

SEGURIDAD DE APLICACIONES

PRÁCTICA 2

Aitor Cavia Varela

Daniel Lorenzo Rama

Javier Ortega Antolínez

Desarrollo e implementación.....	3
Repositorio	8

Desarrollo e implementación

- a. **Terminar de implementar el endpoint de autenticación.** Este endpoint se encuentra en la clase `com.tasks.rest.UserController`, en el método `doLogin`. Este método debe devolver el token de acceso en formato JSON y un código de respuesta HTTP 200

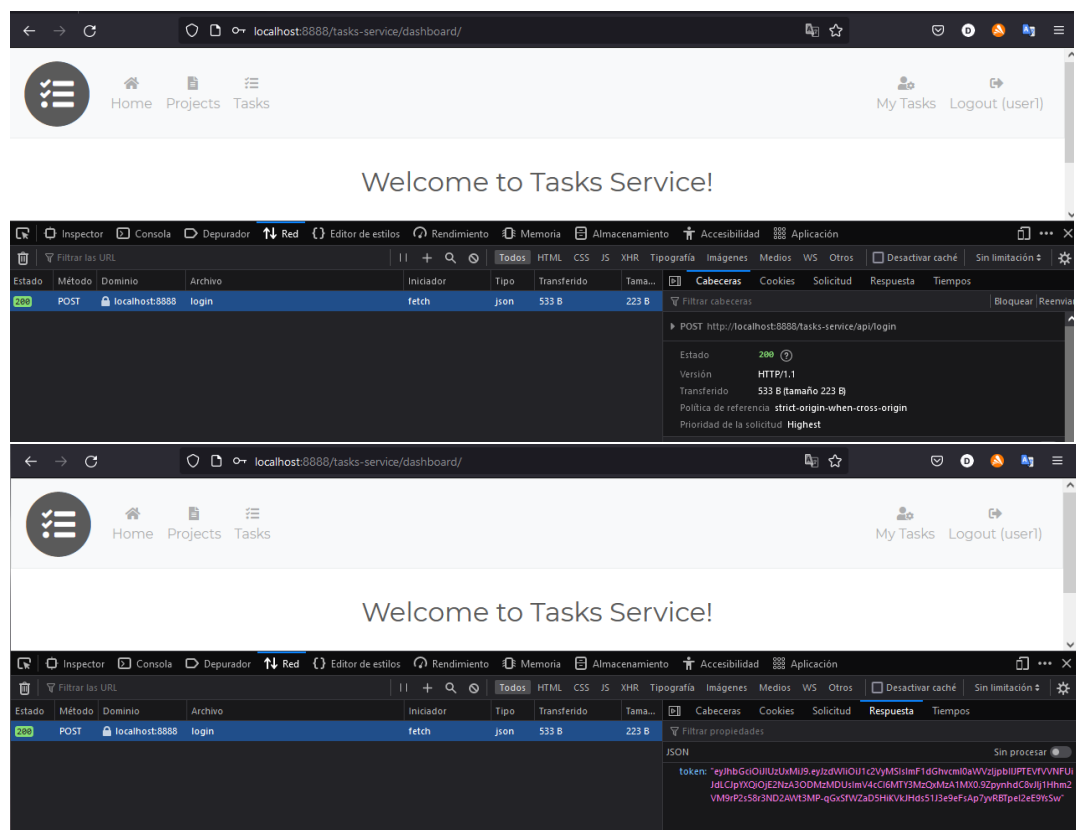
Se modifica el método “doLogin()” de la clase “UserController.java” de forma que se utilice un token autogenerado para la autenticación del usuario.

```
@RequestMapping(value = "/login", method = RequestMethod.POST)
public ResponseEntity<TokenResponse> doLogin(@RequestBody Credentials credentials)
    throws AuthenticationException {

    final Authentication authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(
            credentials.getUsername(),
            credentials.getPassword()
        )
    );
    SecurityContextHolder.getContext().setAuthentication(authentication);

    UserDetails userDetails = this.userService.loadUserByUsername(credentials.getUsername());
    String token = this.tokenProvider.generateToken(userDetails);
    return ResponseEntity.ok(new TokenResponse(token));
}
```

Para comprobar el correcto funcionamiento se realiza el login de un usuario. Se obtiene un status code 200 OK y la respuesta es un JSON:



- b. **Configurar control de acceso (excepto la parte que involucra roles). Para ello se utilizará, en la medida de lo posible, las funcionalidades proporcionadas por Spring Security y será necesario añadir los siguientes elementos en la clase `com.tasks.config.WebSecurityConfig`:**

Añadimos a la configuración el filtro que extrae y procesa el token JWT de las peticiones HTTP al gestor de seguridad de *spring*. Esto se realiza en la clase `“WebSecurityConfig.java”`

```
http.sessionManagement().and().addFilter(new
JwtAuthorizationFilter(tokenProvider,
authenticationManager()));
```

Eliminamos el servicio de cookies convirtiendo el servicio web en un servicio sin estado (stateless) mediante la siguiente línea.

```
http.sessionManagement().sessionCreationPolicy(SessionCr
eationPolicy.STATELESS);
```

```
@Override
protected void configure(HttpSecurity http) throws Exception {

    http.csrf().disable();

    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
    .and()
    .addFilter(new JwtAuthorizationFilter(tokenProvider, authenticationManager()))
    .authorizeRequests();
```

Se configuran los recursos estáticos, tanto ficheros “javascript” como “css”, librerías y demás recursos necesarios para el funcionamiento básico de la aplicación. Estos se configurarán en el método estático de configuración.

```
@Override
public void configure(WebSecurity web) throws Exception {
    web
    .ignoring()
    .antMatchers("/tasks-service/**")
    .antMatchers("/css/**")
    .antMatchers("/javascript-lib/notify/**")
    .antMatchers("/react-lib/**")
    .antMatchers("/webjars/**");
}
```

- c. **Configurar el control de acceso según los roles del usuario: es necesario comprobar el rol del usuario al ejecutar cada una de las operaciones del servicio.**

Se configuran los recursos del API REST, de acuerdo con el enunciado, atendiendo a los roles, tanto los que no requieren autorización como los que, si la quieren, y denegamos el resto de las peticiones.

Además, se añade el acceso a la documentación de la api, así como los recursos necesarios de “swagger” para su correcta imprimación.

```
http.authorizeRequests()

    .antMatchers(HttpMethod.GET, "/dashboard/**").permitAll()

    .antMatchers(HttpMethod.GET, "/swagger-ui.html").permitAll()
    .antMatchers(HttpMethod.GET, "/swagger-resources/**").permitAll()
    .antMatchers(HttpMethod.GET, "/v2/api-docs").permitAll()

    .antMatchers(HttpMethod.POST, "/api/login").permitAll()
    .antMatchers(HttpMethod.GET, "/api/users").hasRole("ADMIN")

    .antMatchers(HttpMethod.GET, "/api/tasks").permitAll()
    .antMatchers(HttpMethod.GET, "/api/tasks/{\\d+}").permitAll()
    .antMatchers(HttpMethod.POST, "/api/tasks").hasRole("ADMIN")
    .antMatchers(HttpMethod.PUT, "/api/tasks/{\\d+}").hasRole("ADMIN")
    .antMatchers(HttpMethod.DELETE, "/api/tasks/{\\d+}").hasRole("ADMIN")
    .antMatchers(HttpMethod.GET, "/api/projects/{\\d+}/tasks").permitAll()

    .antMatchers(HttpMethod.GET, "/api/projects").permitAll()
    .antMatchers(HttpMethod.GET, "/api/projects/{\\d+}").permitAll()
    .antMatchers(HttpMethod.POST, "/api/projects").hasRole("ADMIN")
    .antMatchers(HttpMethod.PUT, "/api/projects/{\\d+}").hasRole("ADMIN")
    .antMatchers(HttpMethod.DELETE, "/api/projects/{\\d+}").hasRole("ADMIN")

    .antMatchers(HttpMethod.GET, "/api/comments/{\\d+}").permitAll()
    .antMatchers(HttpMethod.POST, "/api/comments").authenticated()

    .antMatchers(HttpMethod.POST, "/api/tasks/{\\d+}/changeProgress").hasAnyRole("ADMIN", "USER")
    .antMatchers(HttpMethod.POST, "/api/tasks/{\\d+}/changeResolution").hasRole("USER")
    .antMatchers(HttpMethod.POST, "/api/tasks/{\\d+}/changeState").hasRole("ADMIN")

    .anyRequest().denyAll();
```

d. Implementar la autenticación con OAuth2: es necesario terminar la aplicación SPA para que obtenga el token JWT del servicio OAuth2.

Para el desarrollo de la implementación de autenticación a través de OAuth2, se deben seguir los siguientes pasos:

1. Realizamos los siguientes cambios para que la aplicación pueda emplear el servicio OAuth2:
 - a. Invocar el flujo implicit en la función handleSSO de *tasks-app/src/main/resources/static/application/frontend/users/Login.js*

Para llevar a cabo esta tarea se implementa la redirección de la petición del servidor OAuth <http://localhost:7777/oauth-server/oauth/authorize> que devuelve el access_token a la aplicación <http://localhost:8888/tasks-service/dashboard/loginOAuth>.

Hay que tener en cuenta que se necesitan los parámetros de “response_type”, “token_type”, “client_id” y “redirect_uri” para que el flujo funcione. El flujo implicit se implementará con la función *window.location.replace(url)*:

- url: <http://localhost:7777/oauth-server/oauth/authorize>
- “response_type”: token
- “token_type”: Bearer
- “client_id”: tasks_app

- "redirect_uri": <http://localhost:8888/tasks-service/dashboard/loginOAuth>

```
window.location.replace('http://localhost:7777/oauth-server/oauth/authorize?response_type=token&token_type=Bearer&client_id=tasks_app&redirect_uri=http://localhost:8888/tasks-service/dashboard/loginOAuth')
```

- b. Añadir el access_token que devuelve el servicio OAuth2 para realizar la autenticación en la aplicación en el archivo *tasks-app/src/main/resources/static/application/frontend/users/OAuthLogin.js*:

El servicio OAuth2 devuelve el token a la dirección <http://localhost:8888/tasks-service/dashboard/loginOAuth> que devolverá extraer el token de la URL y añadirlo al contexto de ejecución de la aplicación SPA. Desde el archivo *OAuthLogin.js* se debe realizar dicha operación. Para ello se toma como referencia el hash de la URL de loginOAuth <http://localhost:8888/tasks-service/dashboard/loginOAuth> y se extrae de este el "access_token" que usará la aplicación SPA para la redirección con el usuario logueado a través del servicio OAuth2.

```
var OAuthLogin = (props) => {  
  if(location.hash) {  
    const hash = window.location.hash.substring(1);  
    const params = {};  
    const handleParams = () => {  
      const paramsHash = new URLSearchParams(hash);  
      params.access_token = paramsHash.get("access_token");  
    }  
    handleParams();  
  
    if(params.access_token) {  
      var jwtToken = jwt.parseJwtToken(params.access_token);  
      jwt.storeJwtToken(params.access_token);  
      props.dispatch({  
        type: 'login',  
        user: jwtToken.user_name,  
        authorities: jwtToken.authorities,  
        token: params.access_token  
      });  
      return (<ReactDOM.Redirect to="/" />);  
    }  
  }  
  
  alerts.error('Access denied!');  
  return (<ReactDOM.Redirect to="/login" />);  
};  
OAuthLogin = ReactRedux.connect()(OAuthLogin);
```

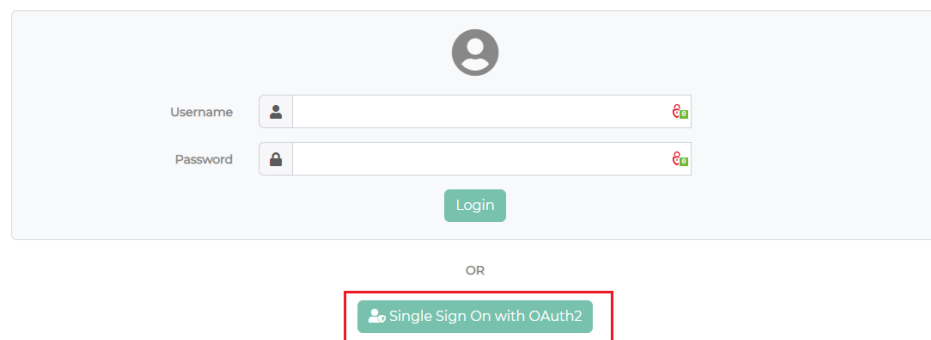
Gracias a esta operación se consigue el “access_token” del servicio OAuth2 para poder redirigir a la aplicación SPA ya logueados. La función `URLSearchParams()` sobre el hash nos permite recuperar los parámetros del hash. En nuestro caso nos interesa el “access_token” para poder obtener la autorización y loguearse en la aplicación.

El uso del servicio OAuth2 permite a la aplicación disponer de un servicio único de login para evitar tener que revelar información de login a servicios de terceros que puedan usar la aplicación.

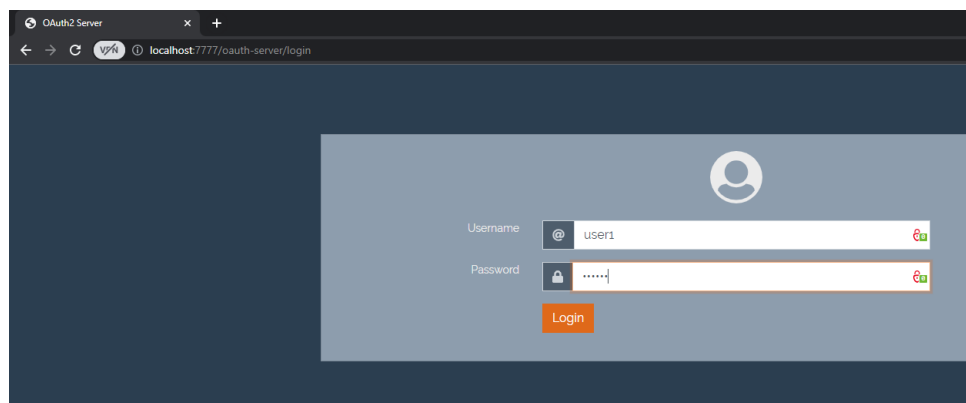
2. Descargar y ejecutar el servidor OAuth2 desde un terminal. En nuestro caso se ha añadido a una carpeta al proyecto denominada OAuth2. Desde esta carpeta ejecutamos el servidor OAuth2:

```
inad_@DESKTOP-41SG6PV MINGW64 /f/MUNICS/SAPP/repo2/OAuth2 (main)
$ java -jar oauth2-server.jar
```

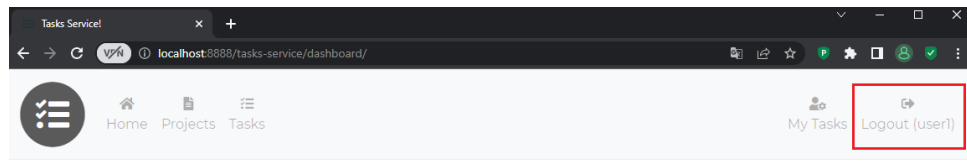
3. Ejecutamos la aplicación SPA.
4. En la pantalla de login aparece una opción para realizar el login con OAuth2:



5. Nos redirecciona a la página de autorización de OAuth2 e insertamos las credenciales:



6. Una vez logueados, si todo funciona correctamente, nos redirige a la página de la aplicación SPA ya logueados:



Welcome to Tasks Service!

Repositorio

El repositorio del proyecto se puede encontrar en:

<https://github.com/aitorcavia/tasks-app.git>