

Proyecto Final Tokio School

Introducción

Este documento es la explicación del Proyecto Final del curso de Spring Boot de Tokio School.

La plataforma esta disponible [aquí](#) pero como esta alojado en un instance de Heroku, puede tardar unos segundos en cargar.

Todo el código del proyecto completo se encuentra en este [GitHub repository](#).

Los commits para cada de los tres pasos del proyecto se encuentran aquí:

- [Página Web Thymeleaf con Bootstrap](#)
- [RESTful API con JWT authentication y OpenAPI 2.0 docs](#)
- [Spring Batch Job](#)

En vez de incluir grabaciones de pantalla, proveeré enlaces al código en el repository de GitHub. También Proporcionaré en el propio documento fragmentos de código que requieren más explicación.

Estructura del Programa

La estructura final del programa es:

```
.
├── HELP.md
├── LICENSE
├── logs
│   ├── film_fanatic@s-2022-03-07.0.log
│   ├── film_fanatic@s-2022-03-08.0.log
│   ├── film_fanatic@s-2022-03-10.0.log
│   ├── film_fanatic@s-2022-03-11.0.log
│   ├── film_fanatic@s-2022-03-29.0.log
│   ├── film_fanatic@s-2022-03-30.0.log
│   ├── film_fanatic@s-2022-04-03.0.log
│   ├── film_fanatic@s-2022-04-03.1.log
│   ├── film_fanatic@s-2022-04-04.0.log
│   └── film_fanatic@s.log
├── mvnw
├── mvnw.cmd
├── pom.xml
├── README.md
├── readme-pics
│   ├── add-film.png
│   ├── add-person.png
│   ├── add-score.png
│   ├── admin-create-user.png
│   ├── admin-user-created.png
│   ├── datepicker.png
│   └── delete-user.png
```

```
├── edit-user.png
├── film.png
├── film-review.png
├── Header-login.png
├── header-search-bar.png
├── Home Page.png
├── login.png
├── logout-message.png
├── loing-failure.png
├── Newly Added Review.png
├── password-modal.png
├── person-created-message.png
├── profile.png
├── register.png
├── search-films.png
├── src
│   ├── main
│   │   ├── java
│   │   │   ├── io
│   │   │   │   ├── chilborne
│   │   │   │   │   ├── filmfanatic
│   │   │   │   │   │   ├── batch
│   │   │   │   │   │   │   ├── FilmItemProcessor.java
│   │   │   │   │   │   │   ├── FilmLineAggregator.java
│   │   │   │   │   │   │   ├── FilmMapper.java
│   │   │   │   │   │   │   ├── MigrateFilmStartListener.java
│   │   │   │   │   │   │   ├── MigrateFilmWriteListener.java
│   │   │   │   │   │   │   └── MigrateJobConfig.java
│   │   │   │   │   │   ├── config
│   │   │   │   │   │   │   ├── ApplicationConfig.java
│   │   │   │   │   │   │   ├── FileServiceConfig.java
│   │   │   │   │   │   │   ├── OpenApiConfig.java
│   │   │   │   │   │   │   └── ValidationMessageConfig.java
│   │   │   │   │   │   ├── controller
│   │   │   │   │   │   │   ├── api
│   │   │   │   │   │   │   │   ├── ApiAuthenticationController.java
│   │   │   │   │   │   │   │   ├── ApiExceptionHandlerAdvice.java
│   │   │   │   │   │   │   │   ├── ApiReviewController.java
│   │   │   │   │   │   │   │   └── ErrorResponse.java
│   │   │   │   │   │   │   └── thymeleaf
│   │   │   │   │   │   │       ├── FilmController.java
│   │   │   │   │   │   │       ├── PersonController.java
│   │   │   │   │   │   │       ├── UserController.java
│   │   │   │   │   │   │       ├── WebController.java
│   │   │   │   │   │   │       └── WebsiteExceptionHandlerAdvice.java
│   │   │   │   │   │   └── domain
│   │   │   │   │   │       ├── dto
│   │   │   │   │   │       │   ├── ChangePasswordDTO.java
│   │   │   │   │   │       │   ├── CreateUserDTO.java
│   │   │   │   │   │       │   ├── EditUserDTO.java
│   │   │   │   │   │       │   ├── FilmSearchCriteriaEnum.java
│   │   │   │   │   │       │   ├── PasswordDTO.java
│   │   │   │   │   │       │   └── ReviewDTO.java
│   │   │   │   │   │       └── Film.java
```

```

├── Person.java
├── PersonTypeEnum.java
├── Review.java
├── Role.java
├── Score.java
├── User.java
├── exception
│   ├── FilmNotFoundException.java
│   ├── ImageUploadException.java
│   ├── ReviewAlreadyExistsException.java
│   ├── UnauthorizedException.java
│   ├── UsernameAlreadyExistsException.java
│   ├── UserNotFoundException.java
│   └── UserRemovalException.java
├── FilmFanaticApplication.java
├── repository
│   ├── FilmRepository.java
│   ├── PersonRepository.java
│   ├── ReviewRepository.java
│   ├── RoleRepository.java
│   ├── ScoreRepository.java
│   └── UserRepository.java
├── security
│   ├── jwt
│   │   ├── JwtAuthenticationEntryPoint.java
│   │   ├── JwtRequestFilter.java
│   │   ├── JwtRequest.java
│   │   ├── JwtResponse.java
│   │   └── JwtTokenUtil.java
│   ├── SuccessfulAuthenticationEventListener.java
│   ├── UserDetailsServiceImpl.java
│   └── WebSecurityConfig.java
├── service
│   ├── FileService.java
│   ├── filmsearch
│   │   ├── FilmSearch.java
│   │   ├── implementation
│   │   │   └── FilmSearchImpl.java
│   │   └── strategy
│   │       ├── FilmSearchStrategy.java
│   │       └── implementation
│   │           ├── FilmActorSearch.java
│   │           ├── FilmAverageScoreSearch.java
│   │           ├── FilmCinematographerSearch.java
│   │           ├── FilmComposerSearch.java
│   │           ├── FilmDirectorSearch.java
│   │           ├── FilmMaxDurationSearch.java
│   │           ├── FilmScreenwriterSearch.java
│   │           ├── FilmTitleSearch.java
│   │           └── FilmYearSearch.java
│   ├── FilmService.java
│   ├── implementation
│   │   ├── FileServiceImpl.java
│   │   └── FilmPosterImageFileService.java

```

```
├── FilmServiceImpl.java
├── PersonServiceImpl.java
├── ReviewServiceImpl.java
├── RoleServiceImpl.java
├── UserServiceImpl.java
├── PersonService.java
├── ReviewService.java
├── RoleService.java
├── UserService.java
├── util
│   ├── Constants.java
│   ├── FileUtils.java
│   └── StringUtil.java
└── resources
    ├── application-dev.properties
    ├── application-prod.properties
    ├── application.properties
    ├── application-test.properties
    ├── data.sql
    ├── films.csv
    ├── logback-spring.xml
    ├── static
    │   ├── images
    │   │   ├── cinema.png
    │   │   ├── films
    │   │   │   ├── kingdom-of-heaven.jpg
    │   │   │   └── the-fellowship-of-the-ring.jpg
    │   │   └── users
    │   │       ├── tokioschool.jpeg
    │   │       └── xsRV^Y3+env9+!3=g77H.jpg
    │   ├── js
    │   │   └── iziToast.min.js
    │   └── styles
    │       └── css
    │           ├── borders.css
    │           ├── custom-colours.css
    │           ├── iziToast.min.css
    │           ├── layout.css
    │           └── login.css
    ├── templates
    │   ├── delete-user.html
    │   ├── edit-profile.html
    │   ├── error.html
    │   ├── film.html
    │   ├── fragments
    │   │   ├── footer.html
    │   │   ├── header.html
    │   │   ├── head.html
    │   │   ├── passwordModal.html
    │   │   ├── personCreatedModal.html
    │   │   └── userCreatedModal.html
    │   ├── index.html
    │   ├── login.html
    │   └── new-film.html
```

```
├── new-person.html
├── profile.html
├── registration.html
├── searched-film.html
├── search-film.html
├── template.html
├── validation-messages.properties
├── test
│   ├── java
│   │   ├── io
│   │   │   ├── chilborne
│   │   │   │   ├── filmfanatic
│   │   │   │   │   ├── batch
│   │   │   │   │   │   ├── MigrateFilmBatchJobTest.java
│   │   │   │   │   ├── controller
│   │   │   │   │   │   ├── api
│   │   │   │   │   │   │   ├── ApiIntegrationTest.java
│   │   │   │   │   ├── FilmFanaticApplicationTests.java
│   │   │   │   │   ├── repository
│   │   │   │   │   │   ├── FilmRepositoryTest.java
│   │   │   │   │   │   ├── PersonRepositoryTest.java
│   │   │   │   │   │   ├── ScoreRepositoryTest.java
│   │   │   │   │   └── TestData.java
│   │   └── resources
│   │       └── test.sql
└── system.properties
```

Sumario

- [Proyecto Final Tokio School](#)
 - [Introducción](#)
 - [Estructura del Programa](#)
 - [Sumario](#)
 - [Usuario](#)
 - [Objeto de Dominio](#)
 - [Creación de Usuario](#)
 - [CreateUserDTO](#)
 - [registration.html](#)
 - [UserController](#)
 - [UserService](#)
 - [addUser](#)
 - [saveUserImageString username, MultipartFile imageFile](#)
 - [FileService](#)
 - [saveFileMultipartFile file, String fileName](#)
 - [Login](#)
 - [WebSecurityConfig](#)
 - [UserDetailsServiceImpl](#)
 - [Login/Logout Success](#)
 - [SuccessfulAuthenticationEventListener](#)

- Login Failure
- User Profile
 - userCreatedModal.html
- Editar Usuario
 - Cambiar imagen de usuario
 - Cambiar contraseña
 - changePasswordString username, String oldPassword, String newPassword
 - Cambiar datos personales de usuario
 - updateUserString username, User user
 - updateSecurityContextString username
- Eliminar Usuario
 - deleteUserPrincipal principal
- Personas
 - PersonController.addPerson
 - PersonService.addPersonPerson person
- Películas
 - objeto de dominio
 - crear película
 - addFilmFilm film
 - savePosterFilm film , MultipartImage posterImage
 - buscar películas
 - FilmSearchCriteriaEnum
 - FilmSearchStrategy
 - FilmSearchImpl
 - Ruta de Solicitud de Búsqueda
 - Info de Película
 - Dar puntuación a una pelí
 - FilmService.addScoreString filmUri, Score score
 - FilmService.getFilmByUriString uri
- Review
 - Solicitar reseñas del usuario
 - Creando una reseña nueva
 - ApiIntegrationTest
 - ReviewService
 - findByUsernameString username
 - addReviewReview review
- Seguridad
 - SuccessfulAuthenticationEventListener
 - Seguridad JWT
 - JwtRequest y JwtResponse
 - JwtRequestFilter
 - Proceso de autenticación
 - Set-Cookie: JSESSIONID=...
- Proceso Batch
 - MigrateJobConfig
 - reader

- [FilmMapper](#)
- [FilmItemProcessor](#)
- [writer](#)
- [FilmLineAggregator](#)
- [step](#)
- [MigrateFilmWriteListener](#)
- [migrateFilmJob](#)
- [MigrateFilmListener](#)
- [Excepciones, Depuración y logging](#)
 - [Gestión de excepciones](#)
 - [WebsiteExceptionHandlerAdvice](#)
 - [ApiExceptionHandlerAdvice](#)
 - [Depuración y logging](#)
- [Aprendizaje y Mejoras](#)

Usuario

El Usuario es la fundación de nuestra app. Casio todos los datos, de [Film](#), [Score](#) y [Review](#) pertenecen a un [User](#). Son los usuarios que añadan los datos del app y, si un [User](#) se elimina del app - dichos datos también serán eliminados.

Objeto de Dominio

[User.java](#)

Como se puede ver, decidí implementar el [interface](#) de [UserDetails](#) con el objeto de dominio [User](#).

Así, se puede cast a [User](#) el objeto [Principal](#) del [Authentication](#) que contiene los detalles del session de usuario en el [SecurityContext](#). Por ejemplo en el [FilmController](#) método [FilmInfo](#):

```
@RequestMapping(path = "films/{filmUrl}/score", method =
RequestMethod.POST)
public String filmInfo(@PathVariable("filmUrl") String filmUrl,
                        @ModelAttribute("newScore") @Valid Score score,
                        BindingResult result,
                        Authentication auth) {
    if (result.hasErrors()) {
        return "film";
    }
    score.setUser((User) auth.getPrincipal());
    filmService.addScore(filmUrl, score);

    return "redirect: /films/" + filmUrl;
}
```

Como el único usuario que puede añadir un nuevo [Score](#) es el que está ya autenticado, no sea necesario hacer otra llamada a la base de datos para añadir lo al nuevo [Score](#), sino solo añadir el [User](#) del

Authentication. Aun que se puede eliminar llamadas innecesarias al base de datos también con el caching de resultados y los anotaciones de Spring `@EnableCaching`, `@Cacheable` y `@ClearCache`.

Para implementar el `UserDetails` interface era necesario `@Override` varios métodos:

```
@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@Override
public boolean isEnabled() {
    return active;
}
```

Para el método de `getAuthorities()` usé una propiedad `Collection<GrantedAuthority> authorities`; con la anotación `@Transient` así que no fuera guardado en el base de datos, que implicaría otro tabla en dicho base. De esta manera era necesario implementar el método así:

```
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    if (authorities == null) {
        Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
        roles.forEach(role -> grantedAuthorities.add(new
SimpleGrantedAuthority(role.getName())));
        authorities = new ArrayList<>(grantedAuthorities);
    }
    return authorities;
}
```

La función es similar de la initialization perezosa de un objeto de tipo singleton. Si la propiedad esta `null`, se crea una colección de autoridades hechas de los `Role` del usuario y se asigna a la propiedad, si no esta null se devuelve. De esta manera solo se necesita cargar las autoridades una vez durante la vida del objeto `User`. Si en algún momento se cambiara los `Role` del usuario, solo sería necesario llamar este mismo método para actualizar las autoridades del usuario.

También se tiene que entender que todos los objetos de dominio que tiene colecciones como miembros, el `User` incluido, esta creados con estas colecciones vacías y métodos de utilidad para añadir y eliminar

objetos de ellas. Por ejemplo, en el **User**:

```
@OneToMany(mappedBy = "user", orphanRemoval = true)
private final Set<Film> films = new HashSet<>();

public void addFilm(Film film) {
    films.add(film);
}

public void removeFilm(Film film) {
    films.remove(film);
}
```

Se hace así porque el framework ORM (Object Relational Mapping) de **Hibernate** se mantiene sesiones de los objetos, y las colecciones miembros están incluidas en estas sesiones. Si se asigna una nueva colección a la propiedad del objeto **Hibernate** no le va a persistir en el base de datos porque no forma parte de dicha sesión.

En el código arriba se ve que los **Film** del **User** tienen la anotación de **@OneToMany** con la opción de **orphanRemoval = true**. Así cuando se elimina el usuario, todas las películas que este ha añadido a la plataforma también serán eliminadas. Es igual para los **Review** y **Score**.

Como el servidor usa el **username** y no el número de identificación para buscar un **User**, añadí otro index al objeto de dominio:

```
@Table(indexes = @Index(name = "username_index", columnList = "username"))
```

Así cuando se busca un **User** con su **username** es más rápido.

Creación de Usuario

- [index.html](#)
- [head.html](#)
- [header.html](#)
- [footer.html](#)

Al llegar a la página inicial de la plataforma de FilmFanatic@s encontramos en el header de la página un botón de 'Register' para crear un usuario nuevo.



En el fichero **header.html** podemos ver que si no hay un usuario autenticado, se muestran los botones para iniciar sesión ('Login') o registrarse ('Register').

```
<div class="d-flex align-items-center" sec:authorize="!isAuthenticated()">
  <a th:href="@{/login}" class="link-primary">Login</a>
```

```
<small class="mx-1" style="color:rgba(209, 204, 192,1.0)" >or</small>
<a class="btn btn-primary btn-sm" th:href="@{/register}">Register</a>
</div>
```

Al pinchar el botón para registrarse, se llama el método con `@RequestMapping` de GET en el controlador `UserController`. Este método crea un nuevo objeto de tipo `CreateUserDTO` y lo adjunta con el model del `registration.html`. También carga el model con los Role disponible por si caso es un Admin que quiere crear un `User` nuevo.

```
@RequestMapping(path = "/register", method = RequestMethod.GET)
public String registerUser(Model model) {
    model.addAttribute("createUserDTO", new CreateUserDTO());
    model.addAttribute("roles", roleService.getAllRoles());
    return "registration";
}
```

CreateUserDTO

- [CreateUserDTO.java](#)
- [PasswordDTO.java](#)

Decidí usar objetos de DTO para la creación y actualización de los usuarios para poder verificar las contraseñas. También solo es necesario incluir los campos necesarios para crear o actualizar un `User` y no todos.

El `CreateUserDTO` extiende el clase abstracta `PasswordDTO`:

```
public abstract class PasswordDTO {

    @NotBlank(message = "{field.mandatory}")
    @Size(min= 8, message = "{field.password.length}")
    protected String password;
    @NotNull(message = "{field.password.match}")
    protected String confirmPassword;

    public void setPassword(String password) {
        this.password = password;
        confirmPassword();
    }

    public void setConfirmPassword(String confirmPassword) {
        this.confirmPassword = confirmPassword;
        confirmPassword();
    }

    private void confirmPassword() {
        if (this.password == null || this.confirmPassword == null) {
            return;
        }
    }
}
```

```
    }  
    else if (!this.password.equals(this.confirmPassword)) {  
        this.confirmPassword = null;  
    }  
}  
}
```

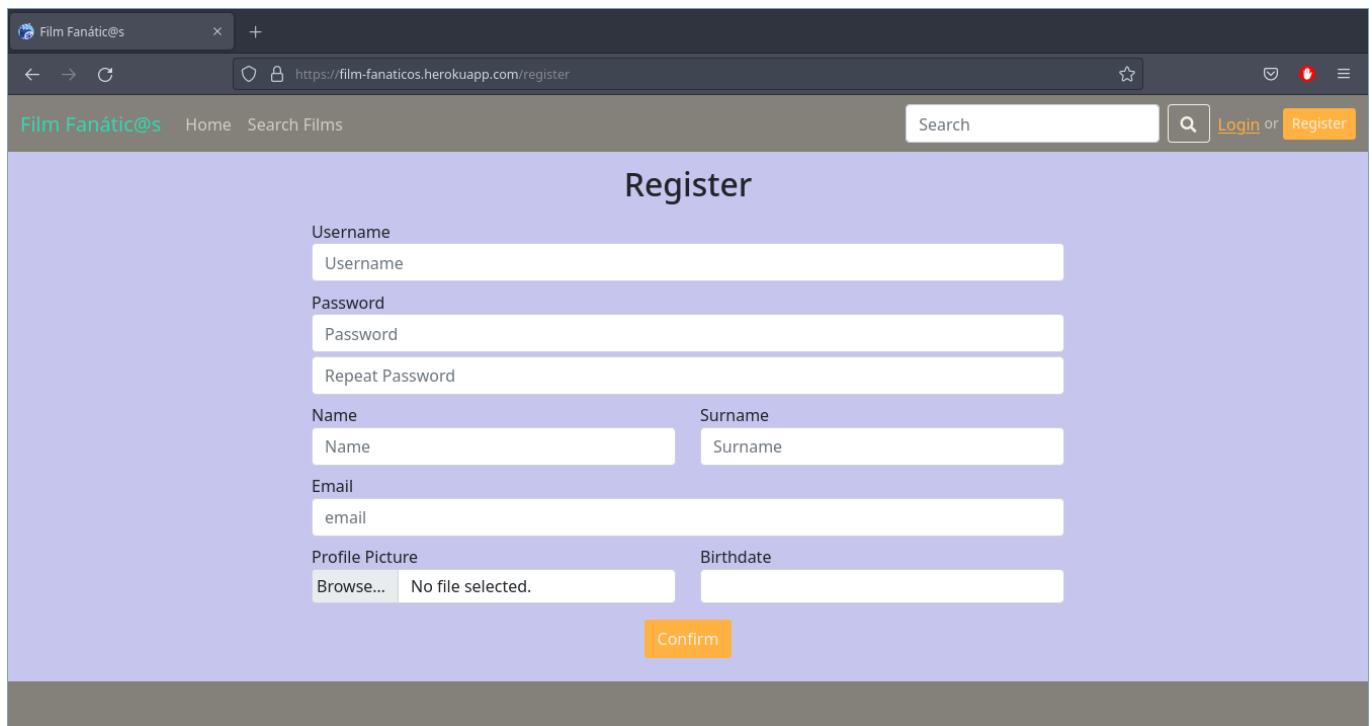
El `PasswordDTO` funciona para verificar que las dos contraseñas que escribe el usuario nuevo son iguales. Si no son, se pone el campo de `confirmPassword` como `null` y eso activa la sistema de validación que esta vinculando al campo con la anotación de `@NotNull(message = "{field.password.match}")`. Todos los mensajes de validación están inducidos en el fichero `validation-messages.properties` ([GitHub](#)) y configurado por el objeto de `@Configuration` de `ValidationMessageConfig` ([GitHub](#)).

registration.html

- [registration.html.java](#)

Cuando se ve cualquier formulario en la plataforma se puede notar que antes hay un par de tags `<form>` `</fom>` vaciás. Es así porque encontré un bug que Thymeleaf no haría el formulario sin esto antes. Ya he creado un 'Issue' en el GitHub de Thymeleaf [aquí](#).

La paginá ya servida lleva el formulario para crear un `User` nuevo.

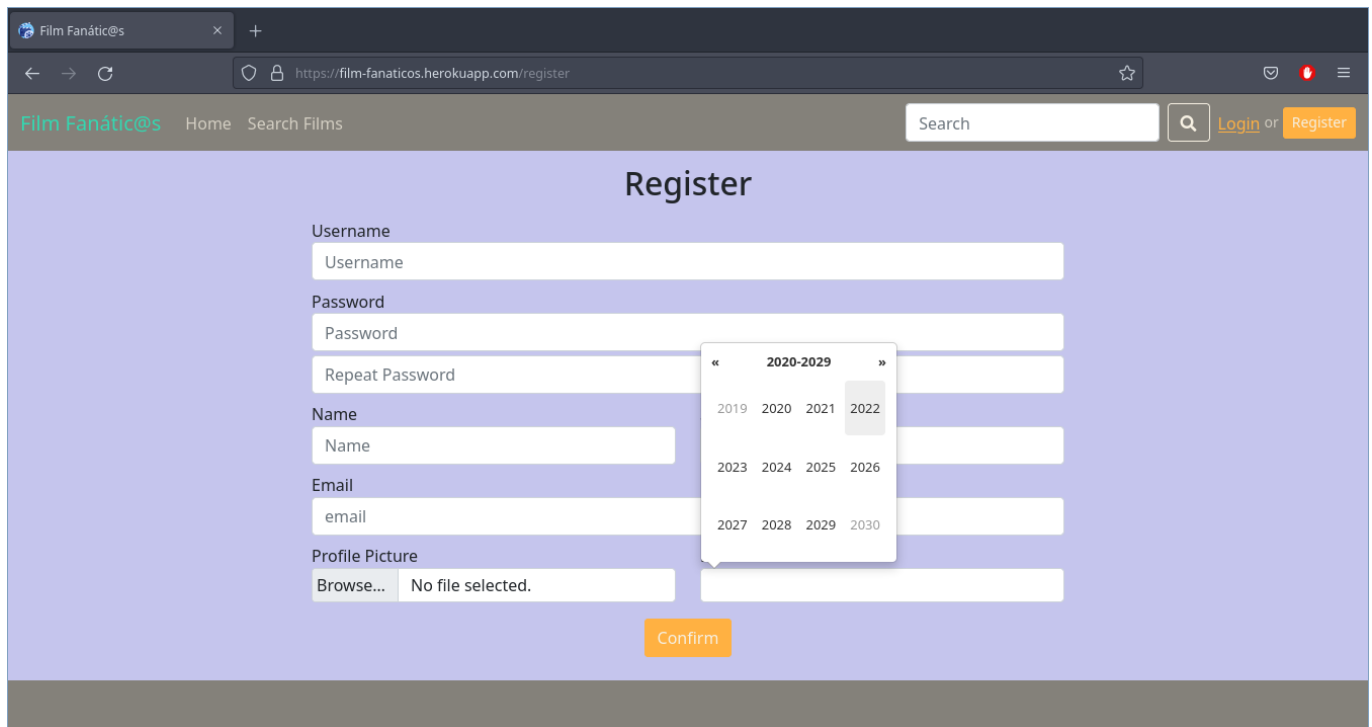


```
<form th:method="POST" enctype="multipart/form-data"  
th:object="${createUserDTO}" th:action="@{/register}">
```

Este formulario carga el objeto de `CreateUserDTO` con los datos ingresados por el usuario, y también puede enviar una imagen del usuario si se quiere.

```
<div class="col">
  <label for="userImage" class="labels">Profile Picture</label>
  <input class="form-control" type="file" name="userImage" id="userImage"
style="padding: 0.25rem;">
</div>
```

Usé el 'widget' de [bootstrap-datepicker](#) para seleccionar la fecha de nacimiento del usuario.



```
<div class="col">
  <label for="birthDate" class="labels">Birthdate</label>
  <input type="text" data-provide="datepicker" data-date-
format="dd/mm/yyyy" class="form-control datepicker" id="birthDate"
autocomplete="off" th:field="*{birthDate}">
</div>
```

Esto requiere que se carga el `.css` y `.js` en el fichero `head.html` que esta incluida en todos los ficheros de `.html` como fragment.

```
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/bootstrap-
datepicker/1.9.0/css/bootstrap-datepicker.min.css" integrity="sha512-
mSYUmp1HYZDFaVKK//63EcZq4iFWFjxSL+Z3T/aCt4I09Cejm03q3NKKYN6pFQzY0SB0r8h+eCI
AZHPXcpZaNw==" crossorigin="anonymous" referrerpolicy="no-referrer" />
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/bootstrap-
datepicker/1.9.0/css/bootstrap-datepicker.standalone.min.css"
integrity="sha512-
```

```
TQQ3J4WkE/rwojNFo60Jdyu6G8Xe9z8rMr1F9y7xpFbQfW5g8aSWcygCQ4vqRiJqFsDsE1T6MoA
OMJkFXlrI9A==" crossorigin="anonymous" referrerpolicy="no-referrer" />
```

Todos los campos del **CreateUserDTO** que tienen anotaciones de validación también llevan un aviso de error en la página de registro si por caso hayan errores en los datos ingresados. Estos avisos solo se muestran si cuando el formulario esta enviado al servidor hay errores. Por ejemplo:

```
<input type="password" placeholder="Repeat Password" th:field="*
{confirmPassword}" class="form-control" style="margin-top: 0.2rem;">
<div class="alert alert-warning mb-0 mt-1 py-1"
th:if="${#fields.hasErrors('confirmPassword')}" th:errors="*
{confirmPassword}"></div>
```

Si es un Admin que quiere crear un **User** nuevo tambien se muestra la opción de decidir los **Role** del usuario nuevo:

```
<div class="col" sec:authorize="hasAuthority('ADMIN')">
  <label for="role">Role</label>
  <select id="role" class="form-select" th:field="*{roles}">
    <option th:each="role : ${roles}" th:value="${role.id}"
th:text="${role.name}"></option>
  </select>
</div>
```

UserController

- [UserController.java](#)

Al enviar los datos del usuario nuevo al servidor se llama el método POST `registerNewUser()` del `UserController`:

```
@RequestMapping(path = "/register", method = RequestMethod.POST)
public String registerNewUser(@RequestParam("userImage")MultipartFile
imageFile,
                                @ModelAttribute @Valid CreateUserDTO
createUserDTO,
                                BindingResult result,
                                Authentication auth)
{
    if (result.hasErrors()) {
        return "registration";
    }
    User newUser = userService.add(createUserDTO.buildUser());
    if (!imageFile.isEmpty()) {
        userService.saveUserImage(newUser.getUsername(), imageFile);
    }
    if (auth != null &&
        auth.getAuthorities().contains(new SimpleGrantedAuthority("ADMIN")))
    {
        return "redirect:/admin/user/" + newUser.getUsername() + "?
user=created";
    }
    return "login";
}
```

Este método tiene 6 pasos:

1. Primero se devuelve la página si el `BindingResult` tiene errores para que los avisos sean mostrados al usuario.
2. Se crea un objeto de `User` con el método `buildUser()` del `CreateUserDTO` que simplemente mapea los campos del DTO al objeto de dominio.
3. Se llama el método `add(User user)` de `UserService` para guardar el `User` creado en el base de datos.
4. Si el formulario lleva un imagen, se llama el método `saveUserImage(String username, MultipartFile imageFile)` de `UserService` para guardarlo.
5. Si el autenticado usuario es de tipo Admin, se envía a la página de información sobre el usuario recién creado.
6. Si no es autenticado ni Admin, se devuelve la página de iniciar sesión para que el usuario nuevo haga el login.

UserService

- [UserServiceImpl.java](#)

add(User)

- [UserRepository.java](#)

```

@Override
@Transactional
public User add(User user) {
    if (userRepo.findByUsername(user.getUsername()).isPresent()) {
        throw new UsernameAlreadyExistsException("Username not available");
    }
    BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
    user.setPassword(encoder.encode(user.getPassword()));
    user.setCreationDate(LocalDate.now());
    user.setActive(true);
    user.setImage(DEFAULT_PROFILE_IMAGE);

    if (user.getRoles().isEmpty()) {
        user.addRole(roleRepo.findByNameIgnoreCase("USER"));
    }

    return userRepo.save(user);
}

```

El método `add(User user)` tiene 8 pasos:

1. Llama el método `findByUsername(String username)` del repository `UserRepository` para busca cualquier `User` con el `username` del objeto nuevo. Si uno ya existe, se lanza un error de tipo `UsernameAlreadyExistsException` ([GitHub](#)).
2. Se crea un nuevo instancia del `BCryptPasswordEncoder` para codificar la contraseña del usuario nuevo antes de guardarlo en el base de datos.
3. Se codificar la contraseña del `User` y se guarda en dicho objeto.
4. Se establece la fecha de creación del `User`.
5. Se activa el `User`.
6. Se asigna al `User` nuevo el imagen de usuario por defecto, si hay un imagen incluido con el formulario de registro, se actualiza más adelante.
7. Si el `User` no tiene `Role` es porque no se ha creado un 'Admin', así que hay que asignarlo el `Role` de 'User'.
8. Se guarda el `User` nuevo en el base de datos con una llamada al método `save(User user)` del `UserRepository` y devuelva el `User` ya guardado.

`saveUserImage(String username, MultipartFile imageFile)`

- [StringUtils.java](#)

```

@Override
@Transactional
public void saveUserImage(String username, MultipartFile imageFile) {
    User toUpdate =
    userRepo.findByUsername(username).orElseThrow(UserNotFoundException::new);
    String imageFileName =
    StringUtil.getUserImageFileName(toUpdate.getId(),
    imageFile.getContentType());
}

```

```
if (!toUpdate.getImage().equals(imageFileName)) {
    toUpdate.setImage(imageFileName);
    userRepo.save(toUpdate);
}
fileService.saveFile(imageFile, imageFileName);
}
```

Este método es usado en el caso de la creación de un usuario nuevo, o el caso del cambio del imagen de un usuario existente.

Tiene 4 pasos:

1. Busca el **User** en el base de datos y si no existe lanza una excepción.
2. Llama el método `getUserImageFileName(long userId, String contentType)` para tener el nombre con que se guarda el imagen.

```
public static String getUserImageFileName(long userId, String contentType)
{
    return userId + getFileExtension(contentType);
}

private static String getFileExtension(String contentType) {
    return "." + contentType.split("/")[1];
}
```

El método usa el numero id del usuario por si en caso de que se cambie el **username**, no es necesario cambiar el capo de **image** en el objeto **User**.

3. Si el campo **image** del **User** es diferente que el **imageFileName** ya devuelto, por ejemplo si es el nombre del imagen que se usa por defecto, se actualiza el campo y guarda el **User** actualizado en el base de datos.
4. Llama el método `saveFile(MultipartFile file, String fileName)` del **FileService** que el **UserServiceImpl** tiene como dependencia.

FileService

- [FileServiceImpl.java](#)
- [FileServiceConfig.java](#)

Se inyecta al **UserServiceImpl** el **FileService** asi.

```
@Service
public class UserServiceImpl implements UserService {

    private final UserRepository userRepo;
    private final RoleRepository roleRepo;
```



```
private final FileService fileService;

private final Logger logger =
LoggerFactory.getLogger(UserServiceImpl.class);

public UserServiceImpl(UserRepository userRepo,
                      RoleRepository roleRepo,
                      @Qualifier("user-image-file-service") FileService
fileService) {
    this.userRepo = userRepo;
    this.roleRepo = roleRepo;
    this.fileService = fileService;
}
```

El **Bean** del **FileService** esta cualificada porque existen dos en el programa. Uno para gestionar los archivos de imágenes del usuarios, y otro para los de películas.

Como los dos **FileService** funcionan igual, solo se guardan los archivo en destinos diferentes, es más fácil usar solo una clase que se carga al momento de su creación con el destino diferente. Esto hacemos en la clase de configuración **FileServiceConfig**.

```
@Configuration
public class FileServiceConfig {

    @Value("${images.upload.directory.user}")
    private String userImageUploadDirectory;
    @Value("${images.upload.directory.film}")
    private String filmPosterUploadDirectory;

    @Bean("user-image-file-service")
    public FileService userImageFileService() {
        return new FileServiceImpl(userImageUploadDirectory);
    }

    @Bean("film-poster-file-service")
    public FileService filmPosterFileService() {
        return new FileServiceImpl(filmPosterUploadDirectory);
    }
}
```

Y el constructor de la clase **FileServiceImpl** asi.

```
@Slf4j
public class FileServiceImpl implements FileService {

    private String uploadDirectory;

    public FileServiceImpl(String uploadDirectory) {
```

```
this.uploadDirectory = uploadDirectory;
}
```

Los destinos están contenidos en la ficha de propiedades `application.properties`.

```
images.upload.directory.user=static/images/users
images.upload.directory.film=static/images/films
```

saveFile(MultipartFile file, String fileName)

- [FileUtils.java](#)

```
@Override
public void saveFile(MultipartFile file, String fileName) throws
ImageUploadException {
    log.info("Saving file {} as {}", file, fileName);
    try (InputStream fileStream = file.getInputStream()) {
        Path imagePath = FileUtils.getResourcePath(uploadDirectory, fileName);
        if (Files.exists(imagePath)) {
            Files.copy(fileStream, imagePath,
StandardCopyOption.REPLACE_EXISTING);
        }
        else {
            Files.write(imagePath, fileStream.readAllBytes(),
StandardOpenOption.CREATE_NEW, StandardOpenOption.WRITE);
        }
    } catch (IOException e) {
        log.error("Error saving file {}", fileName, e);
        throw new ImageUploadException("Image Upload Failed", e);
    }
}
```

El método para guardar un archivo tiene 5 pasos:

1. Escribe al log para informar del proceso de guardar el archivo.
2. Dentro de un bucle de 'try-else' se saca un `InputStream` del archivo. Así que si algo falla, el recurso se cierra automáticamente.
3. Llama el método estático `getResourcePath(String uploadDirectory, String fileName)` de la clase `FileUtils` para obtener el `Path` donde se va guardar el archivo:

```
private static Path getRootResourcePath() throws IOException {
    return Paths.get(resourceLoader.getResource("classpath:").getURI());
}

public static Path getResourcePath(String directory, String fileName)
throws IOException {
```

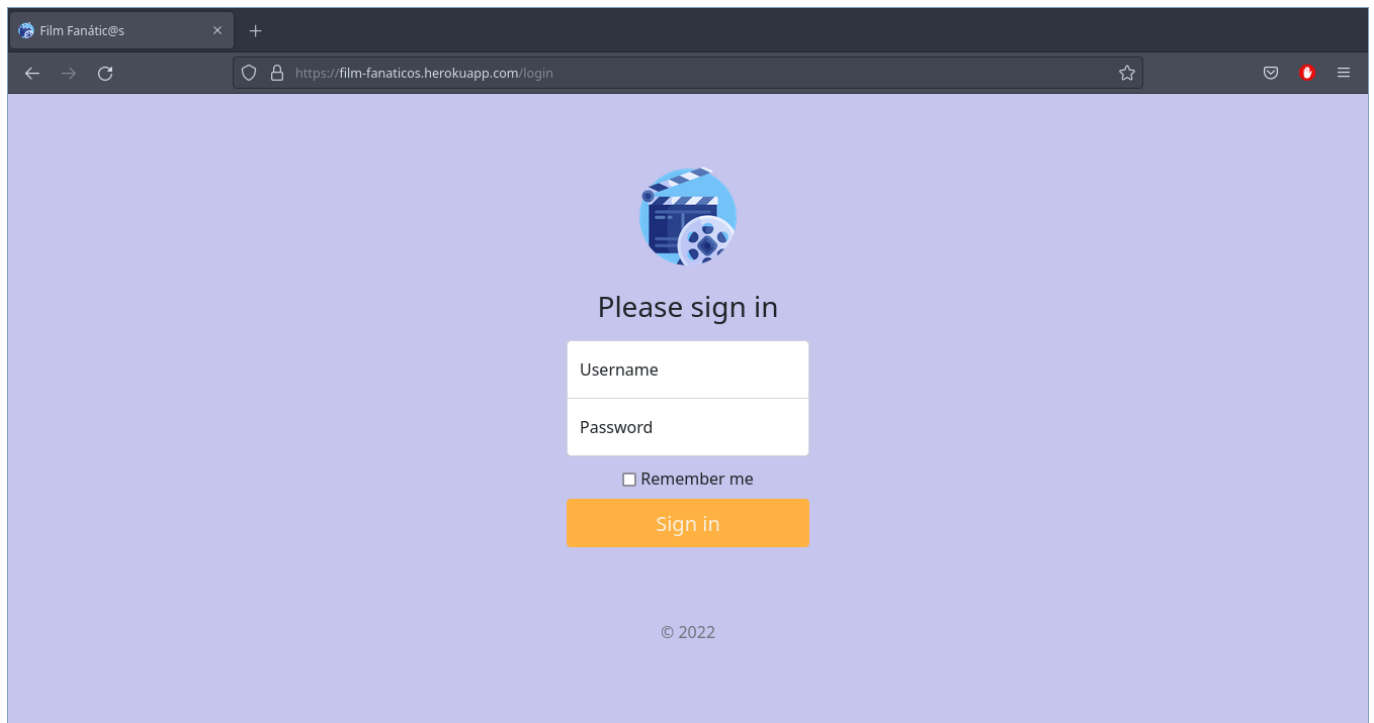
```
Path path = Paths.get(getRootResourcePath() + File.separator + directory
+ File.separator + fileName);
return path;
}
```

4. Si ya hay un archivo guardado a este **Path** se sobrescribe con el archivo nuevo.
5. Si no existe un archivo a este **Path** todavía, se lo escribe nuevo.

Login

- [login.html](#)

Al pinchar en el link de 'Login' para iniciar sesión el método GET **login(Model model)** del **UserController** devuelve la pagina de **login.html** donde se encuentra el formulario para indicar el username y contraseña del **User**.



WebSecurityConfig

- [WebSecurityConfig.java](#)

La clase **WebSecurityConfig** que extiende el **WebSecurityConfigurerAdapter** proporcionado por Spring Security es donde se configura la método de iniciar sesión.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .csrf()
        .disable()
        .authorizeRequests()
        .antMatchers("/").permitAll()
        .antMatchers("/register").permitAll()
}
```

```

        .antMatchers("/login").permitAll()
        .antMatchers("/film/**").permitAll()
        .antMatchers("/films/**").permitAll()
        .antMatchers("/search").permitAll()
        .antMatchers("/h2-console/**").permitAll()
        .antMatchers("/v3/api-docs/**" , "/swagger-ui/**", "swagger-
ui.html").permitAll()
        .antMatchers("/admin/**").hasAnyAuthority(ADMIN_ROLE)
        .anyRequest().authenticated()
    .and()
        .exceptionHandling()
            .authenticationEntryPoint(jwtAuthenticationEntryPoint)
    .and()
        .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
    .and()
        .formLogin()
            .loginPage(LOGIN_URL)
            .loginProcessingUrl(LOGIN_URL)
            .successForwardUrl(LOGIN_SUCCESS_URL)
            .failureUrl(LOGIN_FAILURE)
    .and()
        .logout()
            .logoutRequestMatcher(new AntPathRequestMatcher(LOGOUT_URL))
            .logoutSuccessUrl(LOGOUT_SUCCESS_URL)
    .and()
        .addFilterBefore(jwtRequestFilter,
UsernamePasswordAuthenticationFilter.class)
        // in order to allow access to h2-console for testing
        // TODO remove before release
        .headers().frameOptions().disable();
    }

```

Ignorando las partes del método que pertenecen al tema de RESTful API y autenticación JWT, se puede ver que mientras varios paths están abierto a los usuarios no autenticados, se configura la autenticación del usuario con formulario usando varios variables constantes que se definen en al clase de utilidad [Constants](#) (GitHub).

UserDetailsServiceImpl

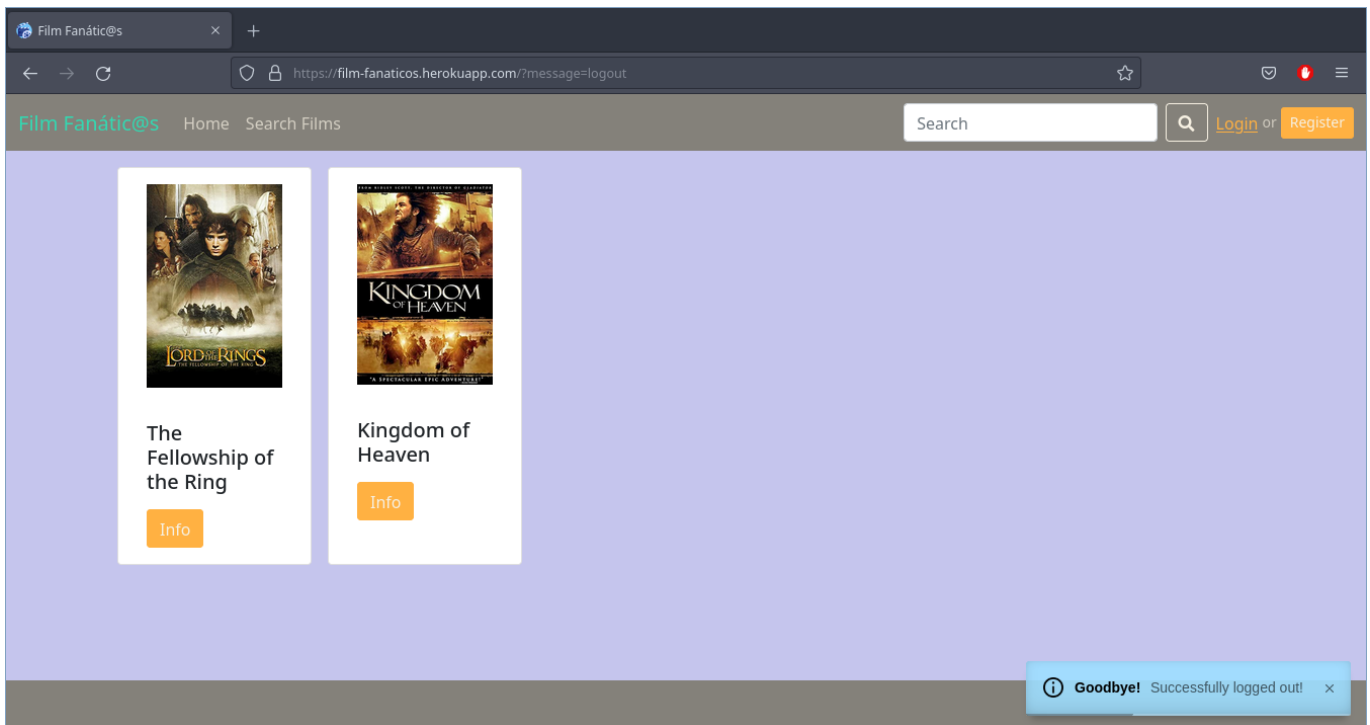
- [UserDetailsServiceImpl.java](#)

Dado que el objeto `User` implemente el `UserDetails` hubiera sido posible sencillamente implementar la interface `UserDetailsService` con el `UserService`, configurarlo en el `WebSecurityConfig` y de allí llevar todas las funciones del `User` - la autenticación incluida. Pero esto hubiera contradicho el principio de responsabilidades singular de SOLID. Además así hubiera sido más difícil de cambiar la manera de autenticación por el fuerte acoplamiento ('coupling' en inglés) entre las partes del programa.

Login/Logout Success

- [header.html](#)

Cuando un usuario iniciar sesión con éxito se reenvía a la pagina principal con el url `/?message=success`. El fragmento `header.html` contiene una función de javascript para probar los parámetros del url, y si se detecta el parámetro 'message' con el valor de 'login' se usa la librería de javascript `iziToast` para mostrar un mensaje de éxito. Lo mismo pasa cuando un usuario termina una sesión solo que el valor del parámetro es 'logout'.



```
<script type="text/javascript">
  (function() {
    console.log("checking params");
    let params = new URLSearchParams(window.location.search);
    if (params.has('message')) {
      console.log("params has message.");
      let message = params.get('message');
      if (message == 'logout') {
        console.log("message is logout");
        iziToast.info({
          title: 'Goodbye!',
          message: 'Successfully logged out!',
        });
      }
      else if (message == 'login') {
        console.log("message is login");
        iziToast.success({
          title: 'Hello!',
          message: 'Successfully logged in!',
        });
      }
    }
  })();
```

SuccessfulAuthenticationEventListener

- [SuccessfulAuthenticationEventListener.java](#)

El objeto `User` tiene un campo de tipo `LocalDateTime` llamado 'lastLogin' que debe guardar el ultimo fecha y hora que el usuario inicio sesión en la plataforma. Para realizar esta funcionamiento decide utilizar la clase `SuccessfulAuthenticationEventListener` que implemente `ApplicationListener<AuthenticationSuccessEvent>` . De esta manera el método `onApplicationEvent(AuthenticationSuccessEvent event)` se llama cada vez que se autentica con éxito.

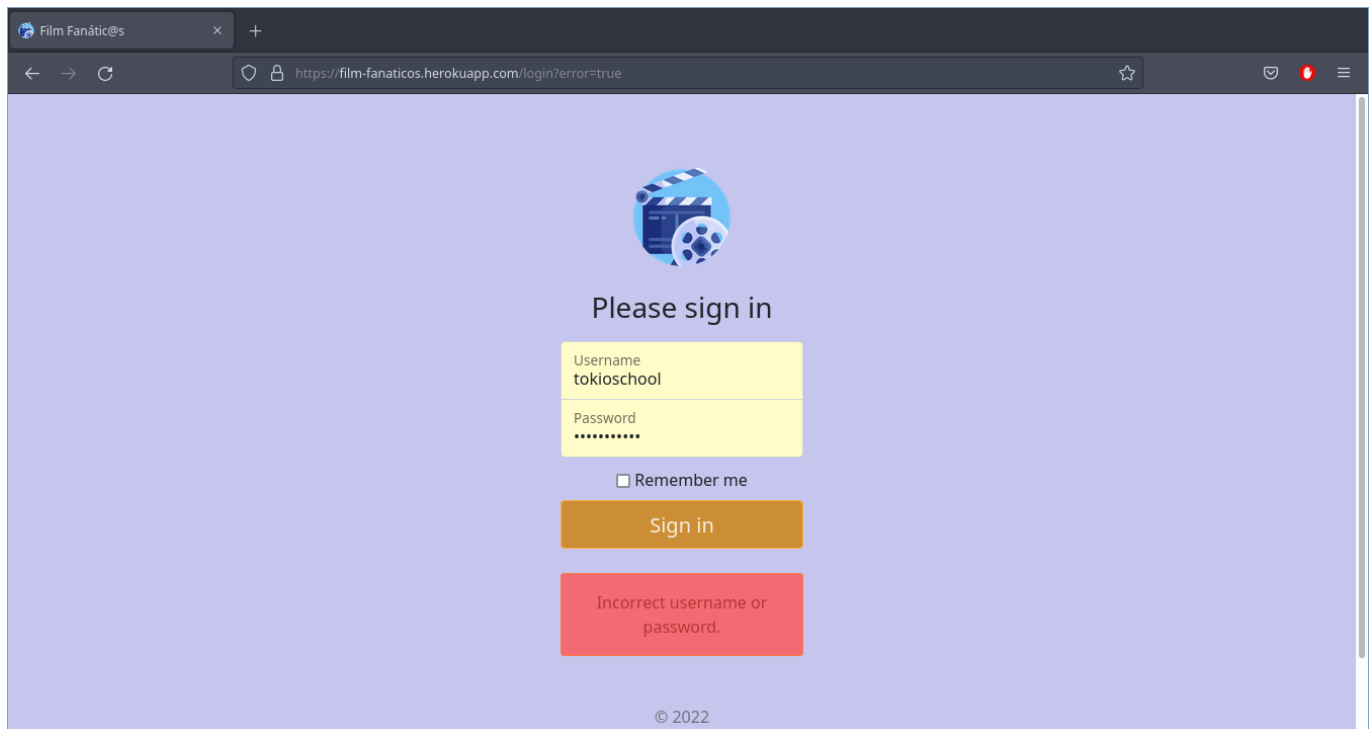
```
@Override
public void onApplicationEvent(AuthenticationSuccessEvent event) {
    User user = (User) event.getAuthentication().getPrincipal();
    log.info("User {} successfully logged in", user.getUsername());
    userService.userLoggedIn(user);
}
```

El método en sí llama otro método de `UserService` llamado `userLoggedIn(User user)` que simplemente actualiza el campo `lastLogin` del `User` y lo guarda en el base de datos.

```
@Override
@Transactional
public void userLoggedIn(User user) {
    LocalDateTime now = LocalDateTime.now();
    User loggedIn =
userRepo.findByUsername(user.getUsername()).orElseThrow(UserNotFoundException::new);
    loggedIn.setLastLogin(now);
    userRepo.save(loggedIn);
}
```

Login Failure

Si por algún motivo el intento del usuario de iniciar sesión se falla, la pagina de `login.html` se muestra un aviso así.

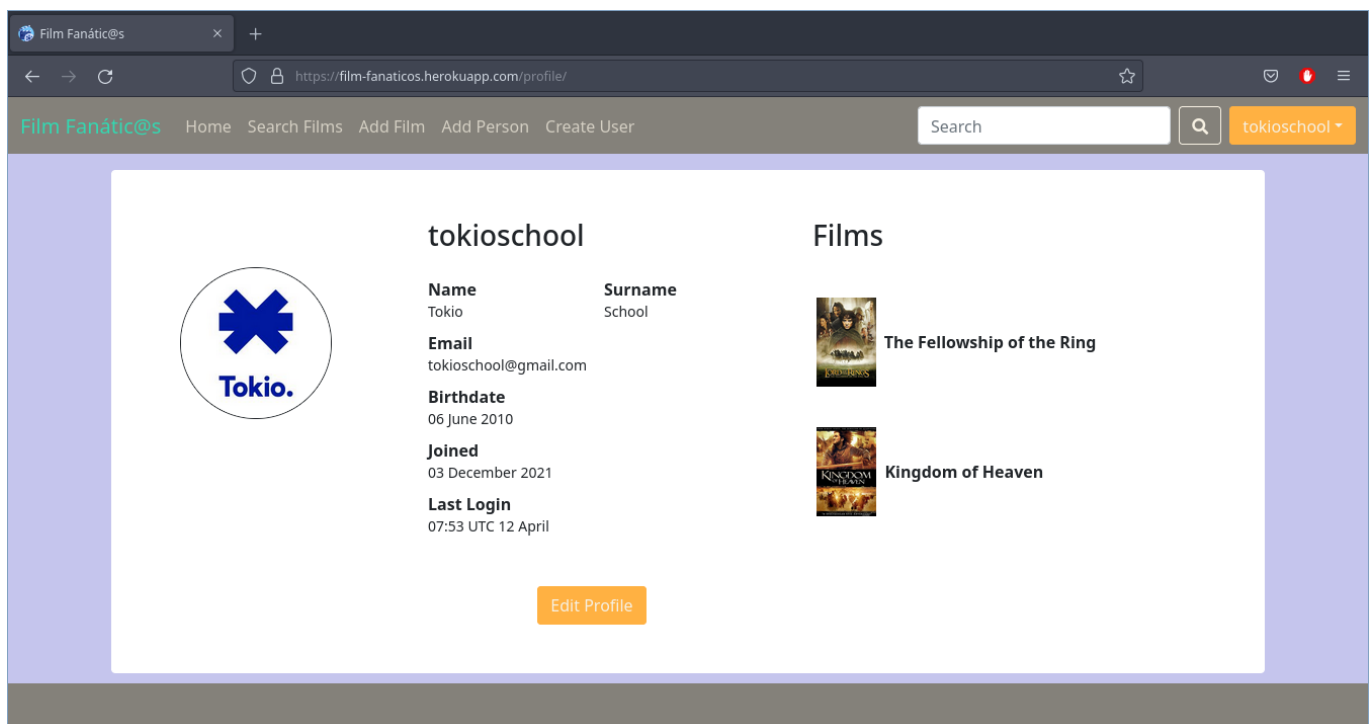


```
<div th:if="${param.error}" class="alert alert-danger" role="alert">
  Incorrect username or password.
</div>
```

User Profile

- [profile.html](#)

Al iniciar una sesión el usuario esta enviado a la pagina del perfil. El método GET del `UserController` usa los datos de la sesión para cargar el `Model` con un objeto de `User` que muestra al usuario autenticado sus datos.



```
@RequestMapping(path = "/profile", method = RequestMethod.GET)
public String profile(Model model, Principal principal) {
    if (principal != null) {
        User user = userService.getUser(principal.getName());
        model.addAttribute("user", user);
        return "profile";
    }
    else return "/";
}
```

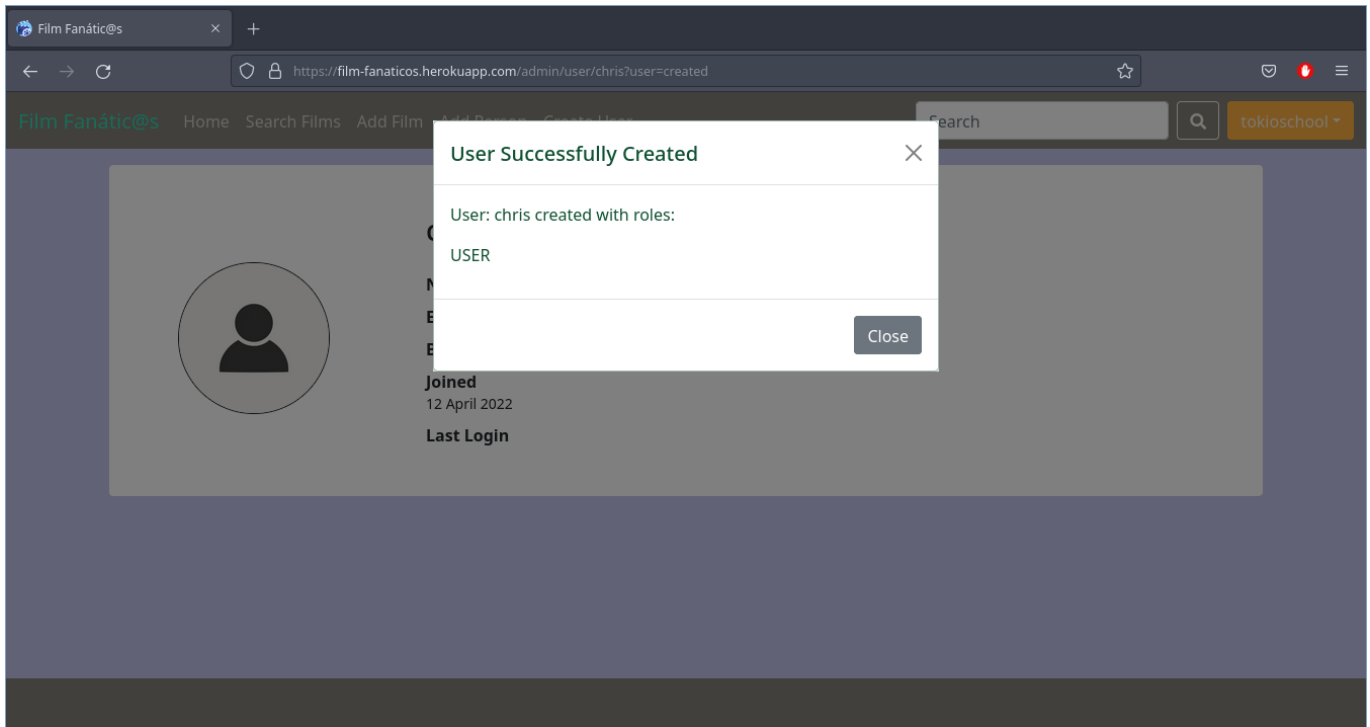
También si un usuario ADMIN crea un usuario nuevo esta enviado a la pagina de perfil para que se pueda confirmar los datos ya ingresados. Esta vez el método de `UserController` coge el username como parámetro de URL.

```
@RequestMapping(path = "admin/user/{username}", method = RequestMethod.GET)
public String profile(@PathVariable("username") String username,
                     @RequestParam(name = "user", required = false) String
requestParam,
                     Model model)
{
    User user = userService.getUser(username);
    model.addAttribute("user", user);
    return "profile";
}
```

De esta manera, un ADMIN puede ver los datos de cualquier usuario con el url path `/admin/user/{username}`.

userCreatedModal.html

- [userCreatedModel.html](#)



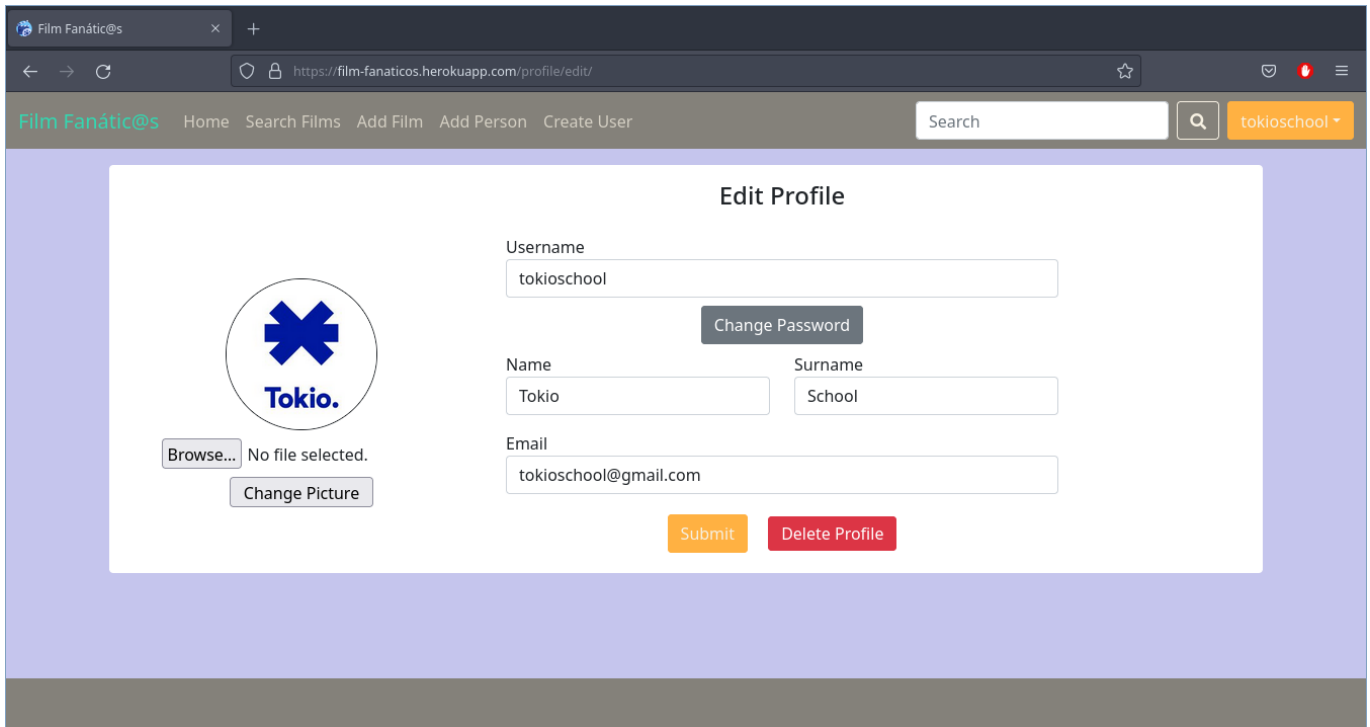
El método `register()` del `UserController` directa un usuario ADMIN a la paginá con la línea de código `return "redirect:/admin/user/" + newUser.getUsername() + "?user=created";`. El parámetro `user=created` activa el `Modal` de bootstrap que esta incluido en la paginá `profile.html` como fragment. Así se anuncia al usuario ADMIN que se ha creado el usuario nuevo con éxito. Abajo se encuentra la función de javascript que `userCreatedModal.html` contiene para activarse.

```
<script type="text/javascript">
  window.onload =
  function() {
    let params = new URLSearchParams(window.location.search);
    if (params.has('user')) {
      let message = params.get('user');
      if (message == 'created') {
        console.log("message is created");
        $('#userCreatedModal').modal("show");
      }
    }
  };
</script>
```

Editar Usuario

- [edit-user.html](#)

El perfil de usuario muestra un botón para editar los datos del usuario. Solo se muestra si es el usuario autenticado que esta viendo sus propios datos.



```
<div class="mt-5 text-center"
th:if="${#authentication.getPrincipal().getUsername() == user.username}">
    <a class="btn btn-primary profile-button mx-1"
th:href="@{/profile/edit/}">Edit Profile</a>
</div>
```

Todavía no he implementado la funcionalidad para que un usuario ADMIN puede editar los datos de otros usuarios.

El método de GET `editProfile(Model model, Principal principal)` del `UserController` es así.

```
@RequestMapping(path = "/profile/edit", method = RequestMethod.GET)
public String editProfile(Model model, Principal principal) {
    User user = userService.getUser(principal.getName());
    model.addAttribute("editUserDTO", new EditUserDTO(user));
    model.addAttribute("changePasswordDTO", new ChangePasswordDTO());
    return "edit-profile";
}
```

Tiene cuatro pasos:

1. Se carga el `User` autenticado del base de datos. Aunque se puede cargar también de la sesión de Spring Security directamente.
2. Usa el `User` que representa el usuario autenticado para crear un nuevo objeto de `EditUserDTO` y se carga al `Model`.
3. Carga el `Model` con un nuevo objeto de `ChangePasswordDTO`
4. Devuelve la página `edit-user.html`

La página [edit-user.html](#) contiene tres formularios.

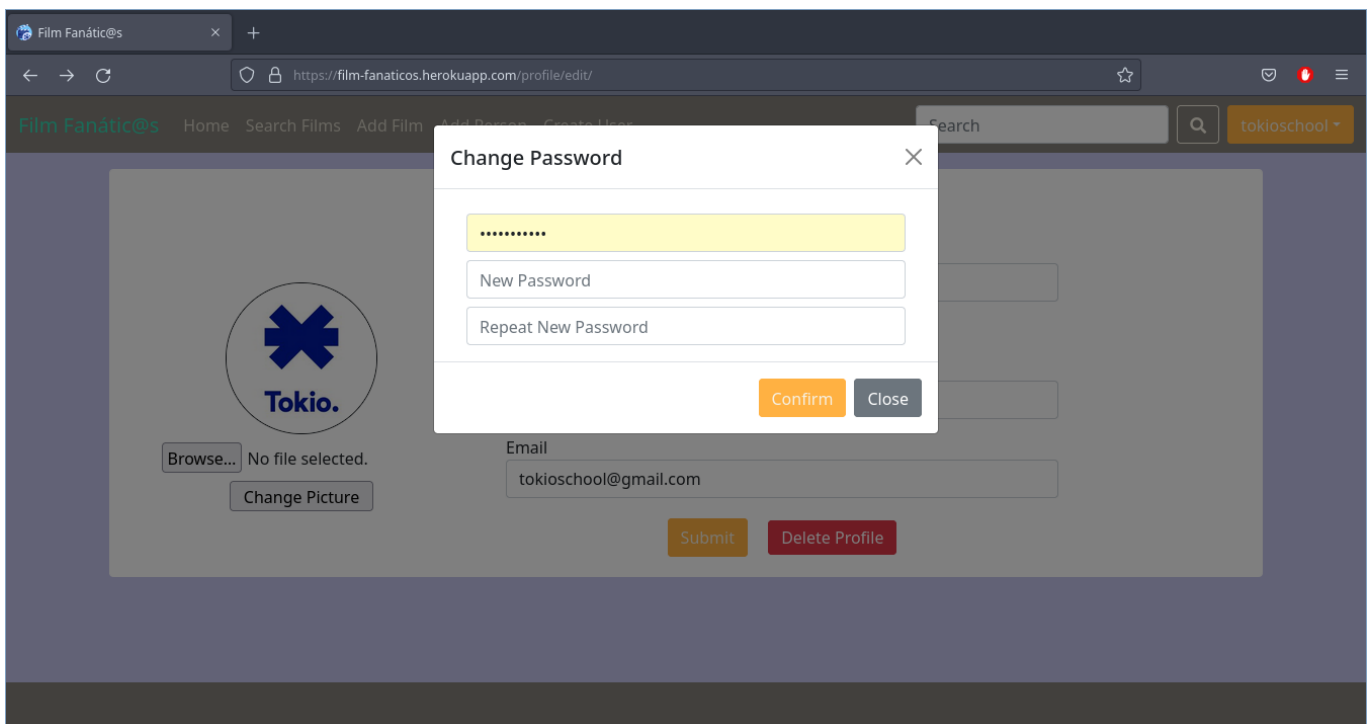
Cambiar imagen de usuario

```
<form th:method="POST" enctype="multipart/form-data" action="/user/image">
  <div class="d-flex flex-column align-items-center text-center p-3 py-5">
    <label for="image-file">
      
    </label>
    <div class="row">
      <input class="px-2 mr-4" id="image-file" type="file"
name="userImage" />
    </div>
    <div class="row mt-2">
      <input type="submit" value="Change Picture" />
    </div>
    <br/>
  </div>
</form>
```

Se envía el nuevo imagen al método `saveUserImage(String username, MultipartFile imageFile)` del `UserController` ya explicado en la sección de la creación del `User` nuevo.

Cambiar contraseña

[passwordModal.html](#) [ChangePasswordDTO.java](#)



Para cambiar la contraseña del usuario se hace clic en el botón 'Change Password' que abre un modal donde se encuentra el formulario de cambiar contraseña. El modal contiene un formulario donde el usuario debe ingresar su contraseña anterior, y su nueva contraseña dos veces. Así se puede verificar que: a) es el usuario que está cambiando su contraseña, como tiene que ingresar su contraseña existente de nuevo; b) que la nueva contraseña está correcta por el hecho de haber sido ingresado dos veces.

```
<form th:method="POST" action="/user/change-password"
th:object="{changePasswordDTO}">
  <div class="modal-body">
    <div class="row mt-2 justify-content-center">
      <div class="col mx-3">
        <input placeholder="Old Password" type="password" th:field="*
{oldPassword}" class="form-control">
      </div>
    </div>
    <div class="row mt-2 justify-content-center">
      <div class="col mx-3">
        <input placeholder="New Password" type="password" th:field="*
{password}" class="form-control">
        <div class="alert alert-warning mb-0 mt-1 py-1"
th:if="{#fields.hasErrors('{changePasswordDTO.password}')}"
th:errors="{changePasswordDTO.password}"></div>
      </div>
    </div>
    <div class="row mt-2 justify-content-center">
      <div class="col mx-3">
        <input placeholder="Repeat New Password" type="password"
th:field="*{confirmPassword}" class="form-control">
        <div class="alert alert-warning mb-0 mt-1 py-1"
th:if="{#fields.hasErrors('{changePasswordDTO.confirmPassword}')}"
th:errors="{changePasswordDTO.confirmPassword}"></div>
      </div>
    </div>
  </div>
  <div class="modal-footer">
    <button type="submit" class="btn btn-primary">Confirm</button>
    <button type="button" class="btn btn-secondary" data-bs-
dismiss="modal">Close</button>
  </div>
</form>
```

El formulario se envía al método POST de `changePassword()` del `UserController`.

```
@RequestMapping(path = "/user/change-password", method =
RequestMethod.POST)
public String changePassword(@Valid @ModelAttribute("changePasswordDTO")
ChangePasswordDTO changePassword,
                             BindingResult result,
                             Model model,
                             Authentication authentication)
```

```

{
    if (result.hasErrors()) {
        model.addAttribute("editUserDTO", new EditUserDTO((User)
authentication.getPrincipal()));
        model.addAttribute("error", true);
        return "edit-profile";
    }
    logger.info("Changing Password for {}", authentication.getName());
    userService.changePassword(authentication.getName(),
changePassword.getOldPassword(), changePassword.getPassword());
    return "redirect:/profile/edit";
}

```

El método tiene 4 pasos:

1. Si hay algún error en el `BindingResult` de `ChangePasswordDTO`, por ejemplo si los dos contraseñas nueva no se corresponden, se devuelve la página de `edit-user.html` de nuevo pero cargada con un nuevo `EditUserDTO` y otro atributo `error`. Aquí encontramos el problema que intenté resolver implementando `UserDetails` con el `User`. Como el `Model` que se usa como argumento del método solo contiene el `ChangePasswordDTO` se tiene que crear un nuevo `EditUserDTO` con un objeto `User`. Si no fuera el `User` una implementación de `UserDetails` esto implicaría una llamada al base de datos cada vez que hubiera un error que podría presentar problemas cuando se aumenta el numero de usuarios. Pero ahora veo que este problema se puede evitar también con almacenamiento en caché.
2. Se escribe un informe de la operación al `Logger`.
3. Llama el método `changePassword(String username, String oldPassword, String newPassword)` del `UserService`.
4. Envía el usuario a la página de `edit-user.html` de nuevo.

`changePassword(String username, String oldPassword, String newPassword)`

El método `changePassword(String username, String oldPassword, String newPassword)` del `UserService` es así.

```

@Override
@Transactional
public void changePassword(String username, String oldPassword, String
newPassword) {
    logger.info("Changing User's {} password", username);
    User toUpdate =
userRepo.findByUsername(username).orElseThrow(UserNotFoundException::new);
    // check user has entered correct old password
    BCryptPasswordEncoder encoder = new BCryptPasswordEncoder(-1);
    if (!encoder.matches(oldPassword, toUpdate.getPassword())) {
        throw new UnauthorizedException("Authorization Failure");
    }
    else {
        toUpdate.setPassword(encoder.encode(newPassword));
        userRepo.save(toUpdate);
    }
}

```

```
}  
}
```

Este método tiene 4 pasos:

1. Carga el usuario del base de datos.
2. Verifica que la contraseña anterior del usuario esta correcta.
3. Cambia la contraseña para la nueva ya codificada.
4. Guarda el **User** ya actualizado en le base de datos.

Cambiar datos personales de usuario

- [EditUserDTO.java](#)

Decidí usar un objeto de tipo **DTO** para la actualización de los datos personales de usuarios por que no me parecía necesario cargar la paginá con todos los datos que contiene el objeto **User**. Además, así creando una capa entre el objeto de dominio y los usuarios me parecía prudente. Así que para cambiar algo en la paginá web no será necesario ni tocar el objeto de dominio. Como los objetos de dominio forman la fundación del programa - cambiar uno de ellos implica una gran cambio por toda el programa. Mientras cambiar un objeto DTO solo implica un cambio en la capa del web y nada mas.

Un vez ingresadas, los cambios del usuario se envían al método **updateUser()** del **UserController**.

```
@RequestMapping(path = "/user/edit", method = RequestMethod.POST)  
public String updateUser(@Valid @ModelAttribute("editUserDTO") EditUserDTO  
editUserDTO,  
                        BindingResult result,  
                        Model model,  
                        Principal principal)  
{  
    if (result.hasErrors()) {  
        model.addAttribute("changePasswordDTO", new ChangePasswordDTO());  
        return "edit-profile";  
    }  
    else {  
        String loggedInUsername = principal.getName();  
        logger.info("Updating {} to {}", loggedInUsername, editUserDTO);  
        User updated = userService.updateUser(loggedInUsername,  
editUserDTO.map());  
        model.addAttribute("user", updated);  
        return "profile";  
    }  
}
```

Este método tiene 4 pasos:

1. Se hay algún error en los datos ingresado, se devuelve la pagina para que sea mostrado al usuario.
2. Escribe un informe de la operación al **Logger**
3. Llama el método **updateUser(String username, User user)** del **UserService**.

4. Devuelve la pagina `profile.html` ya cargada con los datos actualizados del usuario.

updateUser(String username, User user)

```
@Override
@Transactional
public User updateUser(String oldUsername, User user) {
    logger.info("Updating User {}", oldUsername);
    // check if username is new
    if (!oldUsername.equals(user.getUsername()))
        // and check if it's available
        && userRepo.findByUsername(user.getUsername()).isPresent()) {
        throw new UsernameAlreadyExistsException("Username unavailable.");
    }
    User toUpdate = userRepo.findByUsername(oldUsername)
        .orElseThrow(UserNotFoundException::new);
    toUpdate.update(user);
    updateSecurityContext(toUpdate.getUsername());
    return userRepo.save(toUpdate);
}
```

El método para actualizar un objeto de `User` tiene 6 pasos:

1. Primero escribe un informe de la operación al `Logger`.
2. Comprueba si el usuario ha cambiado el `username` y si lo ha cambiado, si el `username` nuevo esta disponible. Si el `username` no esta disponible se tira una excepción.
3. Carga el `User` que se va a cambiar del base de datos.
4. Llama un método de utilidad `update(User user)` del `User` que se va a actualizar para actualizar los campos del `User` ya cargado del base de datos. Se tiene que hacer asi para que Hibernate se mantenga la sesión del objeto.
5. Llama otro método de `UserServiceImpl` para actualizar el `SecurityContext` asi que el nombre del usuario en `header.html` cambia.
6. Guarda el `User` actualizada y lo devuelve.

updateSecurityContext(String username)

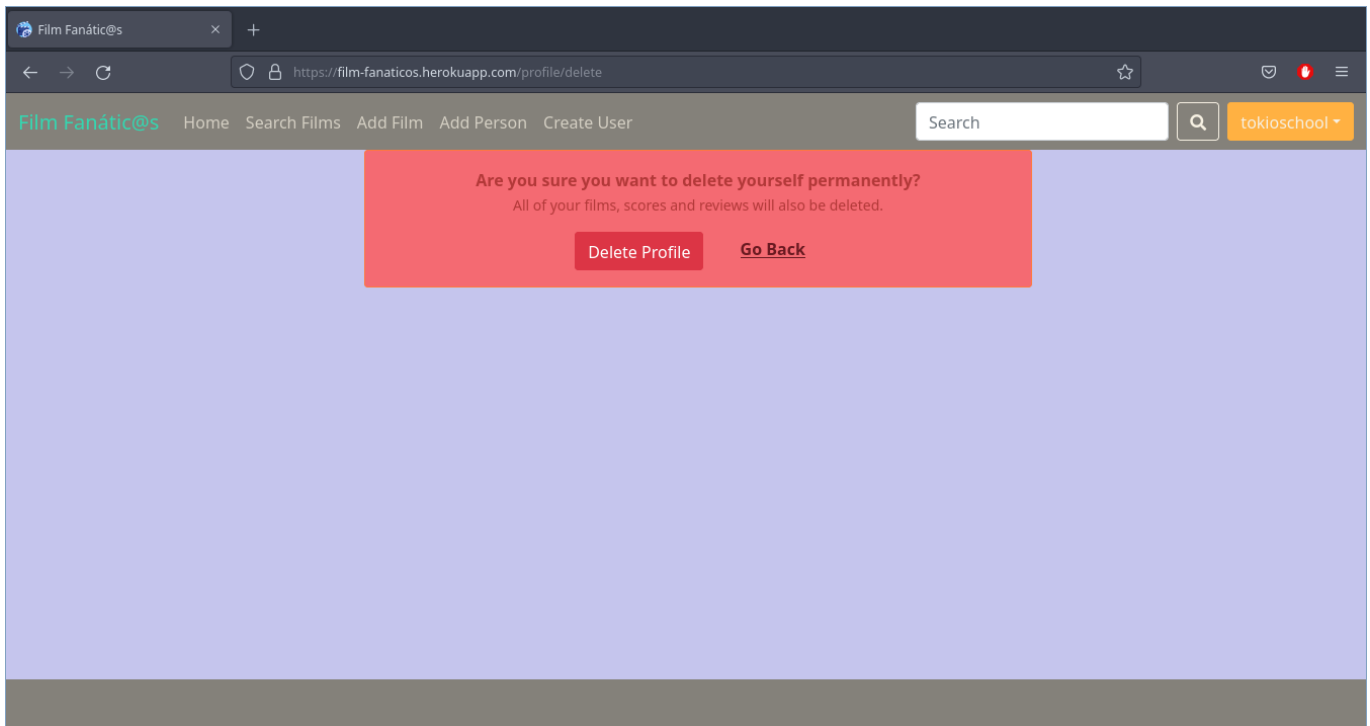
El método `updateSecurityContext` solo funciona para cambiar el nombre del usuario en el `SecurityContext`.

```
private void updateSecurityContext(String username) {
    Authentication authentication =
        SecurityContextHolder.getContext().getAuthentication();
    User userDetails = (User) authentication.getPrincipal();
    userDetails.setUsername(username);
}
```

Eliminar Usuario

- [delete-user.html](#)

En la página [edit-user.html](#) se encuentra un botón para eliminar el usuario de la plataforma. Como la página en sí es solo disponible para que un usuario autorizado edite sus propios datos, la opción de eliminarse también solo se dispone al usuario autenticado.



```
<a class="btn btn-danger profile-button mx-2 my-1 py-1"
th:href="@{/profile/delete}">Delete Profile</a>
```

El botón se envía el usuario a una nueva página [delete-user.html](#) que pide confirmación de la eliminación.

```
<div class="alert alert-danger" role="alert">
  <p class="text-center">
    <strong>Are you sure you want to delete yourself permanently?
  </strong>
    <br>
    <small>All of your films, scores and reviews will also be deleted.
  </small>
  </p>
  <form></form>
  <form class="text-center" th:method="GET" th:action="@{/user/delete}">
    <button class="btn btn-danger profile-button mx-2"
type="submit">Delete Profile</button>
    <a href="#" class="alert-link mx-4" onclick="history.back()">Go
Back</a>
  </form>
</div>
```


deleteUser(Principal principal)

Al pinchar el botón para confirmar. Se envía un solicitud GET al método `deleteUser(Principal principal)` del `UserController` que simplemente coge el nombre del usuario autenticado y llama al método `deleteUser(String username)` del `UserService`. Al eliminar el `User` se envía el usuario a la pagina de 'logout'.

```
@RequestMapping(path = "/user/delete", method = {RequestMethod.GET})
public String deleteUser(Principal principal) throws UnauthorizedException
{
    logger.info("Deleting User {}", principal.getName());
    userService.deleteUser(principal.getName());
    return "redirect:/logout";
}
```

Personas

- [Person.java](#)
- [PersonTypeEnum.java](#)
- [PersonController.java](#)
- [PersonServiceImpl.java](#)
- [PersonRepository.java](#)
- [new-person.html](#)

Las personas que crean las películas que nuestros usuarios interesan, como directores, actores... forman una parte importante del dominio de nuestro plataforma.

Al llegar a la pagina para añadir una persona nueva el usuario tiene ingresar el nombre, apellidos y tipo, que ya esta determinada según el `PersonTypeEnum`.

The screenshot shows a web browser window with the URL `https://film-fanaticos.herokuapp.com/person/add`. The page has a navigation bar with links: Home, Search Films, Add Film, Add Person, and Create User. A search bar and a user profile 'tokioschool' are also visible. The main form area has a light purple background and contains three input fields: 'First Name', 'Last Name', and a dropdown menu for 'Type' with 'Screen Writer' selected. An orange 'Submit' button is centered below the fields.

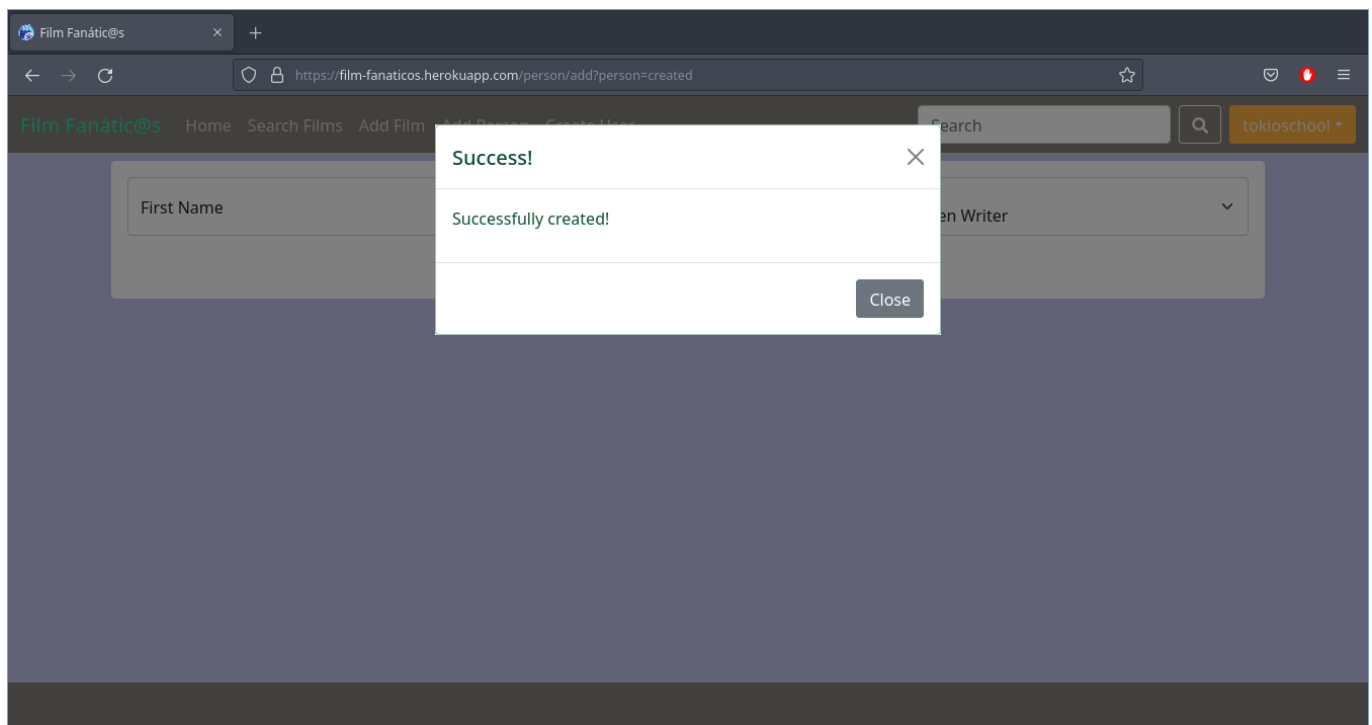
PersonController.addPerson()

Al enviar los datos del person nuevo al servidor - llegan al método `addPerson` del `UserController`.

```
@RequestMapping(path = "/person/add", method = RequestMethod.POST)
public String addPerson(@ModelAttribute @Valid Person person,
                        BindingResult result,
                        Model model,
                        Principal principal)
{
    if (result.hasErrors()) {
        log.error("Creation of Person {} failed because: {}", person,
result.getAllErrors().toArray());
        return "new-person";
    }
    else {
        log.info("User {} added Person {}", principal.getName(), person);
        service.addPerson(person);
        return "redirect:/person/add?person=created";
    }
}
```

Este método tiene 4 pasos:

1. Si hay algún error - los devuelve al usuario.
2. Escribe al `log` que el usuario autenticado ha creado una persona nueva.
3. Llama el método `addPerson(Person person)` del `PersonService`.
4. Reenvía el usuario a la pagina para crear personas donde se muestra un mensaje de confirmación.



PersonService.addPerson(Person person)

```
@Override
public Person addPerson(Person person) {
    log.info("Adding Person: {}", person);
    return personRepo.save(person);
}
```

Este método solo guarda el **Person** nuevo en el base de datos.

Películas

Aparte del **User** el objeto de dominio **Film** es lo más esencial al programa. El propósito de la plataforma es que usuarios pueden subir y buscar datos de pelis diferentes.

objeto de dominio

***Film.java**

En el objeto de dominio de **Film** hay unas cosas notables.

1. He añadido dos indices al base de datos para ayudar aumentar la velocidad de las buscadass:

```
@Table(indexes = {
    @Index(name = "title_index", columnList = "title"),
    @Index(name = "year_index", columnList = "year")})
```

2. Los dos propiedades **scores** y **reviews** tienen **orphanRemoval = true** para que no si se elimina una película del base de datos, no se quedan ningunos **Review** ni **Score** sin una película.
3. He añadido otra propiedad **uri** (universal resource identifier) para cada **Film**. La razón es para que se puede usar una identificador en el url de la pagina. Por ejemplo, le película 'The Fellowship of the Ring' hecho en 2002 tiene el **uri** de 'the-fellowship-of-the-ring-2002' y el url para ir a su pagina es 'www.localhost:8080/the-fellowship-of-the-ring-2002'. Asi los url están más legibles y no mostramos ningún dato del sistema al usuario, en este caso el **id** del **Film**. El **uri** tiene la ventaja de tener el titulo y el año de producción de la película. A veces hay películas con el mismo titulo pero casi nunca esta producidas en el mismo año. Asi podemos evitar colisiones posibles entre los urls de la películas.

Para generar el **uri** usamos un método estático de **StringUtils**

```
public static String getFilmUri(String filmTitle, int filmYear) {
    return String.join("-", filmTitle.toLowerCase(Locale.ROOT).split(" ")) +
    filmYear;
}
```

También hay un método estático de **StringUtil** para sacar el titulo de la película del **uri**:

```
public static String getFilmTitleFromUri(String filmUri) {
    String lowerCaseTitle = filmUri.substring(0, filmUri.length() - 5)
        .replace("-", " ");
    return StringUtils.capitalize(lowerCaseTitle);
}
```

Aquí uso la clase `StringUtils` de `ApacheCommons` para capitalizar el primer carácter de cada palabra.

- Además he añadido un campo más `int avgScore` para guardar el promedio de la puntuación que los usuarios han dado a la película. La puntuación media está calculada cada vez que se añade o quita un `Score` del `Film`.

```
public void calculateAverageScore() {
    this.avgScore = scores.stream().map(Score::getValue).reduce(0,
        Integer::sum) / scores.size();
}

public void addScore(Score score) {
    score.setFilm(this);
    scores.add(score);
    calculateAverageScore();
}

public void removeScore(Score score) {
    scores.remove(score);
    calculateAverageScore();
}
```

crear película

- [new-film.html](#)
- [FilmController.java](#)
- [FilmService.java](#)
- [FilmRepository.java](#)

Un usuario autenticado encontraría un link en el header para crear una película. Al pinchar el link llamará el método GET de `createFilm(Model model)` del `FilmController` que carga el modelo con todos las personas del base de datos antes de devolver la página `new-film.html`.

The screenshot shows a web browser window with the URL `https://film-fanaticos.herokuapp.com/films/add`. The page has a navigation bar with links: Home, Search Films, Add Film, Add Person, Create User, and a search bar. The main form is titled 'Add Film' and contains the following fields:

- Title:** A text input field.
- Year:** A dropdown menu showing '1990'.
- Duration:** A dropdown menu showing '0'.
- Director:** A dropdown menu showing 'Peter Jackson'.
- Poster Image:** A button labeled 'Browse...' and a text area showing 'No file selected.'
- Synopsis:** A large text area.
- Actors:** A list box containing 'Orlando Bloom' and 'Brad Pitt'.
- Screenwriters:** An empty list box.
- Composers:** An empty list box.
- Cinematographers:** An empty list box.

At the bottom of the form are two buttons: 'Submit' (orange) and 'Reset' (dark grey).

```
@RequestMapping(path = "films/add", method = RequestMethod.GET)
public String createFilm(Model model) {
    Film newFilm = new Film();

    model.addAttribute("film", newFilm);
    model.addAttribute("directors", personService.getPeopleByType(DIRECTOR));
    model.addAttribute("actors", personService.getPeopleByType(ACTOR));
    model.addAttribute("screenwriters",
personService.getPeopleByType(SCREEN_WRITER));
    model.addAttribute("cinematographers",
personService.getPeopleByType(CINEMATOGRAPHER));
    model.addAttribute("composers", personService.getPeopleByType(COMPOSER));
    return "new-film";
}
```

La página en sí dispone un formulario para ingresar todos los datos de la película. El formulario se envía al método POST `createFilm()` del `FilmController`.

```
@RequestMapping(path = "films/add", method = RequestMethod.POST)
public String createFilm(@RequestParam("posterImage") MultipartFile
posterImage,

                        @ModelAttribute("film") @Valid Film film,
                        BindingResult result,
                        Model model,
                        Authentication auth)

{
    if (result.hasErrors()) {
        return "new-film";
    }
    else {
```

```

        film.setUser((User) auth.getPrincipal());
        Film createdFilm = filmService.addFilm(film);
        if (!posterImage.isEmpty()) {
            createdFilm = filmService.savePoster(film, posterImage);
        }
        model.addAttribute("film", createdFilm);
        return "redirect: /films/" +
StringUtil.getFilmUri(createdFilm.getTitle(), createdFilm.getYear());
    }
}

```

Este método tiene 5 pasos:

1. Primero devuelva la pagina si se encuentra errors en el **BindingResult**.
2. Pone el usuario ya autenticado como propietario del **Film**.
3. Llama el método de **addFilm(Film film)** del **FilmService** para guardar el **Film** en el base de datos.
4. Si hay una imagen para la película llama el método **savePoster(Film film, MultipartImage posterImage)** del **FilmService** para guardar el imagen de la película y actualizar el **Film** en el base de datos.
5. Añade la película nueva al modelo.
6. Reenvía el usuario a la paginá donde se puede ver los datos de la película ya creada.

addFilm(Film film)

```

@Override
@Transactional
public Film addFilm(Film film) {
    film.setUri(StringUtil.getFilmUri(film.getTitle(), film.getYear()));
    log.info("Saving Film {}", film);
    return filmRepo.save(film);
}

```

El método para guardar una pelicular nueva es bastante simple.

1. Primero añade el **uri** al **Film** nuevo.
2. Lo guarda en el base de datos.
3. Al final devuelve el **Film** ya guardado.

savePoster(Film film , MultipartImage posterImage)

```

@Override
@Transactional
public Film savePoster(Film film, MultipartFile posterImage) {
    Film toUpdate = filmRepo.findByTitleIgnoreCase(film.getTitle())
        .orElseThrow(() -> new FilmNotFoundException());
}

```

```
String fileName = StringUtil.getFilmPosterFilename(film.getTitle(),
film.getYear(), posterImage.getContentType());
fileService.saveFile(posterImage, fileName);

toUpdate.setPoster(fileName);
return filmRepo.save(toUpdate);
}
```

EL método para guardar el imagen de la película tiene 5 pasos:

1. Carga el **Film** para actualizar del base de datos.
2. Llama el método de **StringUtil** para generar el nombre del archivo del imagen. Aunque ahora me he dado cuenta que se podría usar el **uri** para esto también.
3. Llama el método **saveFile(MultipartFile file, String fileName)** de **FileService** ya explicado en la sección de la inscripción de usuarios nuevos.
4. Actualiza el **Film** con el nombre del archivo del imagen.
5. Guarda el **Film** actualizado en el base de datos y lo devuelve.

buscar películas

- [FilmController.java](#)
- [FilmServiceImpl.java](#)
- [FilmSearchImpl.java](#)
- [FilmSearchCriteriaEnum.java](#)
- [FilmSearchStrategy.java](#)
- [FilmTitleSearch.java](#)

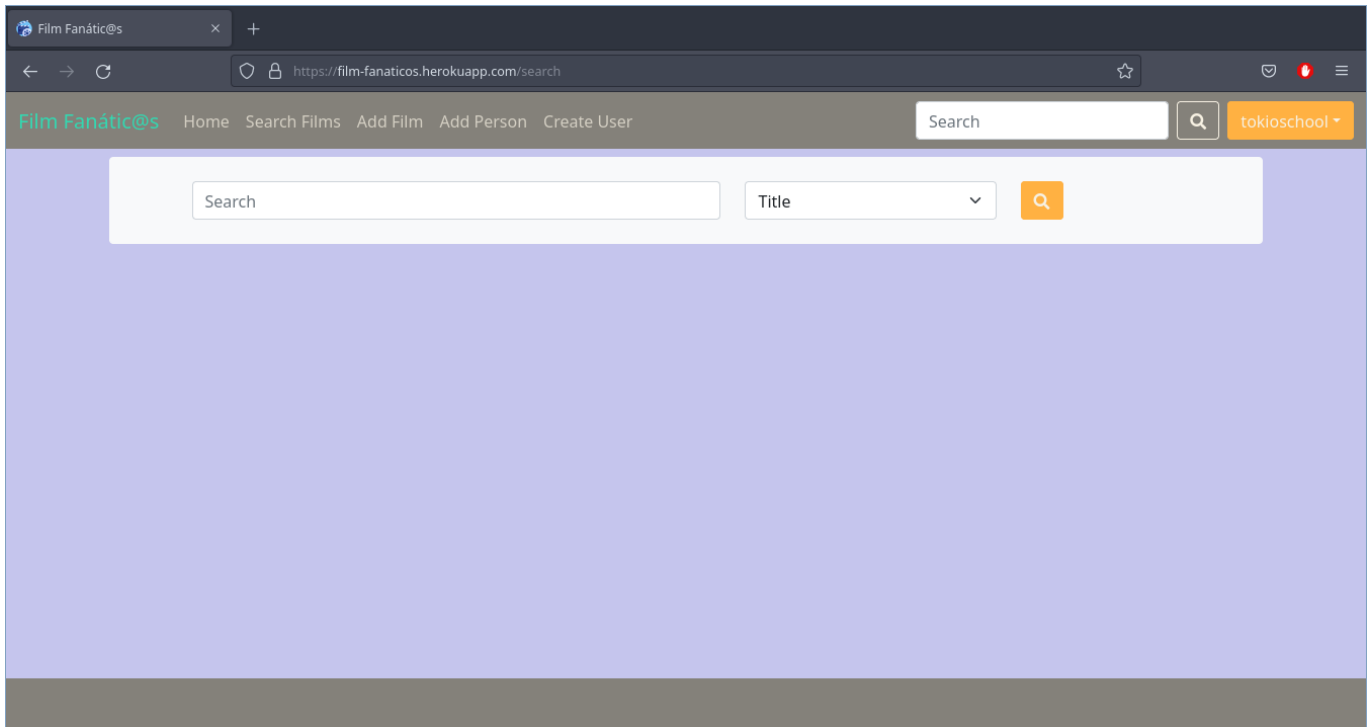
Para buscar una película en la plataforma el usuario tiene dos opciones.

1. En el header de cualquier pagina se encuentra una barra de busqueda donde puede ingresar el titulo de la película que quiere buscar.

```
<form class="d-flex align-items-center" th:method="get"
th:action="@{/films/search}">
  <input class="form-control" type="text" name="query" id="query"
placeholder="Search" aria-label="search">
  <input type="hidden" name="criteria" id="criteria" value="title">
  <button class="btn btn-outline-success mx-2" type="submit">
    <i class="fas fa-search"></i>
  </button>
</form>
```



2. También se encuentra en el header el link para llegar al **film-search.html** donde puede buscar una película segunda varias criteria.



El formulario de `film-search.html` se lee así.

```
<form class="row d-flex justify-content-center p-4 align-items-middle"
th:method="GET" th:action="@{films/search}">
  <div class="col-6">
    <input class="form-control" type="text" name="query" id="query"
placeholder="Search">
  </div>
  <div class="col-3">
    <select class="form-select" name="criteria" id="criteria">
      <option th:each="c : ${criteria}" th:value="${c}"
th:text="${c.displayName}"></option>
    </select>
  </div>
  <div class="col-2">
    <button class="btn btn-primary" type="submit">
      <i class="fas fa-search"></i>
    </button>
  </div>
</form>
```

FilmSearchCriteriaEnum

Los 'criteria' de la búsqueda que están incluido con el formulario o como un input oculto, como en 1., o elegido por el usuario están representados en el programa por el `FilmSearchCriteriaEnum`.

```
@Slf4j
public enum FilmSearchCriteriaEnum {
  TITLE("Title"),
  YEAR("Year"),
```



```

    MAX_DURATION("Max Length"),
    AVG_SCORE("Average Score"),
    ACTOR("Actor"),
    DIRECTOR("Director"),
    SCREENWRITER("Screenwriter"),
    CINEMATOGRAPHER("Cinematographer"),
    COMPOSER("Composer");
    ...

```

FilmSearchStrategy

Cada criteria ENUM tiene su propio implementación de la interface `FilmSearchStrategy`. Así que he usado el patrón de diseño estrategia para programar la búsqueda por criteria diferentes. Lo he usado para que sean separadas las responsabilidades de las partes diferentes de la programa y también para que sea más fácil añadir otros criteria de búsqueda más adelante.

```

public interface FilmSearchStrategy {

    Set<Film> searchFilm(String searchParam);

}

```

Y cada implementación tiene su propio método en el `FilmRepository`.

Por ejemplo el Enum `AVG_SCORE` tiene el `FilmAverageScoreSearch` como su pareja.

```

public class FilmAverageScoreSearch implements FilmSearchStrategy {

    private final FilmRepository repository;

    public FilmAverageScoreSearch(FilmRepository repository) {
        this.repository = repository;
    }

    @Override
    public Set<Film> searchFilm(String searchParam) {
        return
        repository.findByAvgScoreGreaterThanOrEqualTo(Integer.parseInt(searchParam));
    }
}

```

Que llama al método `findByAvgScoreGreaterThanOrEqualTo(int score)` de `FilmRepository`.

FilmSearchImpl

Es el interface `FilmSearch` que toma el criteria de búsqueda y los parámetros de ella y crear una instancia de la implementación correcta de `FilmSearchStrategy` y llama su método `searchFilm(String`

`searchParam).`

```
@Override
public Set<Film> searchFilm(String searchParam, FilmSearchCriteriaEnum
searchCriteria) {
    switch (searchCriteria) {
        case TITLE:
            return new FilmTitleSearch(repository).searchFilm(searchParam);
        case YEAR:
            return new FilmYearSearch(repository).searchFilm(searchParam);
        case MAX_DURATION:
            return new
FilmMaxDurationSearch(repository).searchFilm(searchParam);
        case AVG_SCORE:
            return new
FilmAverageScoreSearch(repository).searchFilm(searchParam);
        case ACTOR:
            return new FilmActorSearch(repository).searchFilm(searchParam);
        case DIRECTOR:
            return new FilmDirectorSearch(repository).searchFilm(searchParam);
        case SCREENWRITER:
            return new
FilmScreenwriterSearch(repository).searchFilm(searchParam);
        case CINEMATOGRAPHER:
            return new
FilmCinematographerSearch(repository).searchFilm(searchParam);
        case COMPOSER:
            return new FilmComposerSearch(repository).searchFilm(searchParam);
    }
    throw new FilmNotFoundException();
}
```

Ruta de Solicitud de Búsqueda

Al hacer una búsqueda de película en la plataforma, la solicitud sigue el siguiente pasos.

1. Primero el formulario llega al `FilmController` que pasa la criteria y los parámetros al `FilmService`.

```
@RequestMapping(path = "films/search", method=RequestMethod.GET)
public String searchFilm(@RequestParam(name = "query") String query,
                        @RequestParam(name = "criteria") String
criteria,
                        Model model) {
    Set<Film> results = filmService.searchFilms(query, criteria);
    model.addAttribute("films", results);
    return "searched-film";
}
```

2. El `FilmService` llama al `FilmSearch` para que se busque la película según la criterio dada.

```
@Override
public Set<Film> searchFilms(String searchParam, String searchCriteria) {
    return filmSearch.searchFilm(searchParam,
        FilmSearchCriteriaEnum.fromString(searchCriteria));
}
```

Se genera el `FilmSearchCriteriaEnum` con un método de utilidad de dada clase `fromString(String criteria)`.

```
public static FilmSearchCriteriaEnum fromString(String value) {
    for (FilmSearchCriteriaEnum criteria : FilmSearchCriteriaEnum.values())
    {
        if (criteria.getDisplayValue().equalsIgnoreCase(value)) {
            return criteria;
        }
        else {
            try {
                if (FilmSearchCriteriaEnum.valueOf(value) == criteria) {
                    return criteria;
                }
            }
            catch (Exception e) {
                log.error("Error when fetching FilmSearchCriteriaEnum", e);
            }
        }
    }
    throw new IllegalArgumentException("No matching search criteria found");
}
```

3. El `FilmSearch` crea y usa la correcta implementación de `FilmSearchStrategy` para devolver las películas que corresponden a los datos ingresados.
4. Cuando se devuelva los `Film` que corresponden con la criterio y los parámetros ingresados, el `FilmController` carga el `Model` y devuelve la página `search-film.html` que muestra una carta de cada película.

```
<div th:fragment="searched-film" class="row">
    <div class="card m-2" th:each="film : ${films}" style="width: 12rem;">
        
        <div class="card-body">
            <h5 class="card-title" th:text="${film.title}">Film Title</h5>
            <p class="card-text" th:text="${film.synopsis}">Synopsis</p>
            <a th:href="@{/films/__$${film.url}__}" class="btn btn-
primary">Info</a>
```

```
        </div>
    </div>
</div>
```

Estas cartas de la información de las películas también están incluido en la pagina inicial como fragmento y inicializado por el `WebController` al cargar la paginá `index.html`.

```
@RequestMapping(path = "/*", method = {GET, RequestMethod.POST})
public String index(Model model) {
    logger.info("Connection... serving index.html");
    model.addAttribute("films", filmService.getAllFilms());
    return "index";
}
```

Info de Película

- [film.html](#)

Al pinchar el botón la carta de una película que se muestra en la pagina de búsqueda, se enviá una solicitud GET a método `filmInfo(String filmUri, Model model, Authentication auth)`.

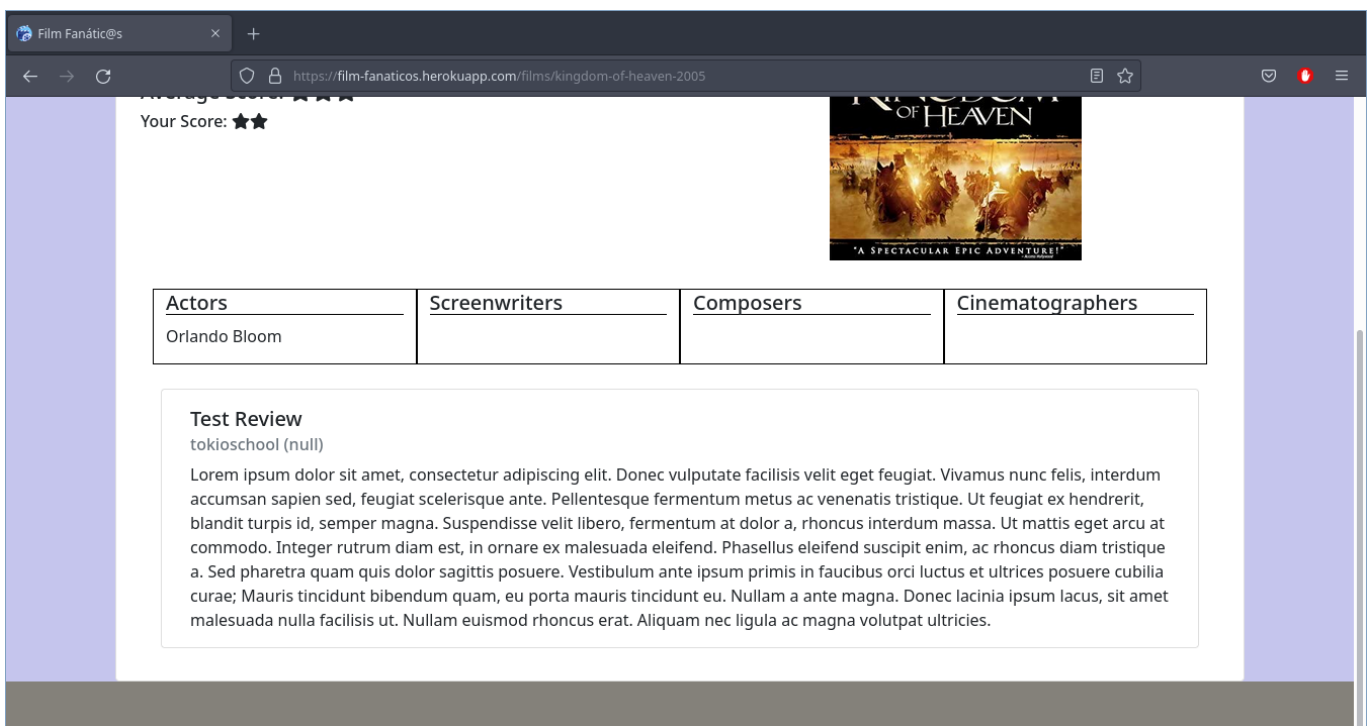
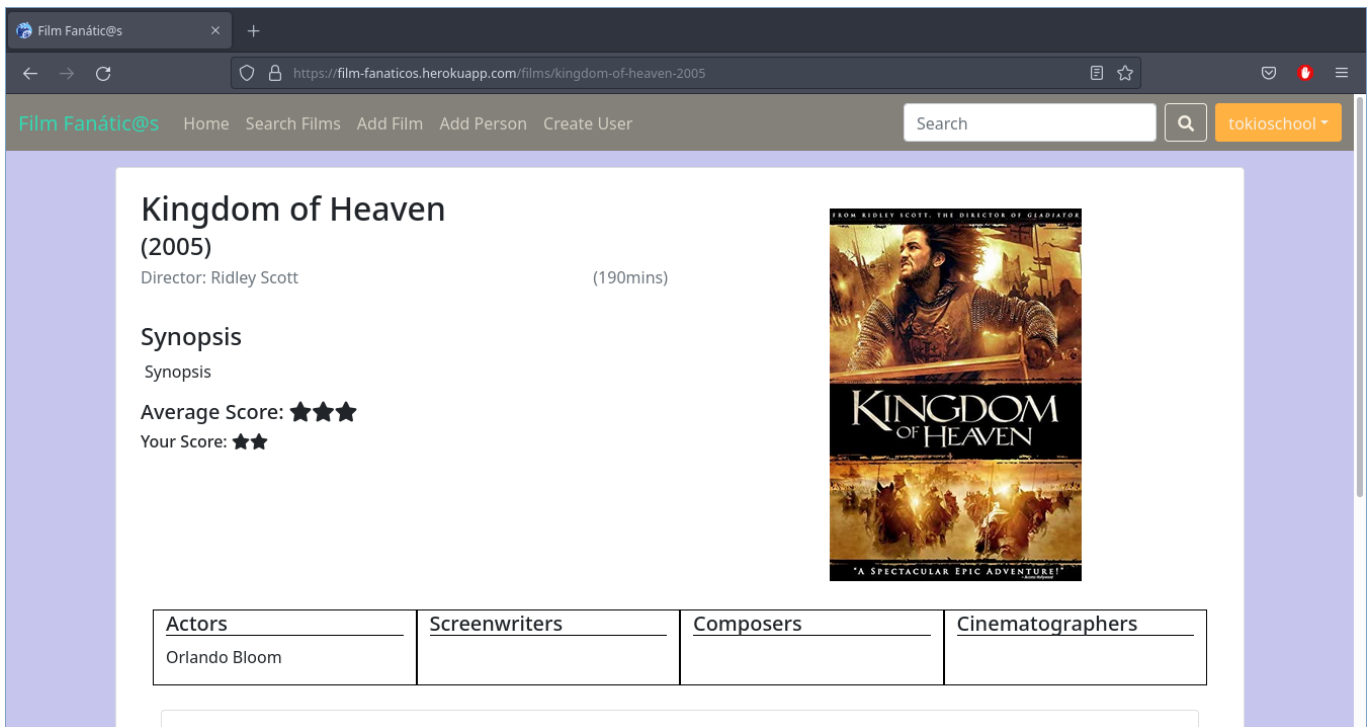
```
@RequestMapping(path = "films/{filmUri}", method = RequestMethod.GET)
public String filmInfo(@PathVariable("filmUri") String filmUri,
                      Model model,
                      Authentication authentication) {
    Film film = filmService.getFilmByUri(filmUri);
    model.addAttribute("film", film);

    if (authentication != null) {
        User authenticatedUser = (User) authentication.getPrincipal();
        Optional<Score> userScore = film.getScores()
            .stream()
            .filter(scr -> scr.getUser().getId() == authenticatedUser.getId())
            .findFirst();
        if (userScore.isPresent()) {
            model.addAttribute("score", userScore.get());
        } else {
            model.addAttribute("newScore", new Score());
        }
    }
    return "film";
}
```

Este método tiene 5 pasos:

1. Primero llama el método de `getFilmByUri(String uri)` del `FilmService` para cargar la película solicitada.
2. Carga el `Model` con la peli.

3. Si la sesión de seguridad contiene un usuario autenticado, busca entre los **Score** del **User** que representa el usuario autenticado para ver si ya ha dado una puntuación a la película en cuestión.
4. Si ya hay una puntuación de la película que la ha dado el usuario - se carga el modelo con esta; y si no hay, se carga con un objeto nuevo de **Score** por si caso el usuario quiere darla una puntuación.
5. Al final devuelve la pagin  **film.html**



La pagin  siempre muestra todo la informaci  n de la pel  cula, las rese  as, su puntuaci  n media incluida y si el usuario esta autenticado muestra o la puntuaci  n que se la ha dado ya, o le muestra la opci  n darla ahora. Usa un icono de estrella para muestra al usuario la puntuaci  n de la pel  cula.

```
<h5 class="mt-3">
  Average Score:
```

```

        <i th:each="i : ${#numbers.sequence(1, film.averageScore)}" class="fa
fa-star"></i>
</h5>
<form></form>
<form sec:authorize="isAuthenticated()" th:if="${newScore}"
th:object="${newScore}" class="row row-cols g-1 align-items-centre"
th:method="POST"
th:action="@{__${#httpServletRequest.requestURI}__/score/}">
    <div class="col-4">
        <select class="form-select" th:field="*{value}" >
            <option th:each="i : ${#numbers.sequence(1, 5)}"
th:value="${i}" th:text="${i}">Score</option>
        </select>
    </div>
    <div class="col-8">
        <button class="btn btn-primary btn-rounded" type="submit">Submit
Your Score</button>
    </div>
</form>
<h6 sec:authorize="isAuthenticated()" th:if="${score}" class="mt-1">
    Your Score:
    <i th:each="i : ${#numbers.sequence(1, score.value)}" class="fa fa-
star"></i>
</h6>

```

Dar puntuación a una pelí

- [Score.java](#)
- [FilmController.java](#)
- [FilmService.java](#)

The Fellowship of the Ring

(2001)

Director: Peter Jackson

(178mins)

Synopsis

Synopsis

Average Score: ★★★★★

▼

Submit Your Score



Al dar la puntuación a una película el se envía al método `addScore(String filmUri, Score score, BindingResult result, Authentication auth)` del `FilmController`.

```

@RequestMapping(path = "films/{filmUri}/score", method =
RequestMethod.POST)

```

```
public String addScore(@PathVariable("filmUri") String filmUri,
                      @ModelAttribute("newScore") @Valid Score score,
                      BindingResult result,
                      Authentication auth) {
    if (result.hasErrors()) {
        return "film";
    }
    score.setUser((User) auth.getPrincipal());
    filmService.addScore(filmUri, score);

    return "redirect: /films/" + filmUri;
}
```

El método tiene 4 pasos:

1. Si hay algún error en el `BindingResult` del `Score` - se devuelve la pagina para mostrarlo al usuario.
2. Coge el `User` del `Authentication` objeto y lo aplica al `Score` nuevo.
3. Llama el método `addScore(String filmUri, Score score)` del `FilmService`
4. Reenvía el usuario a la pagina `film-info.html` para mostrar el `Score` nuevo actualizado.

FilmService.addScore(String filmUri, Score score)

```
@Override
@Transactional
public Film addScore(String filmUri, Score score) {
    Film toUpdate = getFilmByUri(filmUri);
    toUpdate.addScore(score);
    return filmRepo.save(toUpdate);
}
```

El método de `FilmService` tiene 3 pasos:

1. Llama su método `getFilmByUri(String filmUri)` para carga el `Film` a que le va dar la puntuación.
2. Añade el `Score` nuevo al `Film` ya cargado.
3. Guarda el `Film` en el base de datos y lo devuelve.

FilmService.getFilmByUri(String uri)

- [StringUtil.java](#)

```
@Override
@Transactional()
public Film getFilmByUri(String filmUri) {
    String filmTitle = StringUtil.getFilmTitleFromUri(filmUri);
    return filmRepo.findByTitleIgnoreCase(filmTitle)
        .orElseThrow(() -> new FilmNotFoundException());
}
```

Este método tiene dos pasos:

1. Llama el método `StringUtil.getFilmTitleFromUri(String filmUri)` para tener el título de la película.
2. Carga el `Film` del base de datos.

El método `StringUtil.getFilmTitleFromUri(String filmUri)` funciona así:

```
public static String getFilmTitleFromUri(String filmUri) {  
    String lowerCaseTitle = filmUri.substring(0, filmUri.length() - 5)  
        .replace("-", " ");  
    return StringUtils.capitalize(lowerCaseTitle);  
}
```

Usa el objeto `StringUtils` de la librería `Apache Commons`. Aun que he dado cuenta después que no es necesario hacer lo así. Solo se necesita hacer un índice con el campo `uri` de `Film` en el base de datos. Así que no sería necesario hacer el transformación de `title + year` y `filmUri` todo el tiempo.

Review

- [Review.java](#)
- [ReviewDTO.java](#)
- [ApiReviewController.java](#)
- [ReviewServiceImpl.java](#)

Las reseñas de la películas están importantes para nuestros usuarios - dejan que los usuarios compartir sus opiniones sobre la películas ya guardadas en nuestro plataforma. Como estaba descrito en el documento de diseño los objetos de `Review` están disponibles, para crear y guardar, por un RESTful API.

Las peticiones HTTP al API usan el `ReviewDTO` para enviar y recibir los datos.

```
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
@JsonIgnoreProperties(ignoreUnknown = true)  
public class ReviewDTO {  
    @Schema(description = "Title of Review", example = "A Really Good Film")  
    private String title;  
    @Schema(description = "Review text", example = "When I watched this film  
I was amazed!...")  
    private String text;  
    @Schema(description = "Date of Review creation", example = "2021-02-02")  
    private LocalDate date;  
    @Schema(description = "Username of Review owner", example = "username")  
    private String user;  
    @Schema(description = "Title of Film reviewed", example = "A Really Good  
Film")
```



```
private String film;  
}
```

Solicitar reseñas del usuario

- [ApiReviewController.java](#)

Primero un usuario se tiene que autenticar con el API y recibir el token JWT. Una vez que lo tenemos y usando el utilidad de [httpie-jwt-auth](https://github.com/teracyhq/httpie-jwt-auth)(https://github.com/teracyhq/httpie-jwt-auth) podemos solicitar todas las reseñas del usuario ya autenticado así:

```
http --auth-type=jwt --  
auth="eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0b2tpb3NjaG9vbCI6ImV4cCI6MTY0OTcwNjE0NSwiaWF0IjoxNjQ5NzA1MDY1fQ.7yDIMR77-Wmr0_1G3ZTNpeGrxeMnJQ9LfA0hMDqeyDs"  
http://localhost:8080/api/review/user/tokioschool
```

La solicitud esta enviado al metodo `getUserReviews(@PathVariable String username, Authentication auth)` del `ApiReviewController`. Usamos el nombre del usuario como path variable para que los administradores puedan acceder a los datos tambien.

```
@GetMapping(path = "/user/{username}")  
public ResponseEntity<?> getUserReviews(@PathVariable String username,  
Authentication auth) {  
    if (!username.equals(auth.getName())) {  
        if (!auth.getAuthorities().contains(new  
SimpleGrantedAuthority("ADMIN"))) {  
            throw new ResponseStatusException(HttpStatus.UNAUTHORIZED,  
"Unauthorized");  
        }  
    }  
    Set<Review> userReviews = reviewService.findByUsername(username);  
    return ResponseEntity.ok(convertToDtos(userReviews));  
}
```

El método tiene 4 pasos:

1. Verifica que el usuario autenticado es o tiene el role de ADMIN o el usuario de que se esta pidiendo las reseñas, si no se tira una excepción.
2. Llama el método `findByUsername(String username)` del `Review Service` para cargar las reseñas del usuario en un `Set`.
3. Convierte los datos cargados en objetos de DTO.
4. Devuelve un `ResponseEntity` con el estatus HTTP de 200 y las reseñas del usuario en el cuerpo de la respuesta.

La respuesta recibida parece así:

```

HTTP/1.1 200
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Connection: keep-alive
Content-Type: application/json
Date: Mon, 11 Apr 2022 19:35:51 GMT
Expires: 0
Keep-Alive: timeout=60
Pragma: no-cache
Set-Cookie: JSESSIONID=FD7C37FBC295BE986A7050B6EEF1159C; Path=/; HttpOnly
Transfer-Encoding: chunked
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block

[
  {
    "date": null,
    "film": "Kingdom of Heaven",
    "text": "Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Donec vulputate facilisis velit eget feugiat. Vivamus nunc felis, interdum
accumsan sapien sed, feugiat scelerisque ante. Pellentesque fermentum metus
ac venenatis tristique. Ut feugiat ex hendrerit, blandit turpis id, semper
magna. Suspendisse velit libero, fermentum at dolor a, rhoncus interdum
massa. Ut mattis eget arcu at commodo. Integer rutrum diam est, in ornare
ex malesuada eleifend. Phasellus eleifend suscipit enim, ac rhoncus diam
tristique a. Sed pharetra quam quis dolor sagittis posuere. Vestibulum ante
ipsum primis in faucibus orci luctus et ultrices posuere cubilia curae;
Mauris tincidunt bibendum quam, eu porta mauris tincidunt eu. Nullam a ante
magna. Donec lacinia ipsum lacus, sit amet malesuada nulla facilisis ut.
Nullam euismod rhoncus erat. Aliquam nec ligula ac magna volutpat
ultrices.",
    "title": "Test Review",
    "user": "tokioschool"
  }
]
```

Creando una reseña nueva

Para crear una reseña nueva se tiene que enviar los datos requeridos en otro solitudine HTTP, también con un token JWT. Por ejemplo:

```

http --auth-type=jwt --
auth="eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0b2tpb3NjaG9vbCI6ImV4cCI6MTY0OTcwODIw
MiwiaWF0IjoxNjQ5NzA3MTIyYQ.v2HJxItEScF9BgGONao2yvFHg4eeqyR9lBoB8zfWxE"
http://localhost:8080/api/review/new \
title=Really\ Great\ Film \
text=.....
..... \
date=2022-04-11 \
user=tokioschool \
film=The\ Fellowship\ of\ The\ Ring
```

Los datos de la reseña nueva se envían al método `addReview(ReviewDTO reviewDTO, Principal principal)` del `ReviewController`.

```
@PostMapping(path = "/new", consumes = "application/json", produces =
{"application/json", "text/xml"})
public ResponseEntity<?> addReview(@RequestBody ReviewDTO reviewDTO,
Principal principal) {
    if (!reviewDTO.getUser().equals(principal.getName())) {
        throw new ResponseStatusException(HttpStatus.UNAUTHORIZED,
"Unauthorized");
    }
    Review received = convertToEntity(reviewDTO);
    Review added = reviewService.addReview(received);
    ReviewDTO addedDTO = convertToDto(added);
    return ResponseEntity.status(HttpStatus.CREATED).body(addedDTO);
}
```

Este método tiene 5 pasos:

1. Primero verifica que el nombre de usuario incluido en la reseña enviado es los mismo del usuario ya autenticado, si no tira una excepción.
2. Convierte la reseña (en forma `ReviewDTO`) enviado en un objeto de tipo `Review`.
3. LLama el método `addReview(Review review)` para guardarlo en el base de datos.
4. Convierte el `Review` devuelto por el `ReviewService` en `ReviewDTO`.
5. Devuelve un `ResponseEntity` con el estatus HTTP de 201 y la reseña ya añadida en su cuerpo.

En este momento es importante darse cuenta que en el método `convertToEntity(ReviewDTO reviewDto)` el `ReviewController` llama al `UserService` y `FilmService` para cargar el objeto `Review` que crea con el `User` y `Film` a cuales pertenece.

```
private Review convertToEntity(ReviewDTO reviewDTO) {
    Review review = modelMapper.map(reviewDTO, Review.class);
    review.setTextReview(reviewDTO.getText());
    review.setUser(userService.getUser(reviewDTO.getUser()));
    review.setFilm(filmService.findByTitleExact(reviewDTO.getFilm()));
    return review;
}
```

Cuando esta recibida la respuesta parece asi:

```
HTTP/1.1 201
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Connection: keep-alive
Content-Type: application/json
Date: Mon, 11 Apr 2022 20:01:17 GMT
Expires: 0
```

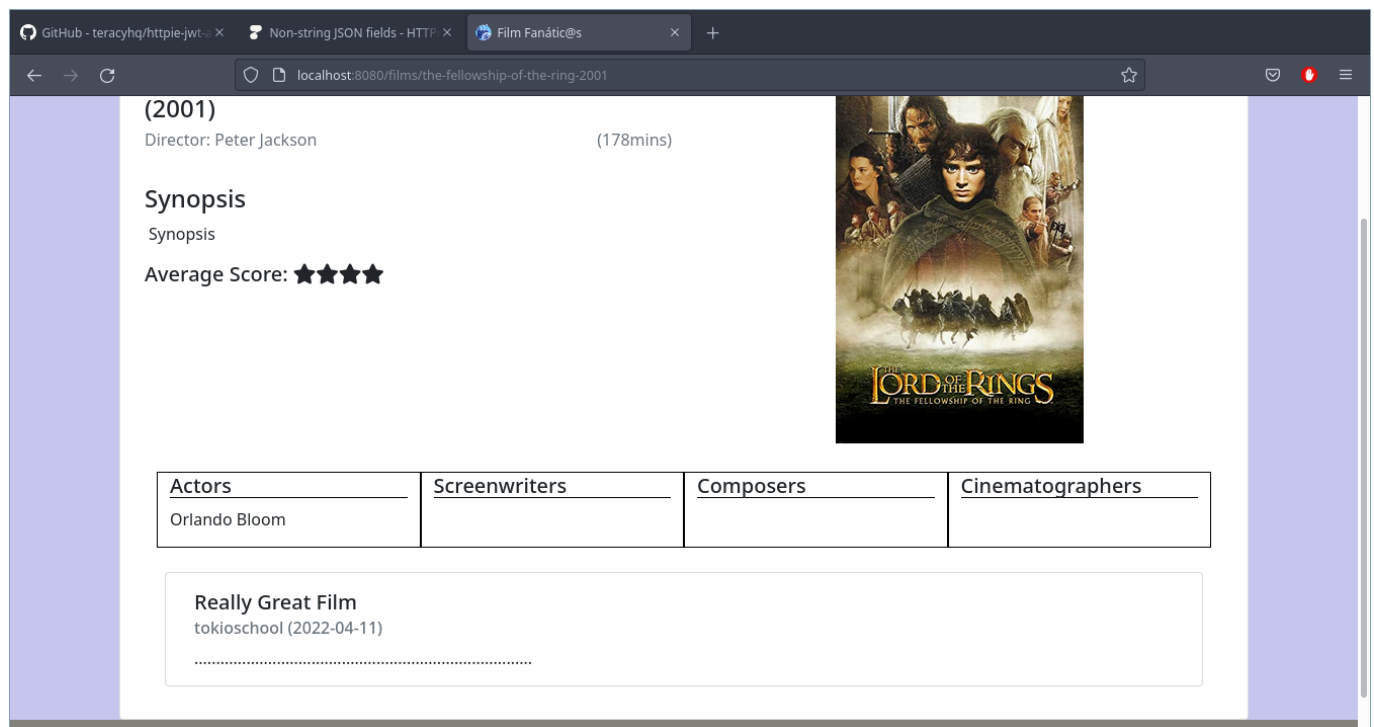
```

Keep-Alive: timeout=60
Pragma: no-cache
Set-Cookie: JSESSIONID=74DB1997C07EA76B4580D84A6E1766DE; Path=/; HttpOnly
Transfer-Encoding: chunked
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block

{
  "date": "2022-04-11",
  "film": "The Fellowship of the Ring",
  "text":
  ".....",
  "title": "Really Great Film",
  "user": "tokioschool"
}

```

Se puede ver la reseña nueva en la pagina de la pelicula también.



ApiIntegrationTest

- [ApiIntegrationTest.java](#)

Para confirmar que el API funciona bien, he creado una prueba de tipo integration para probar la autenticación del usuario y la creación de una reseña nueva. Usé uno datos de test cargados del fichero `test.sql` y la librería `RestAssured` para hacerlo. La prueba primero hace una solicitud de autenticación y luego, cuando tiene el token JWT, intenta crea una reseña nueva con el.

ReviewService

- [ReviewServiceImpl.java](#)

findByUsername(String username)

El método para cargar los **Review** que pertenecen a un usuario es bastante simple:

```
@Override
@Transactional
public Set<Review> findByUsername(String username) {
    return reviewRepo.findByUserUsername(username);
}
```

addReview(Review review)

Sin embargo el método para crear un **Review** nuevo es mas complicado.

```
@Override
@Transactional
public Review addReview(Review review) {
    User reviewingUser = review.getUser();
    Film reviewedFilm = review.getFilm();

    if (reviewingUser.getReviews().stream().anyMatch(r ->
r.getFilm().equals(reviewedFilm))) {
        throw new ReviewAlreadyExistsException(reviewingUser.getUsername(),
reviewedFilm.getTitle());
    }
    else {

        review.setUser(reviewingUser);
        review.setFilm(reviewedFilm);
        return reviewRepo.save(review);
    }
}
```

Este método tiene 4 pasos:

1. Carga el usuario que esta creando la reseña nueva y la pelicula de que es la reseña que se esta creando.
2. Verifica que el usuario ya no ha dado una reseña a la pelicula en cuestión, si lo ha dado antes se tira una excepción.
3. Establece el **Film** y **User** del **Review** nuevo.
4. Guarda y devuelve el **Review** ya creado.

Seguridad

- [WebSecurityConfig.java](#)
- [UserDetailsServiceImpl.java](#)
- [SuccessfulAuthenticationEventListener.java](#)

Como todos los programas, la seguridad de nuestra plataforma es esencial. Nuestros usuarios deben que saber que sus datos están seguros, y que la información que comparten están seguros también. Como hemos visto [antes](#) - el acceso a la pagina web es configurado por el [WebSecurityConfig](#). Los usuarios envían sus datos de autenticación con un formulario y sus sesión esta mantenido por un cookie.

SuccessfulAuthenticationEventListener

El objeto [User](#) tiene un campo llamado [lastLogin](#) para guardar la ultima fecha y hora que un usuario ha accedido a la plataforma. Para proveer esta funcionalidad he usado una clase que implementa el interface [ApplicationListener<AuthenticationSuccessEvent>](#). Asi que se activa cada vez que un usuario se autentica.

```
@Override
public void onApplicationEvent(AuthenticationSuccessEvent event) {
    User user = (User) event.getAuthentication().getPrincipal();
    log.info("User {} successfully logged in", user.getUsername());
    userService.userLoggedIn(user);
}
```

Carga el [User](#) ya autenticado del [AuthenticationSuccessEvent](#) y lo pasa al método [userLoggedIn\(User user\)](#) de [UserService](#).

```
@Override
@Transactional
public void userLoggedIn(User user) {
    LocalDateTime now = LocalDateTime.now();
    User loggedIn =
    userRepo.findByUsername(user.getUsername()).orElseThrow(UserNotFoundException::new);
    loggedIn.setLastLogin(now);
    userRepo.save(loggedIn);
}
```

Este método carga el [User](#), actualiza su dato de la ultima autenticación y lo guarda en el base de datos.

Seguridad JWT

- [JwtRequestFilter.java](#)
- [JwtTokenUtil.java](#)
- [JwtAuthenticationEntryPoint.java](#)

Aparte de la seguridad de la pagina web, también hay que segurar el acceso al REST API. Por eso he usado el modo de autenticación de Json Web Token (JWT).

JwtRequest y JwtResponse

- [JwtRequest.java](#)

- [JwtResponse.java](#)

Usamos los POJOs de `JwtRequest` y `JwtResponse` para enviar los datos de autenticación enter los usuarios y el servidor.

El `JwtRequest` contiene solo el `username` y el `password` del usuario.

```
public class JwtRequest {  
  
    private String username;  
    private String password;  
  
}
```

El `JwtResponse` contiene el token JWT que el usuario puede usar después para autenticar sus solicitudes siguientes.

```
public class JwtResponse {  
  
    @Schema(description = "Authentication Token")  
    private final String token;  
  
}
```

JwtRequestFilter

- [WebSecurityConfig.java](#)
- [JwtRequestFilter.java](#)

Configuramos la autenticación JWT en el método `configure(HttpSecurity http)` del `WebSecurityConfig`. Mas importante es la adición del `JWTRequestFilter` al `FilterChain` por cual pasan todas las solicitudes al servidor:

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .csrf()  
        .disable()  
        .authorizeRequests()  
            .antMatchers("/").permitAll()  
            .antMatchers("/register").permitAll()  
            .antMatchers("/login").permitAll()  
            .antMatchers("/film/**").permitAll()  
            .antMatchers("/films/**").permitAll()  
            .antMatchers("/search").permitAll()  
            .antMatchers("/h2-console/**").permitAll()  
            .antMatchers("/v3/api-docs/**", "/swagger-ui/**", "swagger -
```

```

ui.html").permitAll()
    .antMatchers("/admin/**").hasAnyAuthority(ADMIN_ROLE)
    .anyRequest().authenticated()
.and()
    .exceptionHandling()
    .authenticationEntryPoint(jwtAuthenticationEntryPoint)
.and()
    .sessionManagement()
    .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
.and()
    .formLogin()
    .loginPage(LOGIN_URL)
    .loginProcessingUrl(LOGIN_URL)
    .successForwardUrl(LOGIN_SUCCESS_URL)
    .failureUrl(LOGIN_FAILURE)
.and()
    .logout()
    .logoutRequestMatcher(new AntPathRequestMatcher(LOGOUT_URL))
    .logoutSuccessUrl(LOGOUT_SUCCESS_URL)
.and()
    .addFilterBefore(jwtRequestFilter,
UsernamePasswordAuthenticationFilter.class)
    // in order to allow access to h2-console for testing
    // TODO remove before release
    .headers().frameOptions().disable();

```

El `JWTRequestFilter` extiende el `OncePerRequestFilter` y solo tiene un metodo:

```

@Override
protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain chain)
    throws ServletException, IOException {

    final String requestTokenHeader = request.getHeader("Authorization");
    if (requestTokenHeader != null) {
        if (requestTokenHeader.startsWith("Bearer ")) {
            log.info("Authenticating JWT authentication request");
            String jwtToken = requestTokenHeader.substring(7);
            try {
                String username = jwtTokenUtil.getUsername(jwtToken);
                if (!username.isEmpty()
                    && null ==
SecurityContextHolder.getContext().getAuthentication()) {
                    log.info("Authenticating User {} JWT auth request", username);
                    UserDetails userDetails =
jwtUserDetailsService.loadUserByUsername(username);
                    if (jwtTokenUtil.validateToken(jwtToken, userDetails)) {
                        UsernamePasswordAuthenticationToken
usernamePasswordAuthenticationToken =
                            new UsernamePasswordAuthenticationToken(
                                userDetails, null, userDetails.getAuthorities());
                        usernamePasswordAuthenticationToken

```



```
        .setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
        SecurityContextHolder.getContext()
        .setAuthentication(usernamePasswordAuthenticationToken);
    }
}
} catch (IllegalArgumentException e) {
    logger.error("Unable to fetch JWT Token");
} catch (ExpiredJwtException e) {
    logger.error("JWT Token is expired");
} catch (Exception e) {
    logger.error(e.getMessage());
}
} else {
    logger.warn("JWT Token does not begin with Bearer String");
}
}
chain.doFilter(request, response);
}
```

Este método tiene 9 pasos:

1. Verifica si la solicitud tiene un header de nombre **Authorization** y que empiece con el **String** "Bearer", si no pasa la solicitud al proximo filtro en la cadena.
2. Saca el token JWT del solicitud.
3. Usa el objeto **JwtTokenUtil** para sacar el nombre del usuario del token.
4. Si el username existe y no hay una autenticación que ya existe en el **SecurityContext**, carga el objeto **UserDetails** del usuario por el **jwtUserDetails**.
5. Llama el método **validateToken(String jwtToken, UserDetails userDetails)** de **JwtTokenUtil** para verificar el token JWT.
6. Crea un objeto **UsernamePasswordAuthenticationToken** con el **userDetails** ya cargado.
7. Usa un nuevo objeto **WebAuthenticationDetailsSource** para establecer los datos del **UsernamePasswordAuthenticationToken**.
8. Establece el **Authentication** del **SecurityContext** con el **UsernamePasswordAuthenticationToken** ya creado.
9. Pasa la solicitud al cadena de filtros para seguir con el procedimiento.

Proceso de autenticación

- [ApiAuthenticationController.java](#)

El proceso de autenticación al REST API va así. Usando **httpie** para conectar con el servidor podemos enviar una solicitud de autenticación así:

```
http POST http://localhost:8080/api/auth -a username=tokioschool
password=tokioschool
```

El servidor interpreta el `username` y `password` enviados y los transforma en un objeto de `JwtRequest`, lo cual se envía al método `login(@RequestBody JwtRequest authRequest)` del `ApiAuthenticationController`.

```
@PostMapping(path = "/auth", consumes = "application/json", produces =
"application/json")
public ResponseEntity<?> login(@RequestBody JwtRequest authRequest)
throws Exception
{
    log.info("BEGIN login for username: {}", authRequest.getUsername());
    authenticate(authRequest.getUsername(), authRequest.getPassword());
    final UserDetails userDetails =
userDetailsService.loadUserByUsername(authRequest.getUsername());
    final String jwtToken = jwtTokenUtil.generateToken(userDetails);
    log.info("SUCCESS login:: token - {}", jwtToken);
    return ResponseEntity.ok(new JwtResponse(jwtToken));
}
```

Este método tiene 4 pasos:

1. Autentica el usuario con el `username` y `password` enviados.
2. Carga el `UserDetails` del usuario.
3. Genera el token JWT para el usuario ya autenticado con el `JwtTokenUtil`.
4. Devuelve un `ResponseEntity` con el estatus http 200 y el token JWT en el cuerpo.

El método `authenticate(String username, String password)` de `ApiAuthenticationController` es así:

```
private void authenticate(String username, String password) throws
Exception
{
    try {
        UsernamePasswordAuthenticationToken authenticationToken = new
UsernamePasswordAuthenticationToken(username, password);
        authenticationManager.authenticate(authenticationToken);
    } catch (DisabledException de) {
        throw new RuntimeException("User disabled", de);
    } catch (BadCredentialsException bce) {
        throw new UnauthorizedException("Bad credentials", bce);
    }
}
```

El método tiene 2 pasos:

1. Crea un objeto `UsernamePasswordAuthenticationToken` con el `username` y `password` dados.
2. Intenta autenticar el usuario con una llamada al `AuthenticationManager` que el controlador tiene como dependencia.

Cuando recibamos una respuesta, se parece así con el token JWT en el cuerpo:

```
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Mon, 11 Apr 2022 19:24:25 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

{
  "token":
  "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0b2tpb3NjaG9vbCI6ImV4cCI6MTY0OTcwNjE0NSwiaWF0IjoxNjQ5NzA1MDY1fQ.7yDIMR77-Wmr0_1G3ZTNpeGrxeMnJQ9LfA0hMDqeyDs"
}
```

Set-Cookie: JSESSIONID=...

Se nota que aunque el modo de autenticación del RESTful API es con el token JWT y entonces debe ser sin estado (stateless) las respuestas siempre contienen un Header con un **Cookie** de la sesión. Esto he intentado evitar pero la única manera que he encontrado es crear dos clases de **@Configuration** que extienden el **WebSecurityConfigurerAdapter** pero, aunque lo intenté no podía lograr que las dos configuraciones se aplicaran al mismo tiempo - uno a las solicitudes de REST y el otro a las solicitudes de la página web.

Proceso Batch

Como está descrito en la descripción del proyecto, he creado un pequeño proceso batch para demostrar las habilidades del *framework* Spring de procesar muchos datos de manera regular.

MigrateJobConfig

- [MigrateJobConfig.java](#)

Todo el trabajo de procesamiento está configurado en el **MigrateJobConfig**. Aquí se declaran todos los objetos que van a formar parte del trabajo, los pasos (o en este caso el único paso) del trabajo, y el trabajo en sí.

reader()

El **reader** es el objeto que lee los datos que se van a procesar. En este caso lee los datos de las películas de la base de datos, así que es de tipo **JdbcCursorItemReader<Film>**:

```
@Bean
public JdbcCursorItemReader<Film> reader() {
    log.info("Preparing reader...");
    return new JdbcCursorItemReaderBuilder<>()
        .name("filmReader")
        .dataSource(dataSource)
```

```
.sql("SELECT * FROM films")
.rowMapper(new FilmMapper())
.build();
}
```

FilmMapper

- [FilmMapper.java](#)

El método `.rowMapper()` define el objeto que convierte los datos leídos en un objeto de JAVA. Para hacer que los datos leídos se convierten en objetos de tipo `Film` declaré la clase `FilmMapper` que extiende la clase `RowMapper` y usa un objeto `BeanPropertyRowMapper` para crear el objeto `Film` que queremos. Para cargar los datos de los objetos miembros era necesario coger los datos disponibles directamente del `ResultSetMetaData`. No pude cargar todos los datos de los objetos que tiene relaciones `@ManyToMany` con el `Film`.

```
public Film mapRow(ResultSet rs, int rowNum) throws SQLException {
    Film film = (new BeanPropertyRowMapper<>(Film.class)).mapRow(rs,
        rowNum));

    ResultSetMetaData metaData = rs.getMetaData();

    User user = new User();
    user.setId(rs.getLong("user_id"));
    Person director = new Person();
    director.setId(rs.getLong("director_id"));

    film.setUser(user);
    film.setFilmDirector(director);

    return film;
}
```

FilmItemProcessor

- [FilmItemProcessor.java](#)

Después de esta leído y transformado en objetos de JAVA, los datos tiene que ser procesados. Para esto definí una clase `FilmItemProcessor` que simplemente verifica si la película ha sido migrado ya o no, si no se la pasa al proximo paso del procesamiento.

```
@Override
public Film process(Film film) throws Exception {
    if (!film.getMigrate()) {
        return null;
    }
    else return film;
}
```

writer()

Después de su procesamiento, los datos están escritos al fichero de tipo **.csv**. Para esto definí un objeto de tipo **FlatFileTimeWriter<Film>**:

```
@Bean
public FlatFileItemWriter<Film> writer() {
    log.info("Preparing writer...");
    return new FlatFileItemWriterBuilder<Film>()
        .name("filmWriter")
        .resource(new ClassPathResource("films.csv"))
        .lineAggregator(new FilmLineAggregator())
        .build();
}
```

FilmLineAggregator

- [FilmLineAggregator.java](#)

El **FilmLineAggregator** es el objeto que convierte los objetos de JAVA en un **String** para escribir al fichero ya definido. Como el fichero es de **.csv** (*comma separated values*) simplemente construye un **String** con todos los datos de la película separados por comas.

```
public class FilmLineAggregator implements LineAggregator<Film> {

    private static final String CSV_DELIMITER = ",";

    @Override
    public String aggregate(Film film) {
        StringBuilder builder = new StringBuilder();
        builder.append(film.getId());
        builder.append(CSV_DELIMITER);
        builder.append(film.getTitle());
        builder.append(CSV_DELIMITER);
        builder.append(film.getYear());
        builder.append(CSV_DELIMITER);
        builder.append(film.getDuration());
        builder.append(CSV_DELIMITER);
        builder.append(film.getSynopsis());
        builder.append(CSV_DELIMITER);
        builder.append(film.getPoster());
        builder.append(CSV_DELIMITER);
        builder.append(film.getFilmDirector().getId());
        builder.append(CSV_DELIMITER);
        builder.append(film.getAvgScore());
        builder.append(CSV_DELIMITER);
        builder.append(film.getUser().getId());
    }
}
```

```
    return builder.toString();
}
```

step()

Luego definimos el paso que incluye la lectura por el **reader**, el procesamiento por el **FilmItemProcessor**, y la escritura de los datos ya procesados por el **writer** y el tamaño de cada paso (10 objetos). También definimos un **listener** que actúa en reacción al cumplimiento de cada paso.

```
@Bean
public Step step(FlatFileItemWriter<Film> writer, MigrateFilmWriteListener
writeListener) {
    log.info("Preparing step...");
    return stepBuilderFactory.get("step")
        .<Film, Film>chunk(10)
        .reader(reader())
        .processor(processor())
        .writer(writer)
        .listener(writeListener)
        .build();
}
```

MigrateFilmWriteListener

- [MigrateFilmWriteListener.java](#)

El **MigrateFilmWriteListener** activa después del procesamiento de cada 10 objetos **Film** para actualizar el base de datos con la información que ya han sido migradas. Usa el método **batchUpdate** del **JdbcTemplate** con un **BatchPreparedStatementSetter** para actualizar los datos de la películas que han sido migradas.

```
@Override
public void afterWrite(List<? extends Film> migratedFilms) {
    log.info("Updating migrated films in db");

    jdbcTemplate.batchUpdate(
        "UPDATE films SET migrate=?, date_migrate=? WHERE id=?",
        new BatchPreparedStatementSetter() {
            @Override
            public void setValues(PreparedStatement ps, int i) throws
SQLException {
                ps.setBoolean(1, true);
                ps.setDate(2, Date.valueOf(LocalDate.now()));
                ps.setLong(3, migratedFilms.get(i).getId());
            }

            @Override
```

```
        public int getBatchSize() {
            return migratedFilms.size();
        }
    });
}
```

migrateFilmJob()

Finalmente declaramos el método `migrateFilmJob()` que devuelve el `Job` que se va a ejecutar. Este contiene el `Step` o paso ya definido y otro `listener` que se activa al empiezao del trabajo.

```
@Bean
public Job migrateFilmJob(MigrateFilmStartListener startListener, Step
step) {
    return jobBuilderFactory.get("migrateFimJob")
        .incrementer(new RunIdIncrementer())
        .listener(startListener)
        .flow(step)
        .end()
        .build();
}
```

MigrateFilmListener

- [MigrateFilmStartListener.java](#)

Al empiezao del trabajo de procesamiento el `MigrateFilmStartListener` activa para contar las películas que todavía no han sido migradas y escribir lo a los `logs`.

```
@Override
public void beforeJob(JobExecution jobExecution) {
    log.info("Migrate job starting...");

    jdbcTemplate.query("SELECT COUNT (*) FROM films WHERE migrate = 0", (rs,
    rw) -> rs.getInt(1))
        .forEach(count -> log.info("{} Films to migrate", count));
}
```

Excepciones, Depuración y *logging*

La gestión de errores y *logging* de eventos en el programa son elementos importantes para el buen funcionamiento del programa. Aquí tratamos de los dos.

Gestión de excepciones

- [excepciones](#)

Para manejar excepciones que surgen en el programa decidí usar dos clases que usan la anotación `@ControllerAdvice`. Uno gestiona todas las excepciones de la página web, y el otro las del REST API. Gestionan todas las excepciones ya declaradas en el base de código y usado en el programa, y también gestiona cualquier otra inesperada que surgen.

WebsiteExceptionHandlerAdvice

- [WebsiteExceptionHandlerAdvice.java](#)
- [error.html](#)

```
@ControllerAdvice(basePackages =
{"io.chilborne.filmfanatic.controller.thymeleaf"})
public class WebsiteExceptionHandlerAdvice {

    private final Logger logger =
LoggerFactory.getLogger(WebsiteExceptionHandlerAdvice.class);

    @ExceptionHandler
    public ModelAndView handleException(HttpServletRequest request, Exception
exception){
        logger.error("Exception Caught!", exception);
        exception.printStackTrace();
        ModelAndView mav = new ModelAndView();
        mav.addObject("message", exception.getMessage());
        mav.addObject("exception", exception);
        mav.setViewName("error");
        return mav;
    }

}
```

Este método depura el error y luego devuelve un objeto `ModelAndView` llamada "error" cargada con el mensaje y excepción. Spring luego comparte la página `error.html` cargada con el `ModelAndView`. La página muestra los detalles relevantes del error al usuario.

```
<div class="container">
    <div class="alert alert-danger" role="alert">
        <p><strong>Whoops! Something went wrong.</strong></p>
        <p th:text="${message}"></p>
        <p><a href="#" class="alert-link" onclick="history.back()">Go
Back</a></p>
    </div>
</div>
```

ApiExceptionHandlerAdvice

- [ApiExceptionHandlerAdvice.java](#)

- [ErrorResponse.java](#)

El `ApiExceptionHandlerAdvice` usa un POJO llamado `ErrorResponse` para enviar los datos de los errores:

```
@Data
public class ErrorResponse {

    @Schema(description = "Time error was raised", example = "2021-04-04T12:42:05")
    private String timestamp;
    @Schema(description = "HTTP Status code returned", example = "400")
    private int status;
    @Schema(description = "Brief description of the error", example = "Bad Request")
    private String error;
    @Schema(description = "Message from server with information regarding the error", example = "This is an error message.")
    private String message;
    @Schema(description = "Path of request which raised the error", example = "/path/of/request")
    private String path;
```

Tiene un método privado para construir el objeto `ErrorResponse` con los datos del error:

```
private ResponseEntity<ErrorResponse> buildResponse(HttpStatus status,
Exception exception, HttpServletRequest request) {
    return ResponseEntity.status(status).body(
        new ErrorResponse(LocalDate.now(), status, exception.getMessage(),
request.getContextPath())
    );
}
```

Y varios métodos para gestionar errores diferentes. Por ejemplo el error de tipo `UnauthorizedException` tiene el siguiente método:

```
@ExceptionHandler(UnauthorizedException.class)
public ResponseEntity<ErrorResponse>
handleUnauthorizedException(HttpServletRequest request,
UnauthorizedException ex) {
    return buildResponse(HttpStatus.UNAUTHORIZED, ex, request);
}
```

Depuración y logging

- [logback-spring.xml](#)

Usando la librería *LogBack* que es el que viene por defecto con *Spring Boot*, escribimos trazas de varios evento, errores incluidos, a various *logs* configurados en el fichero *logback-spring.xml*. La configuración declara dos *logs* - una que se escribe directamente al console durante el ejecución del programa, y otro que se escribe a un fichero que se guarda de manera rotativo.

```
<!-- Configuración del log que aparece por consola: Console appender -->
<appender name="Console"
    class="ch.qos.logback.core.ConsoleAppender">
    <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
        <level>DEBUG</level>
    </filter>
    <!-- Configuración de la traza -->
    <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
        <Pattern>
            %white(%d{ISO8601}) %highlight(%-5level) [%blue(%t)]
            %-60.60yellow(%C{20}): %msg%n%throwable
        </Pattern>
    </encoder>
</appender>

<!-- Configuración para que se almacene el log en un fichero: File Appender
-->
<appender name="RollingFile"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>${LOG_DIR}/${LOG_NAME}.log</file>
    <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
        <level>DEBUG</level>
    </filter>
    <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
        <Pattern>%d %p %C{1.}:%M [%t] %m%n</Pattern>
    </encoder>

    <!-- Política de rotado de logs: diario y cuando el fichero llegue a
    los 10 MB -->
    <rollingPolicy
        class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>${LOG_DIR}/${LOG_NAME}-%d{yyyy-MM-dd}.%i.log</fileNamePattern>
        <timeBasedFileNamingAndTriggeringPolicy
            class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
            <maxFileSize>10MB</maxFileSize>
        </timeBasedFileNamingAndTriggeringPolicy>
    </rollingPolicy>
</appender>
```

Aprendizaje y Mejoras

Durante este proyecto he aprendido mucho y, claro, he hecho varios error. Pensándolo de nuevo mientras he estado escribiendo este documento hay unas cosas que hubiera hecho de una manera diferente.

1. No creo que era necesario implementar el `UserDetails` con mi propio objeto `User`. Creo que con la funcionalidad de caching implementado se podría evitar demasiadas llamadas al base de datos sin complicar el programa. Además la implementación de caching ayudaría en otras áreas del programa, por ejemplo cargando el formulario para crear una película nueva con todas las personas guardadas cada vez.
2. El uso del `film-uri` en los `url` de la página web era también innecesario. Se podría usar el `id` de la película sin problema, muchas plataformas grandes lo hacen así. Incluso si se usara el `uri` de ninguna manera sea necesario convertir del título al uri y al revés cada vez que se tiene que cargar una película del base de datos. Podría usar el `uri` como otro índice en el base de datos.
3. No me gusta que las respuestas de API también incluyan un cookie de sesión. No debería ser así y me gustaría haber podido crear dos configuraciones separadas como está explicado [aquí](#).