Sécuriser l'API avec JWT

JWT (JSON Web Token) est un standard de sécurité qui a pour but l'authentification et l'autorisation des clients. La particularité de JWT est qu'il permet de générer un token qui non seulement authentifie le client mais contient aussi des informations signées (on peut en vérifier l'intégraté) à propos de l'utilisateur. Il s'agit aussi d'une authentification sans état (contrairement aux cookies ou à la session coté serveur), ce qui permet de respecter le caractère sans état du standard ReST. Pour sécuriser notre API, nous allons utiliser Spring Security, et nous allons y intégrer JSON Web Tokens.

Installation des dépendances

Spring Security

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

JSON Web Token

```
<dependency>
  <groupId>com.auth0</groupId>
  <artifactId>java-jwt</artifactId>
  <version>3.11.0</version>
</dependency>
```

Ensuite rajoutez le code suivante dans votre `SpringBootTodoApplication` :

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

Cela permet d'importer un algorithme de hachage (BCrypt) pour hacher les motes de passe des utilisateur dans le stockage.

Créez ensuite un package `security` afin de mettre tout notre code de configuration de sécurité.

Constantes

Pour commencer, on va créer une classe `SecurityConstantes` dans laquelle nous allons stocker toutes les valeurs dont nous auront besoin :

```
public class SecurityConstants {
    public static final String SECRET = "SecretKeyToGenJWTs";
    public static final long EXPIRATION_TIME = 864_000_000; // 10 days
    public static final String TOKEN_PREFIX = "Bearer ";
    public static final String HEADER_STRING = "Authorization";
    public static final String SIGN_UP_URL = "/api/users/sign-up";
}
```

- `SECRET` : secret utilisé pour l'algorithme cryptographique de signature du token JWT, doit être le plus random possible.
- `EXPIRATION_TIME` : temps de validité du token JWT
- `TOKEN_PREFIX` : préfixe du token JWT dans le header des requêtes. On utilise `Bearer` par convention
- `HEADER_STRING` : nom du header des requêtes utilisé pour passer le token. On utilise `Authorization` par convention
- `SIGN_UP_URL` : URL pour l'inscription des nouveaux utilisateurs

Authentification

Créez une nouvelle classe `JWTAuthenticationFilter` afin de configurer un filtre d'authentification, c'est à dire vérifier que l'utilisateur a les bons credentials pour lui délivrer un token.

Cette classe va étendre `UsernamePasswordAuthenticationFilter`. Il faut également prendre en paramètre un `AuthenticationManager` (fourni pas le conteneur d'injection de dépendances) afin de pouvoir interagir avec le framework de sécurité. Enfin, on redéfinit les deux méthodes qui nous intéressent : `attemptAuthentication` et `successfulAuthentication`. La première va s'occuper de vérifier les credentials de l'user et la deuxième va générer son token.

```
public class JWTAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
    private AuthenticationManager authenticationManager;

    public JWTAuthenticationFilter(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
```

```
    }

   @Override
    public Authentication attemptAuthentication(HttpServletRequest req,
                                                HttpServletResponse res) throws
AuthenticationException {

      }

  @Override
    protected void successfulAuthentication(HttpServletRequest req,
                                            HttpServletResponse res,
                                            FilterChain chain,
                                            Authentication auth) throws IOException,
ServletException {

      }
```

Dans la première méthode, on va récupérer les credentials de l'utilisateur dans la requête, puis on les passe au framework :

```
    @Override
    public Authentication attemptAuthentication(HttpServletRequest req,
                                                HttpServletResponse res) throws
AuthenticationException {
        try {
            ApplicationUser creds = new ObjectMapper()
                    .readValue(req.getInputStream(), ApplicationUser.class);

            return authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(
                            creds.getUsername(),
                            creds.getPassword(),
                            new ArrayList<>())
            );
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
```

Ce filtre va permettre de gérer les requête POST sur `/login` contenant un `ApplicationUser` dans le Body de la requête.

Dans la seconde, on crée le token de l'utilisateur, et on le place dans le header prévu de la réponse :

```
    @Override
    protected void successfulAuthentication(HttpServletRequest req,
                                            HttpServletResponse res,
                                            FilterChain chain,
                                            Authentication auth) throws IOException,
ServletException {

        String token = JWT.create()
                .withSubject(((User) auth.getPrincipal()).getUsername())
                .withExpiresAt(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
                .sign(HMAC512(SECRET.getBytes()));
        res.addHeader("Access-Control-Expose-Headers", "Authorization");
        res.addHeader("Access-Control-Allow-Headers", "Authorization, X-PINGOTHER, Origin, X-
Requested-With, Content-Type, Accept, X-Custom-header");
        res.addHeader(SecurityConstants.HEADER_STRING, SecurityConstants.TOKEN_PREFIX + token)
    }
}
```

On a aussi besoin d'ajouter quelques header pour dire à la sécurité du navigateur que le client a le droit de lire le header du token.

Autorisation

Créez une nouvelle classe `JWTAuthenticationFilter` afin de configurer un filtre d'autorisation, c'est à dire vérifier que l'utilisateur qui cherche à accèder une ressource protégée, prossède bien un token valide.

Cette classe va étendre `BasicAuthenticationFilter` . Il faut également prendre en paramètre un `AuthenticationManager` (fourni pas le conteneur d'injection de dépendances) afin de pouvoir interagir avec le framework de sécurité. Enfin, on redéfinit une qui nous intéresse : `doFilterInternal`. Elle va dire au framework si l'utilisateur est bien authentifié ou non. Pour éviter que cette méthode soit trop longue nous allons créer une méthode `getAuthentication` qui va s'occuper de valider le token.

```
public class JWTAuthorizationFilter extends BasicAuthenticationFilter {

    public JWTAuthorizationFilter(AuthenticationManager authManager) {
        super(authManager);
    }

    @Override
    protected void doFilterInternal(HttpServletRequest req,
                                    HttpServletResponse res,
                                    FilterChain chain) throws IOException, ServletException {
□□...
    }

    private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request)
{
□□...
    }
}
```

Dans la première méthode on va récupérer le token dans la requête et le passer à la seconde méthode :

```
@Override
protected void doFilterInternal(HttpServletRequest req,
                                HttpServletResponse res,
                                FilterChain chain) throws IOException, ServletException {
  String header = req.getHeader(HEADER_STRING);

  if (header == null || !header.startsWith(TOKEN_PREFIX)) {
    chain.doFilter(req, res);
    return;
  }

  UsernamePasswordAuthenticationToken authentication = getAuthentication(req);

  SecurityContextHolder.getContext().setAuthentication(authentication);
  chain.doFilter(req, res);
}
```

Ensuite on implémente `getAuthentication` pour valider le token :

```
private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {
  String token = request.getHeader(HEADER_STRING);
  if (token != null) {

    String user = JWT.require(Algorithm.HMAC512(SECRET.getBytes()))
      .build()
      .verify(token.replace(TOKEN_PREFIX, ""))
      .getSubject();

    if (user != null) {
      return new UsernamePasswordAuthenticationToken(user, null, new ArrayList<>());
    }
    return null;
  }
  return null;
}
```

Récupérer les utilisateurs

Maintenant, pour que l'authentification puisse marcher, il faut indiquer au framework comment trouver nos utilisateurs. On va donc créer une classe `UserDetailsServiceImpl` qui étend `UserDetailsService`.

Nous allons lui injecter notre repository JPA qui permet de retrouver les utilisateur, et l'utilisateur pour envoyer notre utilisateur au framework :

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {
    private ApplicationUserRepository applicationUserRepository;

    public UserDetailsServiceImpl(ApplicationUserRepository applicationUserRepository) {
        this.applicationUserRepository = applicationUserRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        ApplicationUser applicationUser = applicationUserRepository.findByUsername(username);
        if (applicationUser == null) {
            throw new UsernameNotFoundException(username);
```

```
        }
        return new User(applicationUser.getUsername(), applicationUser.getPassword(),
emptyList());
    }
}
```

Si l'utilisateur n'existe pas, on lance une `UsernameNotFoundException`.

Configuration de Spring Security

On va maintenant créer une classe `WebSecurity` qui va nous permettre de configurer la sécurité en faisant le lien entre tous les composants que nous avons créés ainsi que le framework.

Nous allons lui injecter notre algo de vérification des mots de passe, `BCrypt` ainsi que notre service de récupération d'utilisateur. Nous allons aussi redéfinir deux méthodes `configure` afin de relier tous les éléments dont nous avons besoin :

```
@EnableWebSecurity
public class WebSecurity extends WebSecurityConfigurerAdapter {
    private UserDetailsServiceImpl userDetailsService;
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    public WebSecurity(UserDetailsServiceImpl userDetailsService, BCryptPasswordEncoder
bCryptPasswordEncoder) {
        this.userDetailsService = userDetailsService;
        this.bCryptPasswordEncoder = bCryptPasswordEncoder;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
□□...
    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
       ...
    }
}
```

Pour la première méthode, nous allons configurer quelles routes sont protégées. On va définir celles qui ne seront pas protégé comme cas spécial, et toutes les autres seront protégées. Ici on a définit que l'URL d'inscription n'étais pas protégée.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http.cors().and().csrf().disable().authorizeRequests()
    .antMatchers(HttpMethod.POST, SIGN_UP_URL).permitAll()
    .anyRequest().authenticated()
    .and()
    .addFilter(new JWTAuthenticationFilter(authenticationManager()))
    .addFilter(new JWTAuthorizationFilter(authenticationManager()))
    .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
}
```

Pour rajouter des routes non protégées, il faut rajouter une ligne avec la méthode et l'url de la route :

```
.antMatchers(HttpMethod.POST, "mon url").permitAll()
```

Et ce juste avant `.anyRequest().authenticated()`. Ensuite nous ajoutons nos filtres and nous désactivons les session car avec JWT il n'y en a pas besoin.

Enfin, la seconde méthode `configure` sert juste à passer notre service de récupération d'utilisateurs au framework :

```
@Override
public void configure(AuthenticationManagerBuilder auth) throws Exception {
  auth.userDetailsService(userDetailsService).passwordEncoder(bCryptPasswordEncoder);
}
```

Voilà ! Notre récurité est configurée, il ne nous reste plus qu'à implémenter l'inscription des utilisateurs.

Inscription

Pour l'inscritpion il suffit de créer un endpoint qui permet d'insérer un nouvel utilisateur dans la base en utilisant la route que nous avons préparée pour ce faire dans la configuration de sécurité (POST `/api/users/sign-up`), sans oublier de hacher son mot de passe avec Bcrypt, et le tour est joué !

Revision #6
Created 2021-01-22 09:47:07 UTC by Arsène Lapostolet

Updated 2021-02-03 18:25:42 UTC by Arsène Lapostolet