije objasnili, kod autentikacije korišćenjem JWT, klijent pri svakom zahtevu iznova šalje serveru token koji je od servera dobio pri prijavi. Server je dužan da preuzme token iz zahteva i da uvidom u sadržaj tokena utvrdi da li je token validan, kao i na kojeg se korisnika odnosi. Korišćenjem Spring Security, ovu funkcionalnost je moguće realizovati uvođenjem dodatnog filtera koji će presresti klijentov zahtev i izvršiti proveru tokena. Pogledajmo deo koda konfiguracione klase koji uvodi dodatni filter pri obradi klijentovog zahteva.

```
@Bean
public AuthenticationTokenFilter authenticationTokenFilterBean() throws Exception {

	AuthenticationTokenFilter authenticationTokenFilter = new
	AuthenticationTokenFilter();

	authenticationTokenFilter.setAuthenticationManager(
	authenticationManagerBean());

	return authenticationTokenFilter;
}

@Override
protected void configure(HttpSecurity httpSecurity) throws Exception {
	...

	httpSecurity.addFilterBefore(authenticationTokenFilterBean(),
		UsernamePasswordAuthenticationFilter.class);
}
```

Klasa koja predstavlja filter za obradu tokena je nazvana *AuthenticationTokenFilter* i u prikazanoj konfiguracionoj klasi se definiše kao Spring Bean. Vidimo da se na kraju metode *configure* ovaj filter ugrađuje u standardan Spring tok obrade veb zahteva. Filter će biti postavljen pre ugrađenog Spring filtera koji vrši autentikaciju na osnovu korisničkog imena i lozinke korisnika.

Kao što smo objasnili, zadatak novouvedenog filtera je da preuzme i validira JWT token iz klijentovog zahteva. U nastavku je dat kod ove klase.

```
public class AuthenticationTokenFilter extends UsernamePasswordAuthenticationFilter
{
   @Autowired
   private TokenUtils tokenUtils;

   @Autowired
   private UserDetailsService userDetailsService;

   @Override
   public void doFilter(ServletRequest request, ServletResponse response,
```

```
        FilterChain chain) throws IOException, ServletException {

       HttpServletRequest httpRequest = (HttpServletRequest) request;
       String authToken = httpRequest.getHeader("X-Auth-Token");
       String username = tokenUtils.getUsernameFromToken(authToken);

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

          UserDetails userDetails = userDetailsService.loadUserByUsername(username);


          if (tokenUtils.validateToken(authToken, userDetails)) {
             UsernamePasswordAuthenticationToken authentication = new
               UsernamePasswordAuthenticationToken(userDetails, null,
                                 userDetails.getAuthorities());
             authentication.setDetails(new WebAuthenticationDetailsSource()
.buildDetails(httpRequest));

               SecurityContextHolder.getContext().setAuthentication(authentication);
           }
        }
        chain.doFilter(request, response);
    }
}
```

Metoda *doFilter* je zadužena za presretanje klijentovog zahteva. Vidimo da se na njenom početku preuzima token iz polja X-Auth-Token u zaglavlju HTTP zahteva. Pomoćna klasa TokenUtils je zadužena za preuzimanje korisničkog imena koje je upisano u sadržaju tokena. Na osnovu korisničkog imena se korišćenjem klase koja nasleđuje *UserDetailsService* preuzimaju ostali podaci o korisniku. Metoda *validateToken* klase *TokenUtils* je zadužena da proveri da li je token ispravan. Ako jeste, instancira se objekat klase *UsernamePasswordAuthenticationToken* iz Spring Security modula. Ovaj objekat je potreban Spring Security modulu da bi mogao dalje da upravlja autentikacijom i autorizacijom u aplikaciji. Iz tog razloga se ovaj objekat postavlja u Spring Security kontekst. Dakle, zadatak filtera je samo da na osnovu sadržaja tokena pruži Spring Security modulu informaciju o tome koji korisnik je uputio zahtev. Nakon toga, Spring Security će preuzeti upravljanje kontrolom pristupa resursima u aplikaciji.

5. HTTPS

Na početku lekcije smo objasnili da je, pored kontrole pristupa, važan aspekt bezbednosti i sprečavanje neovlašćenog pristupa porukama koje se razmenjuju između strana u komunikaciji. U slučaju veb aplikacija, klijent i server razmenjuju zahteve i odgovore putem HTTP protokola. Neovlašćena osoba bi mogla da ima uvid u podatke koji se razmenjuju ukoliko bi presrela HTTP poruku. Na taj način bi se mogli preuzeti i podaci za autentikaciju koji se razmenjuju (korisničko ime, lozinka, token), čime bi neovlašćena osoba mogla da pristupi funkcionalnostima aplikacije za koja nema pravo pristupa.

Iz tog razloga se, umesto klasičnog HTTP protokola, koristi njegova sigurnija varijanta pod nazivom HTTPS (HTTP Secure). HTTPS predviđa komunikaciju putem klasičnog HTTP protokola, s tim da su HTTP zahtev i odgovor šifrovani. Šifrovanje se obavlja prema SSL/ TSL protokolu. Ovaj protokol predviđa šifrovanje podataka po simetričnom algoritmu korišćenjem ključa koji je izgenerisan od strane klijenta. Klijent dostavlja ključ za šifrovanje serveru i obe strane koriste taj ključ kako za šifrovanje podataka pri slanju, tako i za dešifrovanje podataka pri prijemu.

Dakle, nakon što server dobije ključ koji je klijent izgenerisao, razmena poruka je sigurna. Postavlja se pitanje šta će se desiti ako neovlašćena osoba presretne upravo poruku u kojoj se serveru šalje ključ. Očigledno, tada bi ona mogla da pročita i sve naredne poruke, obzirom da poseduje ključ za dešifrovanje. Takođe, mogla bi da klijentu šalje maliciozne

odgovore, obzirom da putem ključa može da izvrši šifrovanje poruke. Iz pomenutog razloga, slanje ključa serveru zahteva dodatnu proceduru.

Naime, neophodno je i da se ključ pri slanju serveru šifruje. U ovu svrhu se koristi asimetrični algoritam šifrovanja. Kao što smo ranije objasnili, pri ovom algoritmu se koriste dva ključa, javni i privatni. Šifrovanje se vrši javnim ključem, a dešifrovanje privatnim ključem. Klijent najpre dobavlja od servera serverov javni ključ. Korišćenjem ovog ključa klijent može da izvrši šifrovanje poruke (poruka je simetrični ključ koji je klijent izgenerisao i koji će se koristiti za šifrovanje/dešifrovanje budućih zahteva/ odgovora). Za dešifrovanje ove poruke neophodan je privatni ključ koji poseduje samo server. Ukoliko bi poruku presrela neovlašćena osoba, ne bi mogla da dešifruje poruku jer nema privatni ključ. Na ovaj način samo server može da dešifruje poruku u kojoj se nalazi simetrični ključ koji će se na dalje koristiti za šifrovanje, odnosno dešifrovanje poruka koje se razmenjuju.

Pomenuti način komunikacije ima još jedan potencijalni problem koji HTTPS uspešno rešava. Naime, pomenuli smo da za razmenu simetričnog ključa, server mora klijentu da pošalje svoj javni ključ koji će klijent koristiti za šifrovanje poruke. Ukoliko bi neovlašćena osoba presrela zahtev za dobijanje javnog ključa, ona bi mogla da klijentu vrati odgovor koji sadrži javni ključ koji je ona izgenerisala. Kada klijent tim javnim ključem šifruje poruku, neovlašćena osoba bi presretanjem te poruke mogla da je dešifruje svojim privatnim ključem (obzirom da je šifrovana njenim javnim ključem).

Iz tog razloga je za uspešnu HTTPS komunikaciju neophodno da klijent bude siguran da je javni ključ koji je primio sigurno poslan od strane servera sa kojim komunicira. Za ovu svrhu se koriste SSL sertifikati. Sertifikat je fajl koji sadrži podatke o vlasniku (npr. google) i vlasnikov javni ključ. Sertifikate izdaje ograničen broj pouzdanih institucija koje vrše digitalno potpisivanje sertifikata, tako da internet *browseri* mogu da verifikuju da li je sertifikat zaista izdat od ovlašćene institucije. Na ovaj način, ukoliko bi neovlašćena osoba sama izgenerisala sertifikat u kojem tvrdi da je npr. google, *browser* bi utvrdio da to nije sertifikat čiji je vlasnik *google* i komunikacija ne bi bila uspostavljena. Za kraj, postavimo još pitanje da li bi sigurnost komunikacije bila ugrožena ako bi neovlašćena osoba poslala klijentu validan serverov sertifikat (npr. google sertifikat, obzirom da je sam sertifikat javan)? Sigurnost ne bi bila ugrožena, jer podatke šifrovane tim javnim ključem neovlašćena osoba ne bi mogla da pročita obzirom da nema odgovarajući privatni ključ (privatni ključ ima samo server koji je vlasnik sertifikata, tj. google u ovom primeru).

6. HTTPS u Spring Boot aplikaciji

Za kraj ove lekcije, opisaćemo kako se Spring Boot aplikacija može konfigurisati da koristi HTTPS komunikaciju između klijenta i servera. Kao što smo objasnili, za uspostavljanje HTTPS komunikacije neophodno je da server sadrži sertifikat koji će dostaviti klijentu. Sertifikat se može kupiti od ovlašćene institucije ili samostalno izgenerisati (tzv. *self-signed certificate*). Treba naglasiti da u slučaju korišćenja *self-signed* sertifikata *browser* prijavljuje upozorenje korisniku da komunicira sa serverom koji nema zvanično izdat sertifikat.

Pogledajmo kako možemo samostalno izgenerisati sertifikat. Za ovu svrhu možemo koristiti alat *keytool* koji se instalira zajedno sa Java distribucijom (u *bin* folder Java instalacije). Alat se koristi iz komandne linije i u nastavku je dat primer korišćenja alata za generisanje sertifikata.

```
C:\Program Files\Java\jdk1.8.0_91\bin>keytool -genkey -alias mytestcertificate -storetype PKCS12 -keyalg RSA
 -keysize 2048 -keystore mytest.p12 -validity 3650
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Goran Savic
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:  Novi Sad
What is the name of your State or Province?
  [Unknown]:  Vojvodina
What is the two-letter country code for this unit?
  [Unknown]:  rs
Is CN=Goran Savic, OU=Unknown, O=Unknown, L=Novi Sad, ST=Vojvodina, C=rs correct?
  [no]:  yes
```

Nakon unosa informacija koje keytool zahteva, biće izgenerisan fajl mytest.p12, koji predstavlja SSL sertifikat.

Ovaj fajl je neophodno postaviti na lokaciju koja je dostupna Spring Boot aplikaciji. Prekopiraćemo fajl u Spring Boot aplikaciju u folder main/resources. U ovom folderu se nalazi i fajl application.properties, u kojem je potrebno specificirati da aplikacija koristi HTTPS. U nastavku je dat spisak stavki koje je potrebno dodati u application.properties fajl, ako je sertifikat izgenerisan sa podacima kao na prethodnoj slici, a kao lozinka unešeno *testpassword*.

```
server.port: 8443
server.ssl.key-store: classpath:mytest.p12
server.ssl.key-store-password: testpassword
server.ssl.keyStoreType: PKCS12
server.ssl.keyAlias: mytestcertificate
```

Sada će nakon pokretanja servera, komunikacija ići putem HTTPS protokola na portu 8443. Dat je primer URL-a zahteva ka serveru sa kojim se komunicira putem HTTPS protokola.

```
https://192.168.10.2:8443/api/countries
```

Internet *browser*, kao i popularni REST klijenti (npr. Postman) će automatski obaviti sve procedure koje HTTPS protokol zahteva (preuzimanje sertifikata, generisanje, šifrovanje i slanje ključa, kao i šifrovanje, odnosno dešifrovanje poruka). Takođe, Spring Boot server će automatski uraditi deo posla koji HTTPS zahteva od servera. Pomenimo još da je ako koristimo Postman alat za komunikaciju sa serverom, obzirom da je reč o *self-signed* sertifikatu, neophodno u podešavanjima isključiti verifikaciju SSL sertifikata (Settings -> SSL Certificate Verification -> OFF).