DP1 2020-2021

Documento de Diseño del Sistema

Proyecto Standby

https://github.com/gii-is-DP1/dp1-2020-gi-04

Miembros <en orden alfabético por apellidos>:

* Bogdan, George Laurentiu
* Diz Gil, Guillermo
* Muñoz Pérez, Carmen María
* Rodríguez Pérez, Francisco

Tutor: José María García

GRUPO GI-04

Versión 1

17/01/2021

# Historial de versiones

*Estos son ejemplo del contenido que debería tener el historial de cambios del documento a entregar a lo largo de los sprints del proyecto*

|  |  |  |  |
| --- | --- | --- | --- |
| **Fecha** | **Versión** | **Descripción de los cambios** | **Sprint** |
| 17/01/2021 | V1 | * Creación del documento | 4 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[Historial de versiones 2](#_Toc61816757)

[Introducción 5](#_Toc61816758)

[Diagrama(s) UML: 6](#_Toc61816759)

[Diagrama de Dominio/Diseño 6](#_Toc61816760)

[Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios) 7](#_Toc61816761)

[Patrones de diseño y arquitectónicos aplicados 10](#_Toc61816762)

[Patrón: Model-View-Controller 10](#_Toc61816763)

[Tipo: Design 10](#_Toc61816764)

[Contexto de Aplicación 10](#_Toc61816765)

[Clases o paquetes creados 10](#_Toc61816766)

[Ventajas alcanzadas al aplicar el patrón 11](#_Toc61816767)

[Desventajas de aplicar el patrón 11](#_Toc61816768)

[Design decisions 12](#_Toc61816769)

[Decision 1: Pick the frontend framework 12](#_Toc61816770)

[Problem description: 12](#_Toc61816771)

[Alternative evaluated solutions: 12](#_Toc61816772)

[Justified solution adopted 13](#_Toc61816773)

[Decision 2: Video thumbnail generation 13](#_Toc61816774)

[Problem description: 13](#_Toc61816775)

[Alternative evaluated solutions: 13](#_Toc61816776)

[Justified solution adopted 13](#_Toc61816777)

[Decision 3: Data relation loading 14](#_Toc61816778)

[Problem description: 14](#_Toc61816779)

[Alternative evaluated solutions: 14](#_Toc61816780)

[Justified solution adopted 14](#_Toc61816781)

[Decision 4: Initial data load 15](#_Toc61816782)

[Problem description: 15](#_Toc61816783)

[Alternative evaluated solutions: 15](#_Toc61816784)

[Justified solution adopted 15](#_Toc61816785)

[Decision 5: Security Layer 16](#_Toc61816786)

[Problem description: 16](#_Toc61816787)

[Alternative evaluated solutions: 16](#_Toc61816788)

[Justified solution adopted 16](#_Toc61816789)

[Decision 6: Controller creation 17](#_Toc61816790)

[Problem description: 17](#_Toc61816791)

[Alternative evaluated solutions: 17](#_Toc61816792)

[Justified solution adopted 17](#_Toc61816793)

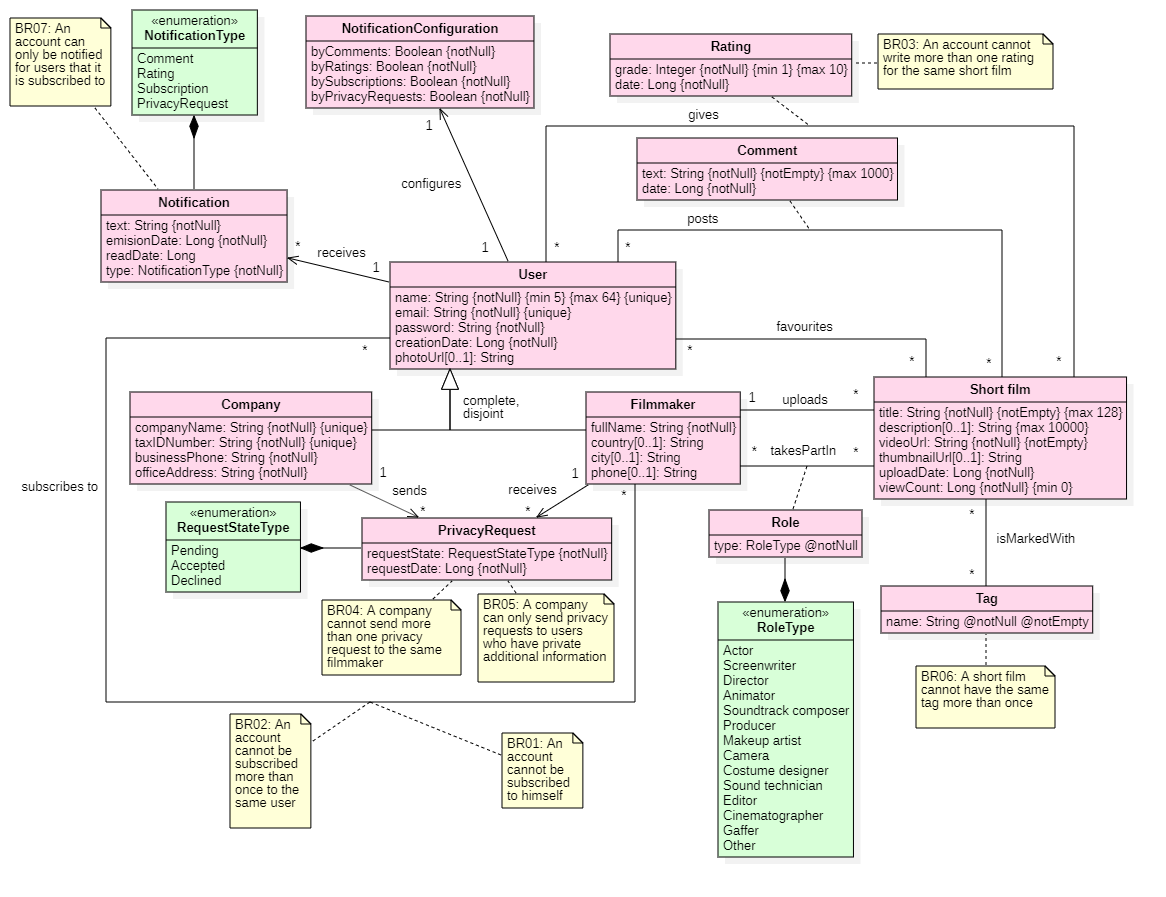
# Introduction

The short films are the first productions that are usually made by those who want to dedicate themselves to the world of cinema, but at the same time they are ignored in the cultural field within the film industry. The best-known short films are those that win very important awards or are created by very important companies such as Disney. But what about that majority that does not win awards? They always fall into oblivion. Despite all this, almost all the great filmmakers of our time have started making short films and these have been vital in their rise to fame. Thus, the Standby project was born out of the need of both amateur and professional filmmakers who want to make themselves known and make their way in the world of cinema through short films, as well as the need to promote and disseminate the importance of short films within the industry.

For all these reasons, this project aims to provide an application that brings together short films from around the world, as well as serving as a stimulus for those who want to join the film industry. Standby will be an application that will allow filmmakers to share their short films. Thus, users will be able to watch, comment and evaluate these short films. In addition, companies will be able to filter them and access additional information that will allow them to contact the filmmakers for possible hiring.

# UML Diagram(s):

## Domain/Design Diagram



# Layer Diagram (including Controllers, Services, and Repositories).

In the following, several layer diagrams have been made to show the relationships between controllers, services and repositories. Due to the large number of them, it has been divided into five different layer diagrams.

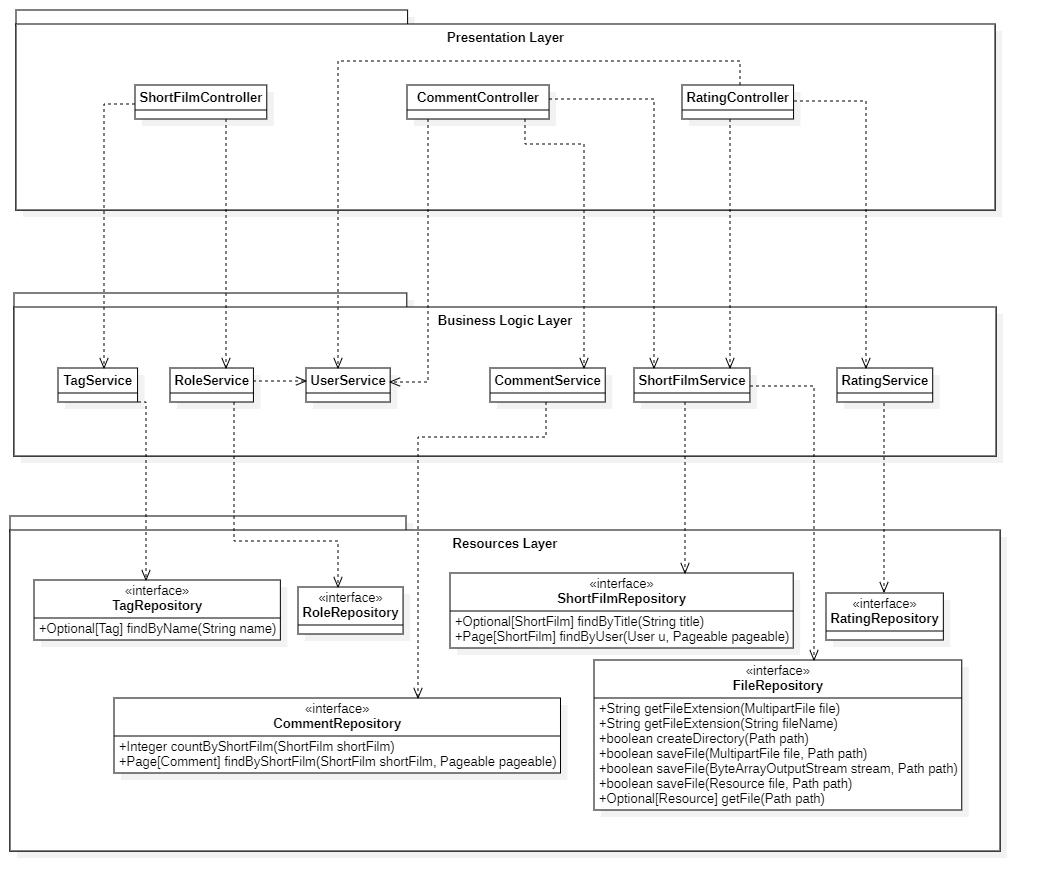
1. This first diagram shows the interaction of the controllers directly related to the users: UserController, FilmmakerController and CompanyController. Not all related services will be detailed in this diagram (ShortFilmService and SubscriptionService, as they will be detailed when the flow of their respective controllers is shown). Thus, the following services are detailed: UserService, FilmmakerService, CompanyService and NotificationConfigurationService.

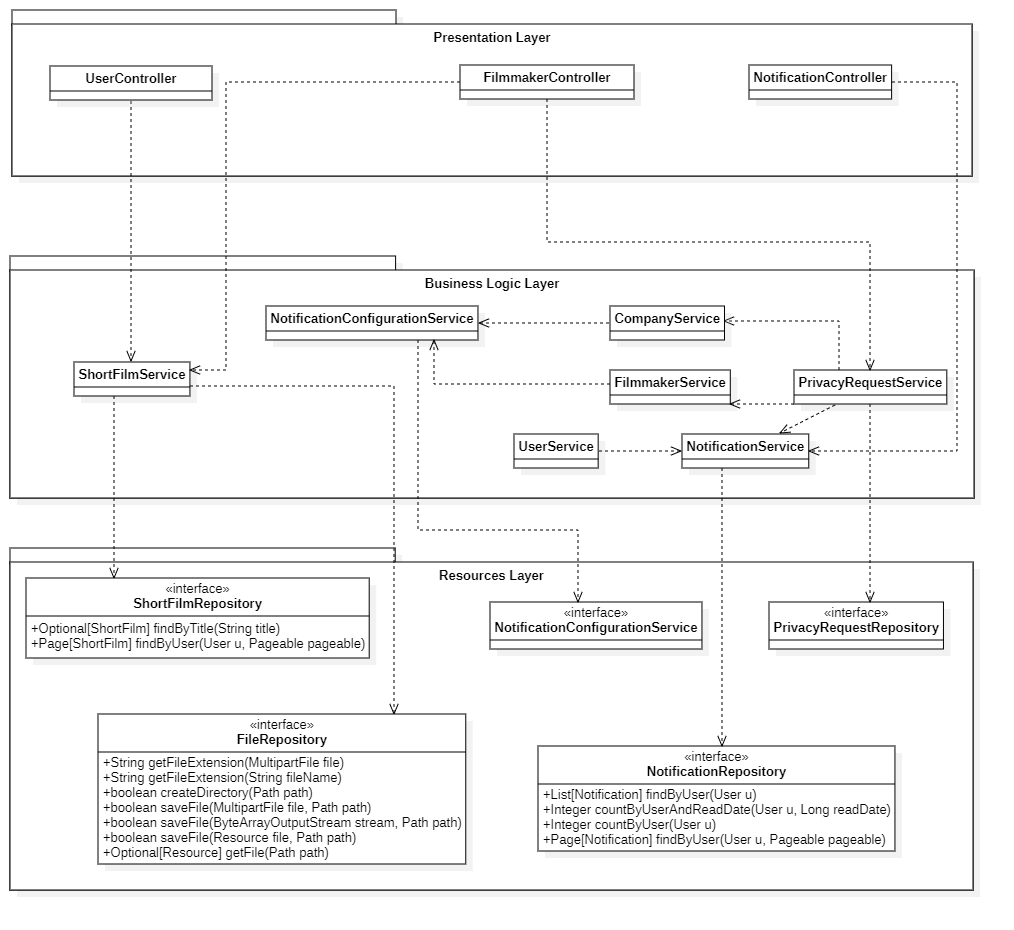
2. The second diagram shows the interaction of the controllers directly related to the user configuration: PrivacyRequestController and NotificationController. UserService will not be detailed, as it has been done in the previous diagram. Thus, the following services are detailed: PrivacyRequestService and NotificationService.

3. The third diagram shows the interaction of the controllers related to two actions that users can perform (commenting and rating): RatingController, CommentController. Not all the related services are detailed in this diagram (ShortFilmService and SubscriptionService, as they will be detailed when the flow of their respective controllers is shown). Thus, the following services are detailed: RatingService, CommentService.

4. The fourth diagram shows the interaction of the ShortFilmController. It has been put in one diagram only, as it encompasses a large number of services. Some services are not detailed because they have already been detailed and others because they will be detailed with their respective controllers. Thus, the following services are detailed: TagService, ShortFilmService and RoleService.

5. The fifth diagram shows the interaction of the controllers related to actions related to users and films: SubscriptionController, SearchController and FavouriteController. Those services that have already been detailed will not be detailed. Thus, the following services are detailed: SubscriptionService and FavouriteService.





# Design and architectural patterns applied

## Pattern: Model-View-Controller

### Type: Architectural

### Application context

Model-view-controller (we will refer to this pattern as its abbreviation “MVC”) is a software pattern commonly used for separate application’s concerns into three interconnected elements.

These three elements are:

* Model: It is the representation of the information. It includes both the data and the business logic that is necessary to work with them. Moreover, it is responsible for managing the data of the application. It receives user input from the controller.
* View: It is the presentation of the model in a way the user can interact with it. It is usually done by means of a User Interface.
* Controller: It responds to events in the user interface, invokes changes in the model in the view.

### Packages and classes created

*Model related:*

**io.github.fourfantastics.standby.model** **:** In this package all the application model classes are encapsulated{ Comment , Company, Filmmaker, Notification, Notification Configuration, Notification Type, Privacy Request, Rating, RequestStateType, Role, RoleType, ShortiFilm, Tag, User, UserType}, on this classes there are the business logic necessary to work with them (relation between the entities, attributes characteristic as @notNull etc …).

**io.github.fourfantastics.standby.model.form** **:** The classes in this package are used to display certain attributes of different models on the view , as well as , to collect the certain information from the view. This collected information will update the corresponding application model once validated.

**io.github.fourfantastics.standby.model.validator :** These validator are used to validate the attributes of the model forms who are sent by the user in the view in order to assure that no invalid Data is being sent and the parsing to Model from “model form” will be performed correctly.

**io.github.fourfantastics.standby.repository :** The repositories in this package provides methods for retrieving and saving domain objects to the database , there is one per entity.

**io.github.fourfantastics.standby.service:** The services provides the functionality of the domain as an API, we have organised them in terms of entities. Some of them perform domain-related validation but never input-domain validation (since the responsible of that validation must be the controller).

*View related:*

**dp1-2020-gi-04/src/main/resources/templates:** Represent the information given by the controllers and enables the user to interact with the application. In our case we are using Thymeleaf and bootstrap on the frontend.

*Controller related:*

**io.github.fourfantastics.standby.web:** On this package all the controller class are encapsulated they are used to manage all the petitions from the user interface (call the corresponding services, validate the input Data, redirecting to another view, etc).

### Benefits achieved by using this pattern

1. MVC offers support for rapid and parallel development. This way, developing web applications using the MVC pattern is possible that one developer work on the view while the another can work on the controller. We found very advantageous the fact that we can easily parallelise the work, so we were able to develop the application faster.
2. Ease to find errors, since the MVC pattern helps the application to separate its concerns we find it quite easy to find an error since we can easily know where the error is.
3. Modifications have few impacts, is obvious that you make frequent changes in your web application like changing colours, fonts, screen layouts, and adding new device support for mobile phones or tablets. Moreover, adding a new type of view are very easy in the MVC pattern because the Model part does not depend on the views part. Therefore, any changes in the Model will not affect the entire architecture

### Downsides of applying this pattern

1. Although there exists a clear separation of concerns the view and the model have certain relations which can lead to problems, e.g.: If the model undergoes frequent changes, the views could be flooded with update requests and outdated.

## Pattern: Front-Controller

### Type: Design

### Application context

This pattern consists of creating a Front-controller that handles all requests from the frontend and then dispatches them to their appropriate controller. In the framework used on this project (Spring) provide us the “Dispatcher servlet” in order to apply this pattern.

### Packages and classes created

None.

### Benefits achieved by using this pattern

1. Centralize control.
2. The implementation of this controller is entirely provided by Spring.
3. The front controller can provide additional functionality such as request processing for parameter validation and transformation.

### Downsides of applying this pattern

1. None significant.

## Pattern: Domain model

The conceptual model represents the entities of a system and the relations between them.

### Type: Design

### Application context

This pattern consists of a conceptual model which incorporates data and behaviour (logic).

### Packages and classes created

**io.github.fourfantastics.standby.model** **:** In this package all the application model classes are encapsulated{ Comment , Company, Filmmaker, Notification, Notification Configuration, Notification Type, Privacy Request, Rating, RequestStateType, Role, RoleType, ShortiFilm, Tag, User, UserType}, on this classes there are the business logic necessary to work with them (relation between the entities, attributes characteristic as @notNull etc …).

### Benefits achieved by using this pattern

1. Allow us to implement complex logic in an easy way
2. Spring and many other frameworks supports this pattern.

### Downsides of applying this pattern

1. None, since the usual downside of this pattern (hard to map in the database) is solved by using Spring who make it easy.

## Pattern: Service Layer

### Type: Design

### Application context

Defines an application’s boundary with a layer of services that establishes a set of available operations and coordinates the application’s response in each operation.

### Packages and classes created

**io.github.fourfantastics.standby.service:** The services provides the functionality of the domain as an API, we have organised them in terms of entities. Some of them perform domain-related validation but never input-domain validation (since the responsible of that validation must be the controller).

**io.github.fourfantastics.standby.web:** On this package all the controller class are encapsulated they are used to manage all the petitions from the user interface (call the corresponding services, validate the input Data, redirecting to another view, etc).

These two packages contains almost all the application logic of the system.

### Benefits achieved by using this pattern

1. By applying this pattern, we split the business logic in two parts: “Domain Logic” and “Application logic” this facilitates the low coupling and high cohesion.
2. By separating the data access model and the service you are able to change the data access level technology without having to change the services.

### Downsides of applying this pattern

1. For those entities who only need simple logic (CRUD operations) creating a service might be unnecessary we will only have function calling the CRUD repository functions.

## Pattern: Data Mapper

### Type: Architectural

### Application context

A Data Mapper is a Data access layer that performs bidirectional transfer of data between a persistent data store and an in-memory data representation (the domain layer). We perform this pattern thanks to Spring Data and Spring framework.

### Packages and classes created

**io.github.fourfantastics.standby.repository :** The repositories in this package provides methods for retrieving and saving domain objects to the database , there is one per entity. (we created these repositories due to this pattern and the Repository pattern).

### Benefits achieved by using this pattern

1. Incredibly easy to perform CRUD operations.
2. We are able to perform more complex logic thanks to this pattern

### Downsides of applying this pattern

1. None significant.

## Pattern: Identity Field

### Type: Design

### Application context

We applied this pattern in order to identify unequivocally each of the entities in our model domain (the type of our ID is Long).

### Packages and classes created

No classes nor packages has been created due to this pattern, the impact that this pattern had is to add one Long attribute to each entity (the ID attribute).

### Benefits achieved by using this pattern

1. It corresponds to the primary key in the data base.
2. Getting an ID (primary key) is done automatically in Spring by autoincrementing the ID.
3. All the classes have the same type of primary key.

### Downsides of applying this pattern

1. Since our ID for all the entities is a Long type, it has no meaning in the entities domains is simply a unique identifier for identifying each entity.

## Pattern: Repository Pattern

### Type: Design

### Application context

We applied this pattern as a way to communicate with the data sources, the repositories implementation is provided by Spring.

### Packages and classes created

**io.github.fourfantastics.standby.repository :** The repositories in this package provides methods for retrieving and saving domain objects to the database , there is one per entity. (we created these repositories due to this pattern and the Repository pattern).

### Benefits achieved by using this pattern

1. More intuitive interaction with the Database (It seems that you are accessing a collections of In-memory objects rather than a database).

### Downsides of applying this pattern

1. Creating one repository per Entity.

## Pattern: Dependency injection

### Type: Design

### Application context

For the mere fact of using spring we are applying this pattern since Spring is based in uses the inversion of control. By using dependency injection we extract the responsibility of creating an instances to the framework (Spring in our case).

### Packages and classes created

None.

### Benefits achieved by using this pattern

1. We ensure that only one instance is created and that only classes that need it will have access to it.
2. By using dependency injection we are applying the proxy pattern.

### Downsides of applying this pattern

1. None significant.

## Pattern: Proxy Pattern

### Type: Design

### Application context

We are applying this pattern as a consequence of applying the dependency injection principle.

### Packages and classes created

None.

### Benefits achieved by using this pattern

1. Avoid duplication of objects.
2. Provides a substitute to another object to control its access.
3. It is totally transparent.

### Downsides of applying this pattern

1. None significant.

## Pattern: Eager Loading

### Type: Design

### Application context

We applied this pattern on all the associations of our project, with this pattern all the data is initialized on the spot.

### Packages and classes created

None, just in the associations between entities.

### Benefits achieved by using this pattern

1. Not delayed performance impact, accessing the data is faster since the data is already initialized.

### Downsides of applying this pattern

1. Longer initial load time.
2. Might load a huge quantity of data.

## Pattern: Pagination

### Type: Design

### Application context

We applied this patter in the parts of our application that that might have too much data to show in a single view such as: the comment section of the short films, description of the short film, to visualize the all the notifications of a filmmaker etc.

### Packages and classes created

**Pagination.html :** a html fragment created by us in order to implement the pagination in different views in an easy wasy.

### Benefits achieved by using this pattern

1. The views doesn’t become an “infinite scrolling page” when ,for instance, a description of a short film is too long.
2. The backend doesn’t have to give enormous piece of data to the frontend.

### Downsides of applying this pattern

1. It could affect the user experience since the user will have to do more clicks in order to see the full information about a comment or description.

# Design decisions

In this section we will describe the design decisions we have made throughout the development process of the application.

## Decision 1: Pick the frontend framework

### Problem description:

We need to display the view to the user and using a frontend framework would ease the workload of writing all the front from scratch.

### Alternative evaluated solutions:

Alternative 1.a: No framework, plain HTML, CSS, JavaScript

**Advantages**:

* We have full control of the application flow.
* Less overhead for the client, because of file size

**Disadvantages**:

* More work as we must write more code
* More complexity as we must write the full system

Alternative 1.b: Use Thymeleaf as server-side rendering.

**Advantages:**

* Less overhead for the client because it is rendered in the server.
* Access to the server data
* Conditional rendering

**Disadvantages:**

* We must learn Thymeleaf
* Page is reloaded almost every time we must render.

Alternative 1.c: Use React as client-side rendering.

**Advantages:**

* Dynamic page rendering
* Reactive to changes.
* Use of existing UI libraries.

**Disadvantages:**

* We must learn React.
* More complex.

### Justified solution adopted

We have chosen the alternative 1.b Thymeleaf because it was the perfect tradeoff between complexity for the developer and user experience.

## Decision 2: Video thumbnail generation

### Problem description:

Some visual description of a short film, like a thumbnail, is attractive to the users to decide to watch a short film. However, video usually is stored as complex data; it is unlikely that a thumbnail can be retrieved easily.

### Alternative evaluated solutions:

Alternative 2.a: Use JCodec library for automatically retrieving a thumbnail at uploading stage

**Advantages**:

* It is more comfortable for the filmmakers that do not want to upload a custom thumbnail

**Disadvantages**:

* JCodec lacks documentation and no other libraries are actively supported
* Increases system complexity and size

Alternative 2.b: Do not generate an automatic thumbnail

**Advantages:**

* Easier for the development
* Keeps the project lean and mean, avoiding an external dependency

**Disadvantages:**

* Filmmakers are forced to upload a custom thumbnail in case they want their short film to have one

### Justified solution adopted

We have chosen the alternative 2.b “Do not generate an automatic thumbnail”, because JCodec was not worth in terms of complexity. Also, we thought that most filmmakers wanted to upload their custom thumbnails, as they can be more attractive than generated ones.

## Decision 3: Data relation loading

### Problem description:

In Spring Data JPA, data from entities can be retrieved by many ways. In specific, using repositories with customized methods whose implementation is injected by Spring. However, data and relations can also be retrieved, in most cases, using Java getters, but only if Spring is told to retrieve data eagerly.

### Alternative evaluated solutions:

Alternative 3.a: Use repository methods to retrieve all data from entities

**Advantages**:

* Allows to keep track of every data access when testing

**Disadvantages**:

* It is necessary to create a method for each attribute we want to access
* Repositories need to be mapped in services, and it would create unnecessary complex data access situations

Alternative 3.b: Use getters to retrieve data eagerly from entities

**Advantages:**

* It is easier for the data to be retrieved
* As Lombok is available, methods do not need to be even created manually

**Disadvantages:**

* Entity data needs to be kept in memory to be retrieved eagerly

### Justified solution adopted

We have chosen the alternative 2.b “Use getters to retrieve data eagerly from entities”. As we already use a in-memory database (H2), it does not affect the performance of the system.

## Decision 4: Initial data load

### Problem description:

Though initial data is not necessary for the system to work, it is a good way of generating a fast showcase of what the application is able to.

### Alternative evaluated solutions:

Alternative 4.a: Use data.sql file

**Advantages**:

* Allows the separation between code and initial data injection

**Disadvantages**:

* Complex SQL syntax

Alternative 4.b: Use a CommandLineRunner class to inject data just before start

**Advantages:**

* It is simpler and cleaner because it directly uses services and repositories

**Disadvantages:**

* It is not as “clean” as using a separate data file.

### Justified solution adopted

We have chosen the alternative 4.b because we think the syntax is cleaner: we preferred to use service methods rather than writing SQL directly.

## Decision 5: Security Layer

### Problem description:

We need a login system to identify and authorize users.

### Alternative evaluated solutions:

Alternative 5.a: Use Spring Security

**Advantages**:

* Manage all the security flow automatically.
* Built-in library, we do not have to program it.

**Disadvantages**:

* We must learn Spring Security
* It is harder to use for more complex cases

Alternative 5.b: Use custom security layer.

**Advantages:**

* We have full control over the security flow.
* Easier to program from the beginning.

**Disadvantages:**

* We must program it ourselves.
* Can lead to data leaks if not implemented correctly.

### Justified solution adopted

We have chosen the alternative 5.b custom layer because it was the easiest to understand and program.

## Decision 6: Controller creation

### Problem description:

We must choose how to organize the creation of controllers

### Alternative evaluated solutions:

Alternative 6.a: Create a controller for each model.

**Advantages**:

* Easy to find where a model is accessed.
* Can create controllers independently.

**Disadvantages**:

* A view is separated in many controllers

Alternative 6.b: Create a controller for each view.

**Advantages:**

* All the dependencies of the view are in one controller
* Easier to implement a new page.

**Disadvantages:**

* If two pages need the same model need to implement again the same logic
* Harder to work in parallel.

### Justified solution adopted

We have chosen the alternative 6.a controller for each model because it is easier for each member to work in a controller independently and it is easier to find the respective controller for a model.