Alumnos:

- Pablo Diz de la Cruz
- Sergio Vila Riveira
- Hugo Freire Blanco

A) Endpoint de Autenticación contra BD Local

Archivo: `com.tasks.rest.UserController`

Se implementó generación dinámica de JWT reemplazando tokens hardcodeados:

```
String token = tokenProvider.generateHs256SignedToken(
    userService.loadUserByUsername(credentials.getUsername())
);
return ResponseEntity.ok(new TokenResponse(token));
```

Flujo:

1. `doLogin()` recibe credenciales y crea `UsernamePasswordAuthenticationToken`
2. `AuthenticationManager` valida contra BD → devuelve `Authentication` con roles
3. Se almacena en `SecurityContext`: `SecurityContextHolder.getContext().setAuthentication(authentication)`
4. `JwtTokenProvider.generateHs256SignedToken()` crea JWT firmado con `jwt.secret`
5. Respuesta HTTP 200 con token en formato `TokenResponse`

Respuesta:

```
{
  "token": "eyJhbGciOiJIUzI1NiJ9..."
}
```

El cliente recibe el JWT emitido y puede almacenarlo y reutilizarlo en peticiones posteriores añadiéndolo en el header `Authorization: Bearer <token>`.

B) Control de Acceso Inicial sin Roles

WebSecurityConfig - Recursos estáticos:

```
.requestMatchers(HttpMethod.GET, "/swagger-ui.html").permitAll()
.requestMatchers(HttpMethod.GET, "/swagger-ui/*").permitAll()
.requestMatchers(HttpMethod.GET, "/v3/api-docs/**").permitAll()
.requestMatchers(HttpMethod.GET, "/webjars/**").permitAll()
.requestMatchers(HttpMethod.GET, "/dashboard/**").permitAll()
.requestMatchers(HttpMethod.GET, "/javascript-libs/**").permitAll()
.requestMatchers(HttpMethod.GET, "/react-libs/*").permitAll()
.requestMatchers(HttpMethod.GET, "/css/*").permitAll()
.requestMatchers(HttpMethod.GET, "/application/**").permitAll()
```

WebSecurityConfig - Endpoints públicos:

```
.requestMatchers(HttpMethod.POST, "/api/login").permitAll()
.requestMatchers(HttpMethod.GET, "/api/projects").permitAll()
.requestMatchers(HttpMethod.GET, "/api/projects/*").permitAll()
.requestMatchers(HttpMethod.GET, "/api/projects/*/tasks").permitAll()
.requestMatchers(HttpMethod.GET, "/api/tasks*").permitAll()
.requestMatchers(HttpMethod.GET, "/api/tasks/*").permitAll()
```

C) Control de Acceso según Roles

WebSecurityConfig - Proyectos:

```
.requestMatchers(HttpMethod.POST, "/api/projects").hasRole("ADMIN")
.requestMatchers(HttpMethod.GET, "/api/projects").permitAll()
.requestMatchers(HttpMethod.GET, "/api/projects/*").permitAll()
.requestMatchers(HttpMethod.PUT, "/api/projects/*").hasRole("ADMIN")
.requestMatchers(HttpMethod.DELETE, "/api/projects/*").hasRole("ADMIN")
```

WebSecurityConfig - Tareas:

```
.requestMatchers(HttpMethod.GET, "/api/tasks*").permitAll()
.requestMatchers(HttpMethod.GET, "/api/tasks/*").permitAll()
.requestMatchers(HttpMethod.POST, "/api/tasks").hasRole("ADMIN")
.requestMatchers(HttpMethod.PUT, "/api/tasks/*").hasRole("ADMIN")
.requestMatchers(HttpMethod.DELETE, "/api/tasks/*").hasRole("ADMIN")
.requestMatchers(HttpMethod.POST, "/api/tasks/*/changeState").hasRole("ADMIN")
.requestMatchers(HttpMethod.POST, "/api/tasks/*/changeResolution").hasRole("USER")
.requestMatchers(HttpMethod.POST, "/api/tasks/*/changeProgress").hasRole("USER")
```

WebSecurityConfig - Proyectos/Tareas:

```
.requestMatchers(HttpMethod.GET, "/api/projects/*/tasks").permitAll()
.requestMatchers(HttpMethod.POST, "/api/projects/*/tasks").hasRole("ADMIN")
.requestMatchers(HttpMethod.DELETE, "/api/projects/*/tasks/*").hasRole("ADMIN")
```

WebSecurityConfig - Usuarios y Comentarios:

```
.requestMatchers(HttpMethod.GET, "/api/users").hasRole("ADMIN")
.requestMatchers(HttpMethod.POST, "/api/comments").authenticated()
.anyRequest().denyAll()
```

ProjectService:

Validar que solo el admin del proyecto puede modificarlo o eliminarlo. Spring Security solo valida el rol ADMIN, pero no que el usuario sea propietario:

```
if(!project.get().getAdmin().getUsername().equals(userName)) {
    throw new PermissionException();
}
```

TasksService - Crear/Actualizar:

Validar que el usuario es admin del proyecto antes de crear/modificar tareas:

```
Optional<User> user = userRepository.findById(username);
if(!user.isPresent()) throw new UsernameNotFoundException(...);
task.setOwner(user.get());
if(!project.get().getAdmin().getUsername().equals(userName)) {
    throw new PermissionException();
}
```

TasksService - changeState:

Validar que el admin del proyecto es quien cambia el estado:

```
if(!optTask.get().getProject().getAdmin().getUsername().equals(userName)) {
    throw new PermissionException();
}
```

TasksService - changeResolution:

Solo el desarrollador asignado a la tarea puede cambiar su resolución:

```
if(task.getState().equals(TaskState.CLOSED)) {
    throw new InalidStateException("Task is closed.");
}
if(!task.getOwner().getUsername().equals(userName)) {
    throw new PermissionException();
}
```

TasksService - changeProgress:

Solo el desarrollador asignado puede cambiar el progreso:

```
if(task.getState().equals(TaskState.CLOSED)) {
    throw new InalidStateException("Task is closed.");
}
if(!task.getOwner().getUsername().equals(userName)) {
    throw new PermissionException();
}
if(!task.getResolution().equals(TaskResolution.IN_PROGRESS)) {
    throw new InalidStateException("Task is not in progress.");
}
```

TasksService - removeById (DELETE):

Solo el admin del proyecto puede eliminar tareas:

```
if(!optTask.get().getProject().getAdmin().getUsername().equals(userName)) {
    throw new PermissionException();
}
Project project = optTask.get().getProject();
project.setTasksCount(project.getTasksCount() - 1);
tasksRepository.delete(optTask.get());
```

TaskController:

Pasar el `Principal` (usuario autenticado) a los métodos de servicio para poder validar permisos a nivel de negocio:

```
public ResponseEntity<?> doChangeTaskResolution(Principal principal,
                                                @PathVariable("id") Long id,
                                                @RequestBody TextNode resolution) {
    Task task = tasksService.changeResolution(principal.getName(), id,

TaskResolution.valueOf(resolution.asText()));
    return ResponseEntity.ok(task);
}

public ResponseEntity<?> doChangeTaskProgress(Principal principal,
                                              @PathVariable("id") Long id,
                                              @RequestBody TextNode progress) {
    Task task = tasksService.changeProgress(principal.getName(), id,
                                            (byte) progress.asInt());
    return ResponseEntity.ok(task);
}
```

Flujos:

- **Crear tarea:** Spring Security (ADMIN) → TasksService (es admin del proyecto) → asigna owner (sin validar rol)
- **changeState:** Spring Security (ADMIN) → TasksService (es admin del proyecto)
- **changeResolution:** Spring Security (USER) → TasksService (es owner + no CLOSED)
- **changeProgress:** Spring Security (USER) → TasksService (es owner + IN_PROGRESS)

D) Autenticación OAuth 2.0 (PCKE)

WebSecurityConfig:

```
.requestMatchers(HttpMethod.GET, "/dashboard/**").permitAll()

// Permitir acceso a la aplicación frontend
.requestMatchers(HttpMethod.GET, "/application/**").permitAll()
```

Login.js - Paso 1 (cliente):

Redirigir al servidor OAuth con parámetros PCKE. El `code_verifier` se genera y almacena para validar la respuesta después:

```
const handleSSO = async () => {
    // Configurar parámetros OAuth
    const redirectUri = 'http://127.0.0.1:8888/tasks-service/dashboard/loginOAuth';
    const codeVerifier = pkceUtils.generateRandomString();
    const codeChallenge = await pkceUtils.getChallenge(codeVerifier);

    // Almacenar el code_verifier en sessionStorage para usarlo después
    sessionStorage.setItem('code_verifier', codeVerifier);

    // Redirigir al servidor de autorización
    window.location.replace(
        'http://127.0.0.1:7777/oauth2/authorize?' +
        'response_type=code&' +
        'client_id=tasks_app&' +
        'redirect_uri=' + encodeURIComponent(redirectUri) + '&' +
        'scope=openid&' +
        'code_challenge=' + codeChallenge + '&' +
        'code_challenge_method=S256'
    );
};
```

OAuthLogin.js - Paso 2 (callback):

Procesar el código de autorización recibido. Intercambiarlo por un token usando el `code_verifier` almacenado (PCKE):

```
const urlParams = new URLSearchParams(location.search.substring(1));
const code = urlParams.get("code");
const codeVerifier = sessionStorage.getItem('code_verifier');

const tokenParams = new Map();
tokenParams.set('grant_type', 'authorization_code');
tokenParams.set('code', code);
tokenParams.set('code_verifier', codeVerifier);
tokenParams.set('redirect_uri', 'http://127.0.0.1:8888/tasks-
service/dashboard/loginOAuth');
tokenParams.set('client_id', 'tasks_app');

oauthService.getToken(tokenParams, response => {
    const accessToken = response.access_token;
    const jwtToken = jwt.parseJwtToken(accessToken);
    jwt.storeJwtToken(accessToken);
    props.dispatch({type: 'login', user: jwtToken.sub, roles: jwtToken.roles, token:
accessToken});
});
```

Flujo del servidor OAuth:

Proceso PCKE para autenticar sin guardar secretos en el cliente:

1. Recibe `GET /oauth2/authorize` con `code_challenge` (PCKE)
2. Autentica usuario y genera `authorization_code`
3. Redirige a: `redirect_uri?code=<authorization_code>`
4. Recibe `POST /oauth2/token` con `code` y `code_verifier`
5. Valida: `code_challenge == base64url(sha256(code_verifier))`
6. Emite `access_token` e `id_token` con claims (sub, roles, exp)

Compatibilidad: Los tokens OAuth tienen la misma estructura que los del servidor local → funcionan con `JwtAuthorizationFilter` sin cambios