DP1 2020-2021

Documento de Diseño del Sistema

Proyecto **PeliculasOnline**

<https://github.com/dantolvil91/dp1-2020-grupo-dicembre-2.git>

Miembros:

* Toledo Villalba, Daniel

Tutor:

* Parejo, José Antonio

# GRUPO G2-Dicimbre 2021

Versión 1.9

07/12/2021

# Historial de versiones

|  |  |  |  |
| --- | --- | --- | --- |
| **Fecha** | **Versión** | **Descripción de los cambios** | **Sprint** |
| 15/09/2021 | V1.1 | * Creación del documento. * Añadidos primeros puntos y configuración del documento. | Entregable Diciembre |
| 19/09/2021 | V1.2 | * Añadidos diagrama UML de dominio/diseño. | Entregable Diciembre |
| 03/10/2021 | V1.3 | * Añadidas las decisiones 1 y 2. * Añadido diagrama de capas. * Actualizado diagrama UML. | Entregable Diciembre |
| 10/10/2021 | V1.4 | * Añadido diagrama de capas. * Actualizadas las decisiones | Entregable Diciembre |
| 21/10/2021 | V1.5 | * Actualizados los diagramas de capas y UML. | Entregable Diciembre |
| 5/11/2021 | V1.6 | * Añadidos los patrones arquitectónicos FrontController, Domain Model, Repository,   Service Layer, Layer Supertype. | Entregable Diciembre |
| 13/11/2021 | V1.7 | * Actualizado diagrama de capas. * Actualizado diagrama UML de dominio. | Entregable Diciembre |
| 27/11/2021 | V1.8 | * Actualizadas las decisiones de diseño. * Añadidas últimas modificaciones pre entrega para cerrar el documento. | Entregable Diciembre |
| 06/12/2021 | V1.9 | * Revisión del documento y actualización de los puntos faltantes. * Finalización del documento | Entregable Diciembre |

Contents

[GRUPO G2-Dicimbre 2021 1](#_Toc89633864)

[Historial de versiones 1](#_Toc89633865)

[Introducción 5](#_Toc89633866)

[Diagrama(s) UML: 5](#_Toc89633867)

[Diagrama de Dominio/Diseño 5](#_Toc89633868)

[Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios) 6](#_Toc89633869)

[Patrones de diseño y arquitectónicos aplicados 6](#_Toc89633870)

[Patrón: Modelo Vista Controlador (MVC) 6](#_Toc89633871)

[Tipo: de Diseño 6](#_Toc89633872)

[Contexto de Aplicación 6](#_Toc89633873)

[Clases o paquetes creados 6](#_Toc89633874)

[Ventajas alcanzadas al aplicar el patrón 7](#_Toc89633875)

[Patrón: Front Controller. 7](#_Toc89633876)

[Tipo: de Diseño 7](#_Toc89633877)

[Contexto de Aplicación 7](#_Toc89633878)

[Clases o paquetes creados 7](#_Toc89633879)

[ActorController, AdministratorController, CinemaController, CrashController, DirectorController, CinemaRoomController, SeatController, SessionController, SubscriberController, TrailerController, UserController, WelcomeController. 7](#_Toc89633880)

[Ventajas alcanzadas al aplicar el patrón 7](#_Toc89633881)

[Patrón: Domain Model. 7](#_Toc89633882)

[Tipo: de Diseño 7](#_Toc89633883)

[Contexto de Aplicación 7](#_Toc89633884)

[Clases o paquetes creados 7](#_Toc89633885)

[Actor, Administrator, Authorities, BaseEntity, Cinema, CinemaRoom, Director, Film, Seat, Session, ReservaVuelo, Subscriber, Trailer, User. 7](#_Toc89633886)

[Ventajas alcanzadas al aplicar el patrón 8](#_Toc89633887)

[Patrón: Repository 8](#_Toc89633888)

[Tipo: de Diseño 8](#_Toc89633889)

[Contexto de Aplicación 8](#_Toc89633890)

[Clases o paquetes creados 8](#_Toc89633891)

[Ventajas alcanzadas al aplicar el patrón 8](#_Toc89633892)

[Patrón: Service Layer 8](#_Toc89633893)

[Tipo: de Diseño 8](#_Toc89633894)

[Contexto de Aplicación 8](#_Toc89633895)

[Clases o paquetes creados 8](#_Toc89633896)

[Ventajas alcanzadas al aplicar el patrón 8](#_Toc89633897)

[Patrón: Layer Supertype. 8](#_Toc89633898)

[Tipo: de Diseño 8](#_Toc89633899)

[Contexto de Aplicación 8](#_Toc89633900)

[Clases o paquetes creados 8](#_Toc89633901)

[Ventajas alcanzadas al aplicar el patrón 9](#_Toc89633902)

[Patrón: Capas. 9](#_Toc89633903)

[Tipo: Arquitectónico 9](#_Toc89633904)

[Contexto de Aplicación 9](#_Toc89633905)

[Clases o paquetes creados 9](#_Toc89633906)

[Ventajas alcanzadas al aplicar el patrón 9](#_Toc89633907)

[Decisiones de diseño 10](#_Toc89633908)

[Decisión 1: Importación de datos al sistema 10](#_Toc89633909)

[Descripción del problema: 10](#_Toc89633910)

[Alternativas de solución evaluadas: 10](#_Toc89633911)

[Justificación de la solución adoptada 11](#_Toc89633912)

[Decisión 2: Implementación de la entidad Reservation 11](#_Toc89633913)

[Descripción del problema: 11](#_Toc89633914)

[Alternativas de solución evaluadas: 11](#_Toc89633915)

[Justificación de la solución adoptada 11](#_Toc89633916)

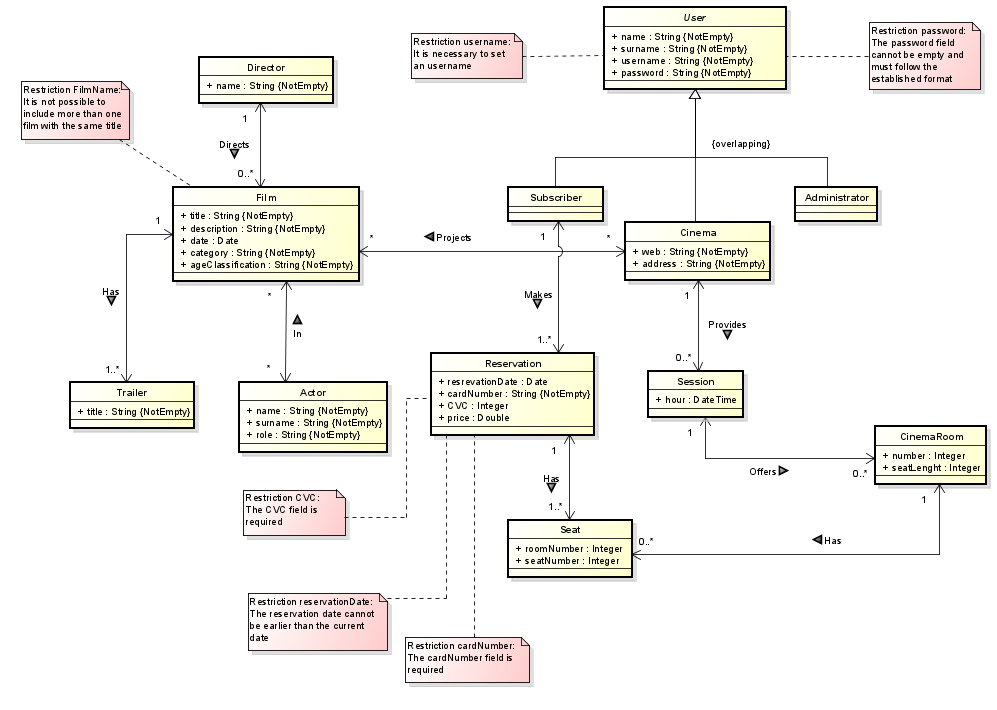
# Introducción

La idea principal de la plataforma PeliculasOnline es la de crear un sistema informático en el cual los usuarios puedan buscar información sobre las películas en cines, realizar una reserva seleccionando el horario de sesión y cine en el que quieren ver la película. Las funciones principales son: control y gestión de las películas que se proyectan en los cines, facilitar el contacto de los cines con los espectadores, realización de reservas para ver las películas. Todos los datos del cine, subscriptores y películas serán almacenados en la base de datos.

# Diagrama(s) UML:

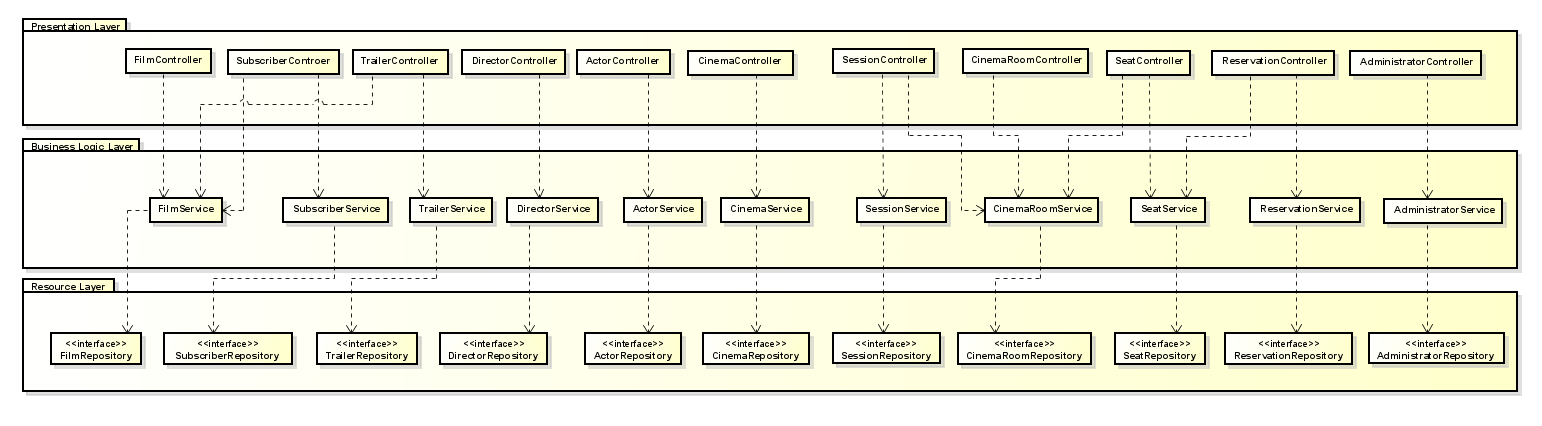
## Diagrama de Dominio/Diseño

Todos los modelos incluidos en el proyecto heredan de la entidad BaseEntity, por ello se han omitido la estructura de entidades/relaciones hacia BaseEntity para simplificar el diagrama:



## Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

En este apartado se muestran las relaciones entre los controladores, servicios y repositorios mediante las capas que se han implementado y forman la aplicación.



# Patrones de diseño y arquitectónicos aplicados

## Patrón: Modelo Vista Controlador (MVC)

### Tipo: de Diseño

### Contexto de Aplicación

El proyecto se estructurado según el patrón MVC (Modelo-Vista-Controlador).

Las clases de modelo representan el modelo del proyecto de cada una de las entidades, los archivos.jsp representan las vistas y las clases Controller para los controladores.

### Clases o paquetes creados

#### En este punto se indican las clases o paquetes creados como resultado de la aplicación del patrón.

* model(Modelos): Actor, Administrator, Authorities, BaseEntity, Cinema, CinemaRoom, Director, Film, Seat, Session, ReservaVuelo, Subscriber, Trailer, User.
* web(controladores): ActorController, AdministratorController, CinemaController, CrashController, DirectorController, CinemaRoomController, SeatController, SessionController, SubscriberController, TrailerController, UserController, WelcomeController.
* jsp(vistas): Se han dividido en carpetas con el nombre de la entidad a la que hacen referencia. actors, administrators, cinemaRooms, cinemas, directors, films, seats, sessions, subscribers, trailers, users.

### Ventajas alcanzadas al aplicar el patrón

* Identificación de forma clara del tipo de lógica aplicado en cada parte, facilidad de mantenimiento y escalabilidad de la aplicación.
* Facilidad para implementar distintas representaciones de los mismos datos.
* Facilidad para la realización de pruebas unitarias de cada uno de los componentes y para aplicar un desarrollo guiado por pruebas.
* Reutilización de los componentes desarrollados en el modelo.
* Facilidad para el diseño de la aplicación web.
* Gran control sobre el comportamiento de la aplicación web.

## Patrón: Front Controller.

### Tipo: de Diseño

### Contexto de Aplicación

Se ha usado para mostrar los datos en la interfaz de usuario, para la creación de las vistas necesarias para el proyecto y para establecer las URLs del mismo.

Se encuentra en el paquete /spring/controllers

### Clases o paquetes creados

### ActorController, AdministratorController, CinemaController, CrashController, DirectorController, CinemaRoomController, SeatController, SessionController, SubscriberController, TrailerController, UserController, WelcomeController.

### Ventajas alcanzadas al aplicar el patrón

* Flexibilidad a la hora de establecer un controlador apropiado para las solicitudes a través de las URLs.
* Procesar los datos de los formularios y poder validarlos y transformarlos en el propio controlador.

## Patrón: Domain Model.

### Tipo: de Diseño

### Contexto de Aplicación

Creamos objetos en el modelo a partir del dominio, donde guardaremos todos los datos que necesitemos relacionados con el dominio, como son las clases y los atributos.

Se encuentran en el paquete /spring/models

### Clases o paquetes creados

### Actor, Administrator, Authorities, BaseEntity, Cinema, CinemaRoom, Director, Film, Seat, Session, ReservaVuelo, Subscriber, Trailer, User.

### Ventajas alcanzadas al aplicar el patrón

* Permite implementar una lógica de negocio más compleja.

## Patrón: Repository

### Tipo: de Diseño

### Contexto de Aplicación

Lo usamos para encapsular la lógica requerida para acceder a los datos.

### Clases o paquetes creados

* repository(Repositorios): ActorRepository, AdministratorRepository, AuthoritiesRepository, CinemaRepository, CinemaRoomRepository, DirectorRepository, FilmRepository, SeatRepository, SessionRepository, SubscriberRepository, TrailerRepository, UserRepository.

### Ventajas alcanzadas al aplicar el patrón

* Centraliza la lógica de datos
* Proporciona una arquitectura flexible
* Si se quiere modificar el acceso a los datos, no es necesario cambiar la lógica del repositorio.

## Patrón: Service Layer

### Tipo: de Diseño

### Contexto de Aplicación

Lo utilizamos para establecer un conjunto de operaciones disponibles.

### Clases o paquetes creados

* service(servicios): ActorService, AdministratorService, AuthoritiesService, CinemaService, CinemaRoomService, DirectorService, FilmService, SeatService, SessionService, SubscriberService, TraileServicrService, UserService.

### Ventajas alcanzadas al aplicar el patrón

* Ayuda a reducir la sobrecarga conceptual relacionada con la gestión de servicios.

## Patrón: Layer Supertype.

### Tipo: de Diseño

### Contexto de Aplicación

Se crea una clase abstracta común que contiene el campo identidad para los modelos que implementamos en la aplicación.

Se encuentran en el paquete /spring/models/baseEntity.java

### Clases o paquetes creados

BaseEntity

### Ventajas alcanzadas al aplicar el patrón

* Al crear una clase común para todas las entidades no se sobrecargan con más información de la necesaria.

## Patrón: Capas.

### Tipo: Arquitectónico

### Contexto de Aplicación

Lo hemos usado para dividir las responsabilidades del proyecto en 3 capas. La capa de recursos formada por el repositorio; la capa de lógica de negocio formada por los servicios y las entidades; y la capa de presentación formada por las vistas y los controladores.

### Clases o paquetes creados

#### Capa de recursos:

* Repository(Repositorios): ActorRepository, AdministratorRepository, AuthoritiesRepository, CinemaRepository, CinemaRoomRepository, DirectorRepository, FilmRepository, SeatRepository, SessionRepository, SubscriberRepository, TrailerRepository, UserRepository.

#### Capa de lógica de negocio:

* model(Modelos): Actor, Administrator, Authorities, BaseEntity, Cinema, CinemaRoom, Director, Film, Seat, Session, ReservaVuelo, Subscriber, Trailer, User.
* service(servicios): ActorService, AdministratorService, AuthoritiesService, CinemaService, CinemaRoomService, DirectorService, FilmService, SeatService, SessionService, SubscriberService, TraileServicrService, UserService.

#### Capa de presentación:

* web(controladores): ActorController, AdministratorController, CinemaController, CrashController, DirectorController, CinemaRoomController, SeatController, SessionController, SubscriberController, TrailerController, UserController, WelcomeController.
* jsp(vistas): Están divididos en carpetas con el nombre de la entidad a la que hacen referencia. actors, administrators, cinemaRooms, cinemas, directors, films, seats, sessions, subscribers, trailers y users.

### Ventajas alcanzadas al aplicar el patrón

* Alta cohesión (cada capa se centra en una tarea determinada) y necesitan pocas dependencias.
* Facilidad de separación de responsabilidades.
* Las capas son independientes unas de otras
* Facilidad para reemplazar las capas.

# Decisiones de diseño

En esta sección se describen las decisiones de diseño aplicadas a lo largo del desarrollo de la aplicación “PeliculasOnline” que no están incluidas en la aplicación de patrones de diseño o arquitectónicos.

## Decisión 1: Importación de datos al sistema

### Descripción del problema:

Para el correcto funcionamiento y prueba de la aplicación es fundamental tener una base de datos lo más completa posible que cubra todas las casuísticas contempladas.

Para ello se necesitan datos con los que se puedan ejecutar todas las pruebas y comprobar el funcionamiento de todos los métodos implementados en la aplicación.

En esta tarea se han encontrado problemas al incluir los datos como parte del script de inicialización de la base de datos. El arranque del sistema se ralentiza y las pruebas resultan más tediosas.

A continuación, se detallan una serie de alternativas exponiendo las ventajas y desventajas de cada una de ellas para elegir la mejor.

### Alternativas de solución evaluadas:

*Alternativa 1.a: Incluir los datos en el script de Inicialización de la BD (data.sql).*

#### Ventajas:

* Simple, no requiere nada más que escribir el SQL que genera los datos.

#### Inconvenientes:

* Se ralentiza el sistema a la hora de trabajar en la implementación.
* Es necesario poblar el script de la Base de datos con datos suficientes para cubrir todas las casuísticas.

*Alternativa 1.b: Crear otro script con datos adicionales (data2.sql). Además, sería necesario implementar un nuevo controlador encargado de leer este script y ejecutar las consultas para cargar estos nuevos datos.*

#### Ventajas:

* Reutilización de los datos contemplados en el script original (data.sql).
* Se mejora el funcionamiento del sistema a la hora de desarrollar y realizar pruebas.

#### Inconvenientes:

* Impacto en modelo de división en capas si no se usa un servicio para cargar los datos que se relacione con el controlador asociado a la carga de datos desde el script.
* Necesidad de carga de más datos para rellenar el script (data2.sql).

### Justificación de la solución adoptada

Puesto que la división en capas del sistema es fundamental y al revisar las diferentes alternativas, tras valorar las ventajas e inconvenientes, se ha elegido usar la alternativa 1.a.

Con esta alternativa se reduce notablemente la cantidad de datos que es necesario crear al sólo hacer uso de un script (data.sql). Con ello se consigue disponer de los diferentes datos necesarios para las pruebas de una manera sencilla y rápida.

## Decisión 2: Implementación de la entidad Reservation

### Descripción del problema:

Para la gestión de las reservas de asientos(Seats) para cada sala(cinemaRoom) asociada a la sesión(Session) del cine correspondiente es necesario almacenar estos datos de una forma clara y fácil para favorecer la gestión de los mimos.

A continuación, se indican una serie de alternativas para poder implementar y gestionar todos estos datos referentes a las reservas de asientos para asistir a una sesión de cine que serán incluidos en el sistema.

### Alternativas de solución evaluadas:

*Alternativa 2.a: Crear la entidad Reservation para incluir toda la información referente a la reserva del asiento en la sala escogida y la sesión de cine asociada.*

#### Ventajas:

* Facilidad de realizar cambios y claridad en el almacenamiento de las reservas.
* Facilidad de gestión de las reservas.

#### Inconvenientes:

* Añade una nueva entidad al sistema y toda la lógica necesaria para gestionarla.

*Alternativa 2.b: Incluir la lógica de las reservas en la relación de entidades CinemaRoom - Seat*

#### *Ventajas:*

* No es necesario la creación de la entidad Reservation.

#### Inconvenientes:

* Menos claridad para administrar las reservas de asientos que al usar la entidad Reservation.

### Justificación de la solución adoptada

Se ha optado por la alternativa 2.a ya que, aunque es necesario implementar una entidad nueva (Reservation), queda mucho más claro y es más fácil de gestionar todo lo relacionado con la reserva de asientos de este modo que incluyendo las reservas en la relación CinemaRoom – Seat.