



Spring Boot en Hibernate

Security



Leerdoelen

- Wat is authenticatie en wat is autorisatie
- Form-based authentication
- Basic Auth
- Zelf een gebruiker toevoegen
- Password encoding
- Basic auth gebruiken in JS

Authorizatie vs Authenticatie

Deze twee termen worden in de volksmond vaak door elkaar gebruikt, maar maken voor ons een wezenlijk verschil.

Authenticatie

- Wie ben jij?
- Bewijs dat jij bent wie je zegt dat je bent



Authorizatie

- Okee, we weten wie je bent
- Maar mag je bij de informatie die je opvraagt?
- Heb je toestemming om een operatie uit te voeren?



Authenticatie vs Autorizatie

Authenticatie is dus het controleren wie je bent

En autorizatie is controleren of jij toestemming hebt om iets te doen

Authenticatie vs Autorisatie

Authentication



Who are you?

Validate a system is accessing by the right person

Authorization



Are you allowed to do that?

Check users' permissions to access data

Beginnen met security in Spring

Spring heeft een dependency waarmee we simpel security kunnen toepassen

- We gaan naar start.spring.io en voegen de volgende dependencies toe:
 - Spring Security
 - Spring Web
 - PostgreSQL driver
 - Spring data JPA

Eerst een endpoint

Maak een simpel endpoint in een controller welke een `GetMapping` heeft naar de root. Bijvoorbeeld:

```
1  package com.example.securityles.controller;
2
3  import org.springframework.security.access.prepost.PreAuthorize;
4  import org.springframework.web.bind.annotation.CrossOrigin;
5  import org.springframework.web.bind.annotation.GetMapping;
6  import org.springframework.web.bind.annotation.RestController;
7
8  @RestController
9  @CrossOrigin
10 public class ResourceController {
11     @GetMapping("/") // root
12     public String home() {
13         return "<h1>Home</h1>";
14     }
15 }
```

En draaien

- Als we de applicatie draaien, zien we in de console het volgende bericht verschijnen:

```
Using generated security password: 4670ebb7-10cb-46ad-967d-bb48e3af37bf
```

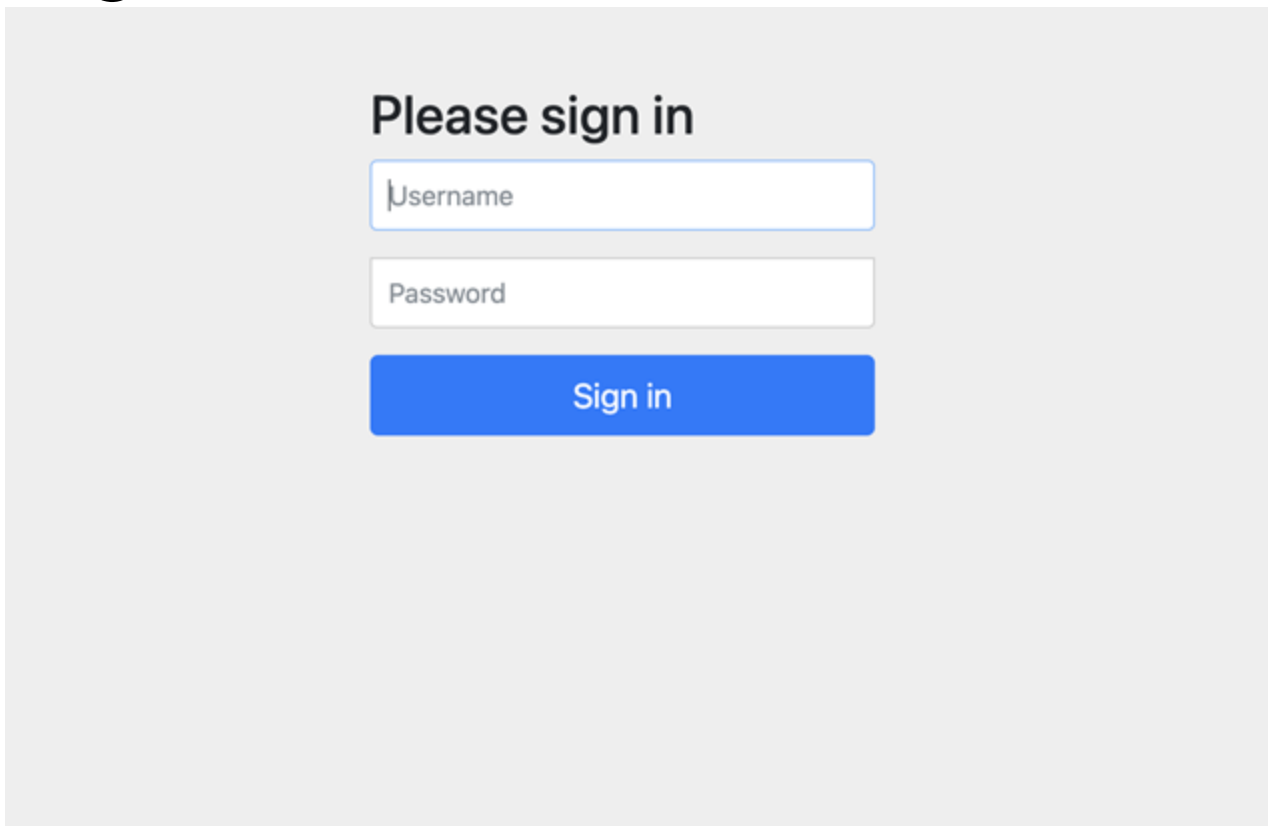
```
This generated password is for development use only. Your security configuration must be updated before running your application in production.
```

- De applicatie heeft een wachtwoord gegenereerd
- Verder een waarschuwing om dit niet in productie te gebruiken en alleen tijdens het ontwikkelen

Endpoint aanspreken

- Laten we een browser openen en naar localhost:8080/ gaan, waar onze applicatie draait.
- We verwachten hier ons stukje HTML uit ons endpoint te zien
- Maar...

Een loginscherm



Please sign in

Username

Password

Sign in

Spring Security

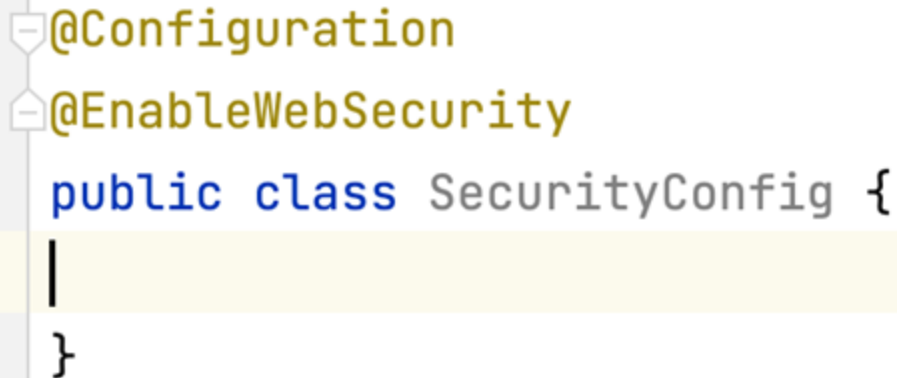
- Dit is een standaard inlogschermb van Spring Security
- Ziet er prima uit, maar dit is niet helemaal wat we willen
- Als we via httpie proberen dit endpoint aan te spreken, krijgen we hetzelfde scherm te zien
- Ook niet zo handig als we vanuit een frontend dit endpoint aanspreken
- Gelukkig kunnen we dit gedrag aanpassen!

Spring Security

- We maken een nieuwe package in ons project: configuration
- Daarin maken we een nieuwe klasse: SecurityConfig
- Boven de klasse zetten we twee annotaties:
 - @Configuration
 - @EnableWebSecurity
- Met @Configuration geven we aan dat deze klasse onderdeel is van het Spring framework
- Met @EnableWebSecurity geven we dat we in deze klasse de beveiliging van onze applicatie gaat inrichten

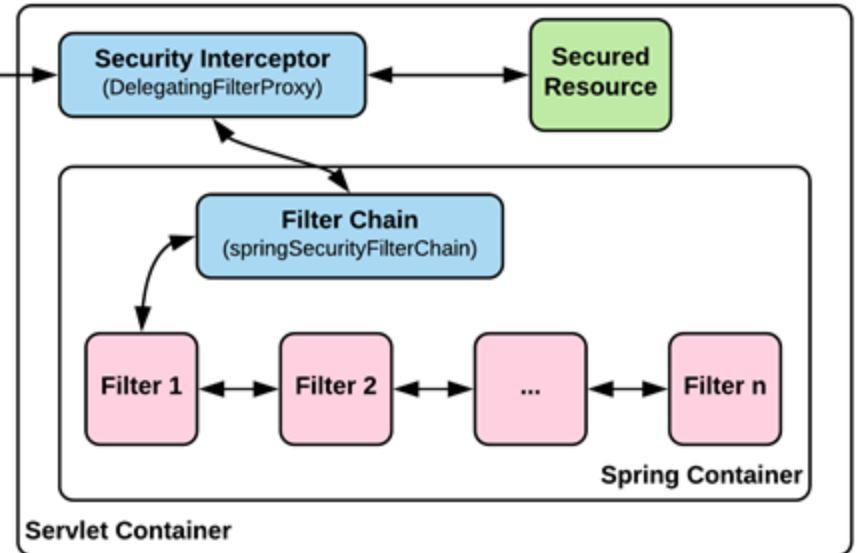
Spring Security

De basis van onze configuratie!

```
A code snippet for Spring Security configuration. It features a light gray background with a vertical line on the left. Two annotations, @Configuration and @EnableWebSecurity, are shown with small shield icons to their left. The class name SecurityConfig is in blue. The opening curly brace of the class is on a yellow background, and the closing curly brace is on a gray background.  
@Configuration  
@EnableWebSecurity  
public class SecurityConfig {  
|  
}
```

En nu?

- We kunnen beginnen met configureren, maar waar beginnen we?
- We gebruiken een filter om toegang tot endpoints te beheren
- Deze filter komt tussen de gebruiker en onze endpoints in te staan
- Laten we eerst een tweede endpoint maken
- Dit endpoint gaan we zo beveiligen



Tweede endpoint

```
3 @RestController
4
5 public class ResourceController {
6
7     @GetMapping("/")
8     public String home() { return ("<h1>Welcome</h1>"); }
9
10    @GetMapping("/user")
11    public String user() {
12        return ("<h1>User</h1>");
13    }
14 }
```

Security Filter

- We kunnen nu onze filter op gaan bouwen
- We maken in de SecurityConfiguration klasse een nieuwe methode, filterChain
- Deze methode
 - neemt een HttpSecurity object
 - throws een Exception
 - returned een SecurityFilterChain
 - Is een @Bean

Security Filter

@Configuration

@EnableWebSecurity

```
public class SecurityConfig {
```

@Bean

```
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
```

```
}
```

```
|
```

```
}
```

Bonen?

- We hebben de annotatie @Bean gebruikt voor de securityfilter
- Maar wat hebben bonen te maken met methodes?
- Beans zijn objecten en methodes die door het Spring framework gemanaged worden
- Doordat we de SecurityConfiguration klasse met @Configuration gemarkeerd hebben, gaat Spring op zoek naar @Bean 's in die klasse
- Spring zorgt ervoor dat deze methodes beschikbaar zijn voor de applicatie

Filters maken

- We kunnen nu in de `filterChain` methode regels gaan maken waar het verkeer van onze applicatie zo aan moet voldoen.
- We gaan matchen op urls van onze endpoints en kunnen gebruikers met een rol (Authorizatie) toegang geven tot deze endpoints
- Dit doen we met behulp van method chaining
- We kijken eerst even naar een voorbeeld

Filters maken

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests((requests) -> requests
            .requestMatchers(...patterns: "/user").hasRole(role: "USER")
        );
    return http.build();
}
```

Filters maken

- We pakken het `HttpSecurity` object wat binnenkomt
- We geven eerst aan dat http requests geauthoriseerd moeten worden.
- Vervolgens matchen we op een url met `requestMatchers(url)`
- Wat opvalt is dat deze methode direct op een andere methode wordt aangeroepen. Dit heet method chaining.
- Daarna geven we aan welke rol toegang heeft tot de url, namelijk de `USER` rol.
- Als laatste bouwen we het resultaat van onze method chain en returnen dit.

Terug naar de applicatie

- Als we nu proberen een endpoint aan te spreken in onze browser, krijgen we een 403 error.
- We zijn unauthorized om een endpoint aan te spreken
- Zelfs de root van onze applicatie ("/") kunnen we niet aanspreken. We hebben hier nog geen regels voor ingesteld, en dus is deze op slot gedaan
- Daarnaast is onze inlogformulier verdwenen

Eerst het formulier

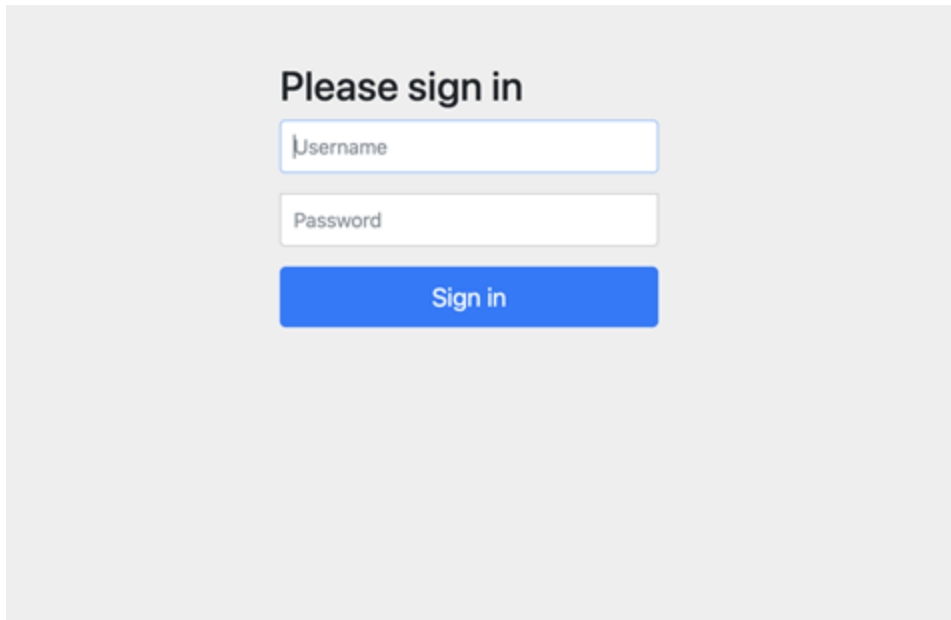
- We kunnen het inlogformulier terug krijgen door een extra regel aan onze filterchain toe te voegen
- Door achteraan onze chain `.formLogin()` toe te voegen, krijgen we het formulier terug.
- Als we dit neerzetten, word `formLogin()` rood.
- We moeten eerst het blokje van matchers afsluiten voor we aan de volgende beginnen. Dit doen we met `.and()`

Eerst het formulier

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests((requests) -> requests
            .requestMatchers(...patterns: "/user").hasRole(role: "USER")
        ).formLogin(Customizer.withDefaults());
    return http.build();
}
```

Inlog

- Ons inlogschermb is weer terug!
- Als we naar /user gaan, en inloggen krijgen we de header te zien
- Maar, als we naar / gaan, mogen we er nog steeds niet in



Please sign in

Username

Password

Sign in

Tijd voor een nieuwe regel

- We willen een wildcard hebben, zodat we op al onze endpoints die niet /user zijn iedereen toelaten
- Gelukkig kunnen we dit maken met de requestMatchers() methode door “/**” als argument mee te geven.
- Vervolgens geven we met permitAll() aan dat iedereen hierbij mag
- De volgorde van filters is belangrijk! Daar komen we zo op terug

Tijd voor een nieuwe regel

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests((requests) -> requests
            .requestMatchers(...patterns: "/user").hasRole(role: "USER")
            .requestMatchers(...patterns: "**").permitAll()
        ).formLogin(Customizer.withDefaults());
    return http.build();
}
```

We hebben weer toegang

- We kunnen weer bij onze root pagina ("/")
- We hebben aangegeven dat alleen een gebruiker met rol "USER" bij /user kan
- Maar waar komt deze rol vandaag?
- Dit is een ingebouwde rol van Spring Security
- Er is ook nog een "Admin" rol
- Laten we een extra endpoint maken en alleen een admin toegang geven

Admin Endpoint

```
@RestController
public class ResourceController {

    @GetMapping("/")
    public String home() { return "<h1>Welcome</h1>"; }

    @GetMapping("/user")
    public String user() {
        return "<h1>User</h1>";
    }

    @GetMapping("/admin")
    public String admin() {
        return "<h1>Admin</h1>";
    }
}
```

Admin regel

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests((requests) -> requests
            .requestMatchers(...patterns: "/user").hasRole(role: "USER")
            .requestMatchers(...patterns: "**").permitAll()
            .requestMatchers(...patterns: "/admin").hasRole(role: "ADMIN")
        ).formLogin(Customizer.withDefaults());
    return http.build();
}
```


CORS en CSRF

- Cross-origin resource sharing is het mechanisme waarmee bronnen op een webpagina toegankelijk zijn vanaf externe domeinen. We zetten de filters voor CORS uit.
- Cross-site resource forgery is een soort aanval waarbij iemand zich voordoeft als een geautoriseerde gebruiker en zo opdrachten uitvoert. We zetten CSRF uit.

CORS en CSRF

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .cors(crs -> crs.disable())
        .csrf(csrf -> csrf.disable())
        .authorizeHttpRequests((requests) -> requests
            .requestMatchers(...patterns: "/user").hasRole(role: "USER")
            .requestMatchers(...patterns: "/**").permitAll()
            .requestMatchers(...patterns: "/admin").hasRole(role: "ADMIN")
        ).formLogin(Customizer.withDefaults());
    return http.build();
}
```

User model

- Maak een nieuwe package genaamd "model"
- Maak een nieuwe klasse genaamd "User"
- Maak hier het model voor de User entity, welke een auto generated Long id heeft, String username, String password en String roles
- Maak het model zoals je normaal zou doen

AuthUserDetails model

- Maak nog een klasse genaamd AuthUserDetails welke de UserDetails interface implementeert
- Dit model gaat functioneren als een soort wrapper class van de User klasse en krijgt dus ook niet de @Entity annotation omdat het geen eigen tabel in de DB is
- Het handelt het authenticatie aspect achter het User model af

```
public class AuthUserDetails implements UserDetails {  
    private final User user;  
    public AuthUserDetails(User user) {  
        this.user = user;  
    }  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
        return Arrays.stream(user  
            .getRoles()  
            .split(",")  
            .map(SimpleGrantedAuthority::new)  
            .toList());  
    }  
    @Override  
    public String getPassword() {  
        return user.getPassword();  
    }  
    @Override  
    public String getUsername() {  
        return user.getUsername();  
    }  
    @Override  
    public boolean isAccountNonExpired() {  
        return true;  
    }  
    @Override  
    public boolean isAccountNonLocked() {  
        return true;  
    }  
    @Override  
    public boolean isCredentialsNonExpired() {  
        return true;  
    }  
    @Override  
    public boolean isEnabled() {  
        return true;  
    }  
}
```

UserRepository

- Maak een nieuwe package aan genaamd repository
- Maak hier een nieuwe klasse aan genaamd UserRepository
- Maak de repository zoals je normaal zou doen, maar voeg één methode toe, deze retournt een Optional<User> heet findByUsername en ontvangt een String username
- Deze methode maakt gebruik van JPA derived queries

UserRepository

```
package com.test.sec.repository;

import com.test.sec.model.User;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
}
```

User details service

- De UserDetailsService van Spring security handelt authenticatie af waarbij gebruikersnamen en wachtwoorden worden gebruikt.
- Maak een nieuwe package genaamd "service"
- Maak een nieuwe klasse genaamd "MyUserDetailsService" welke de UserDetailsService interface implementeert
- Maak een methode "loadUserByUsername" welke een UserDetails teruggeeft, een String username ontvangt en een UsernameNotFoundException kan throwen.

User details service

```
package com.test.sec.service;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
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;

import com.test.sec.model.AuthUserDetails;
import com.test.sec.model.User;
import com.test.sec.repository.UserRepository;

@Service
public class MyUserDetailsService implements UserDetailsService {

    @Autowired
    UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        Optional<User> user = userRepository.findByUsername(username);
        if (user.isEmpty()) {
            throw new UsernameNotFoundException("User not found: " + username);
        }

        return new AuthUserDetails(user.get());
    }
}
```


Update security config

- Nu voegen we de user details service toe aan onze security config en stellen we de http basis authenticatie in.

```
public class SecurityConfig {  
    @Autowired  
    MyDetailsService userDetailsService;  
  
    @Bean  
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {  
        http  
            .cors(cors -> cors.disable())  
            .csrf(csrf -> csrf.disable())  
            .authorizeHttpRequests((requests) -> requests  
                .requestMatchers(...patterns:"/**").permitAll()  
                .requestMatchers(...patterns:"/user").hasRole(role:"USER")  
                .requestMatchers(...patterns:"admin").hasRole(role:"ADMIN")  
            )  
            .userDetailsService(userDetailsService)  
            .httpBasic(Customizer.withDefaults());  
        return http.build();  
    }  
}
```

Update security config

- Voeg ook de password encoder toe (hier wordt later op teruggekomen)

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

Setup (seeder)

```
@Component
public class Setup {
    @Autowired
    private PasswordEncoder passwordEncoder;
    @Autowired
    private UserRepository userDetailsRepository;

    @EventListener
    @Transactional
    public void onApplicationEvent(ApplicationReadyEvent event) {
        User user = new User(username:"admin", passwordEncoder.encode(rawPassword:"admin"), roles:"ROLE_USER");
        userDetailsRepository.save(user);
    }
}
```

Applicatie draaien

- Nu kan je de applicatie draaien, echter zal je erachter komen dat dat niet goed gaat.
- Waarom?
- In PostgreSQL is user een reserved keyword
- Gebruik dus @Table(name = "`user`")
- Draai de applicatie opnieuw en probeer in te loggen en de endpoints te testen

Password hashing

- Wachtwoorden worden opgeslagen in de DB, maar nooit in plain text
- De wachtwoorden worden eerst gehashed en gesalt
- Een hashfunctie is een functie die van een input een fixed size output maakt
- De hashfunctie kan maar een kant op, dus van hash terug naar wachtwoord is onmogelijk.
- Echter hashed hetzelfde wachtwoord altijd naar dezelfde hash
- Wat voor probleem levert dit op?

Salting

- Voor een wachtwoord gehashed wordt, wordt een salt toegevoegd
- Een salt is een random generated waarde welke aan het wachtwoord geplakt wordt
- Vervolgens wordt deze combinatie gehashed en wordt de gebruikte salt weer aan de hash geplakt
- Zo is voor hetzelfde wachtwoord elke hash toch anders
- Hoe bevestig je dan het wachtwoord?
- De hash wordt uit de DB opgehaald en de salt wordt geëxtraheerd, de salt wordt aan het ingevoerde wachtwoord geplakt, en dit wordt gehashed. De resulterende hash wordt met de hash uit de DB vergeleken

Vragen?

- E-mail mij op voornaam.achternaam@code-cafe.nl!
- Join de Code-Café community op discord!

