DP1 2020-2021

Documento de Diseño del Sistema

Proyecto “**PelículasOnline**”

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

# Miembros:

* Toledo Villalba, Daniel

Tutor:

* Parejo, José Antonio

# GRUPO G2-Diciembre

Versión 1.3

# 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 |
| 20/09/2021 | V1.2 | * Añadido diagrama UML de dominio/diseño. * Añadido patrón caché en la aplicación PeliculasOnline. | Entregable Diciembre |
| 03/10/2021 | V1.3 | * Añadidas las decisiones 1,2 y 3. * Añadido diagrama de capas. * Actualizado diagrama UML. | Entregable Diciembre |
| 10/10/2021 | V1.4 | * Añadido diagrama de capas. * Actualizadas las decisiones 2 y 3. * Añadidas las decisiones 4 y 5. | Entregable Diciembre |
| 21/10/2021 | V1.5 | * Añadida Decisión 4. * 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 díseño. * 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

[Historial de versiones 1](#_TOC_250059)

[Contenidos 2](#_TOC_250058)

[Introducción **5**](#_TOC_250057)

[Diagramas UML: **5**](#_TOC_250056)

[Diagrama de Dominio/Diseño 5](#_TOC_250055)

[Diagrama de Capas (se incluyen Controladores, Servicios y Repositorios) 6](#_TOC_250054)

[Patrones de diseño y arquitectónicos aplicados **6**](#_TOC_250053)

[Patrón: Modelo Vista Controlador (MVC) 6](#_TOC_250052)

[Contexto de Aplicación 6](#_TOC_250051)

[Clases o paquetes creados 6](#_TOC_250050)

[Ventajas alcanzadas al aplicar el patrón 6](#_TOC_250049)

[Patrón: Front Controller. 7](#_TOC_250048)

[Contexto de Aplicación 7](#_TOC_250047)

[Clases o paquetes creados 7](#_TOC_250046)

[Ventajas alcanzadas al aplicar el patrón 7](#_TOC_250045)

[Patrón: Domain Model. 7](#_TOC_250044)

[Contexto de Aplicación 7](#_TOC_250043)

[Clases o paquetes creados 7](#_TOC_250042)

[Ventajas alcanzadas al aplicar el patrón 7](#_TOC_250041)

[Patrón: Repository 8](#_TOC_250040)

[Contexto de Aplicación 8](#_TOC_250039)

[Clases o paquetes creados 8](#_TOC_250038)

[Ventajas alcanzadas al aplicar el patrón 8](#_TOC_250037)

[Patrón: Service Layer 8](#_TOC_250036)

[Contexto de Aplicación 8](#_TOC_250035)

[Clases o paquetes creados 8](#_TOC_250034)

[Ventajas alcanzadas al aplicar el patrón 8](#_TOC_250033)

[Patrón: Layer Supertype. 8](#_TOC_250032)

[Contexto de Aplicación 8](#_TOC_250031)

[Clases o paquetes creados 8](#_TOC_250030)

[Ventajas alcanzadas al aplicar el patrón 8](#_TOC_250029)

[Patrón: Capas. 9](#_TOC_250028)

[Contexto de Aplicación 9](#_TOC_250027)

[Clases o paquetes creados 9](#_TOC_250026)

[Ventajas alcanzadas al aplicar el patrón 9](#_TOC_250025)

[Decisiones de diseño 10](#_TOC_250024)

[Decisión 1: Importación de datos reales para demostración 10](#_TOC_250023)

[Descripción del problema: 10](#_TOC_250022)

[Alternativas de solución evaluadas: 10](#_TOC_250021)

[Justificación de la solución adoptada 11](#_TOC_250020)

[Decisión 2: Mostrar los hoteles por la provincia 12](#_TOC_250019)

[Descripción del problema: 12](#_TOC_250018)

[Alternativas de solución evaluadas: 12](#_TOC_250017)

[Justificación de la solución adoptada 12](#_TOC_250016)

[Decisión 3: Entidades comentarios 13](#_TOC_250015)

[Descripción del problema: 13](#_TOC_250014)

[Alternativas de solución evaluadas: 13](#_TOC_250013)

[Justificación de la solución adoptada 13](#_TOC_250012)

[Decisión 4: Solicitud de inscripción hotel 14](#_TOC_250011)

[Descripción del problema: 14](#_TOC_250010)

[Alternativas de solución evaluadas: 14](#_TOC_250009)

[Justificación de la solución adoptada 14](#_TOC_250008)

[Decisión 5: Regla de negocio 3 15](#_TOC_250007)

[Descripción del problema: 15](#_TOC_250006)

[Alternativas de solución evaluadas: 15](#_TOC_250005)

[Justificación de la solución adoptada 15](#_TOC_250004)

[Decisión 6: Elección del validador para la regla de negocio 2 16](#_TOC_250003)

[Descripción del problema: 16](#_TOC_250002)

[Alternativas de solución evaluadas: 16](#_TOC_250001)

[Justificación de la solución adoptada 16](#_TOC_250000)

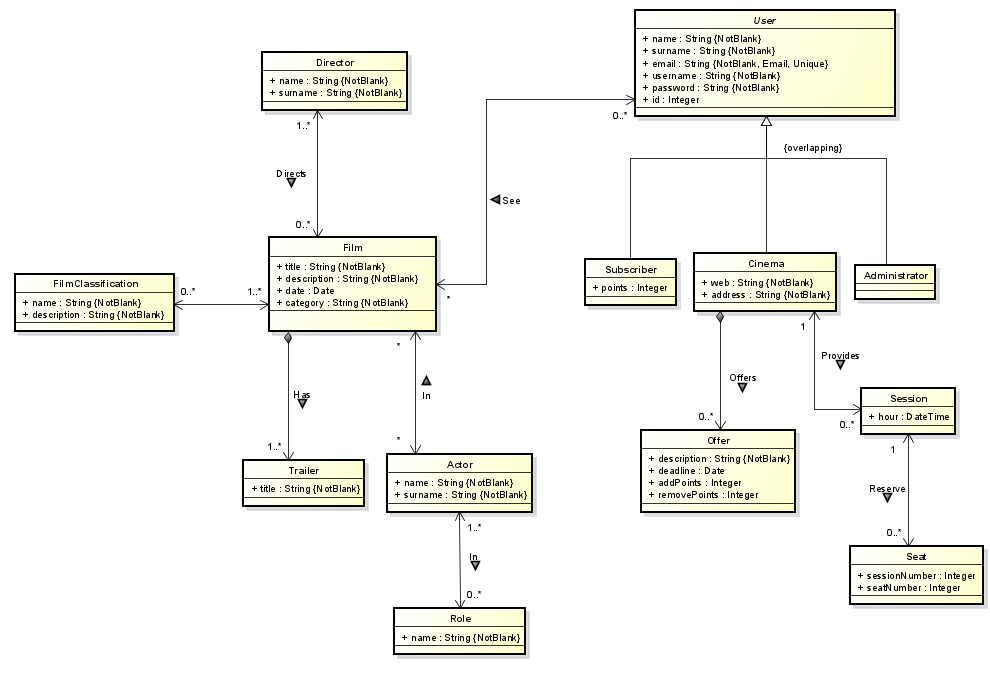
# Introducción

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 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 y obtención de bonificaciones al hacer uso de la plataforma. Todos los datos del cine, subscriptores y películas serán almacenados en una base de datos.

# Diagramas UML:

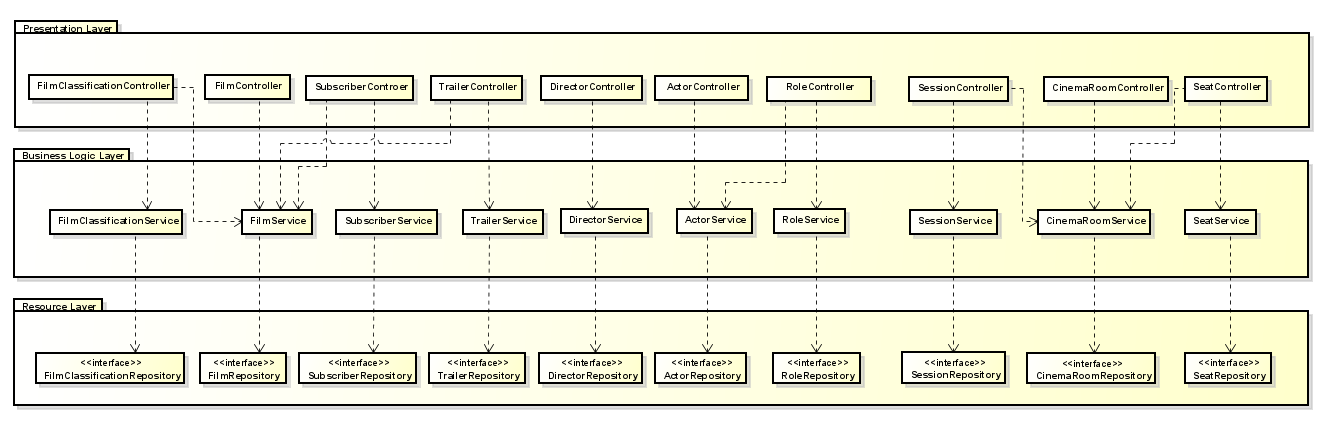
## Diagrama de Dominio/Diseño

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, FilmClassification, Role, Seat, Session, ReservaVuelo, Subscriber, Trailer, User.
* web(controladores): ActorController, AdministratorController, CinemaController, CrashController, DirectorController, CinemaRoomController, FilmClassificationController, RoleController, 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, filmClassifications, films, roles, 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, FilmClassificationController, RoleController, 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, FilmClassification, Role, 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, RoleRepository,FilmClassificationRepository, 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, FilmClassificationService, RoleService, 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, RoleRepository,FilmClassificationRepository, SeatRepository, SessionRepository, SubscriberRepository, TrailerRepository, UserRepository.

#### Capa de lógica de negocio:

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

#### Capa de presentación:

* web(controladores): ActorController, AdministratorController, CinemaController, CrashController, DirectorController, CinemaRoomController, FilmClassificationController, RoleController, SeatController, SessionController, SubscriberController, TrailerController, UserController, WelcomeController.
* jsp(vistas): Están divididos en carpetas con el nombre de la entidad a la que hacen referencia. Actividades, AgenActs, CompVuelo, Hoteles, inscripcionHoteles, reservasHotel, reservasVuelo, reservasActividad, Search, User, Vuelos.

### Ventajas alcanzadas al aplicar el patrón

* Favorece una alta cohesión (cada capa se centra en una tarea determinada) y bajo acoplamiento (pocas dependencias).
* Promueve la separación de responsabilidades.
* Las capas son independientes (no necesitan saber de otras y sus detalles) y son fáciles de reemplazar.
* Permite el desarrollo de capas concurrentes.

# Decisiones de diseño

En esta sección se describen las decisiones de diseño aplicadas a lo largo del desarrollo de la aplicación que van más allá de la mera aplicación de patrones de diseño o arquitectónicos.

## Decisión 1: Importación de datos reales para demostración

### Descripción del problema:

A la hora de comprobar el correcto funcionamiento de nuestra aplicación, es fundamental tener un banco de pruebas lo más completo posible, analizando todas las posibilidades. Para ello, necesitamos datos con los que podamos ejecutar ciertas pruebas y comprobar realmente el funcionamiento de los métodos implementados en nuestra aplicación.

Como grupo nos gustaría poder hacer pruebas con un conjunto de datos reales suficientes, porque resulta más motivador. El problema es que al incluir todos esos datos como parte del script de inicialización de la base de datos, el arranque del sistema para desarrollo y pruebas resulta muy tedioso.

A continuación, vamos a elaborar una serie de alternativas exponiendo sus ventajas y desventajas para así poder debatir y elegir cual sería la mejor.

### Alternativas de solución evaluadas:

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

#### Ventajas:

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

#### Inconvenientes:

* Ralentiza todo el trabajo con el sistema para el desarrollo.
* Tenemos que buscar nosotros los datos reales.

*Alternativa 1.b: Crear un script con los datos adicionales a incluir (extra-data.sql) y un controlador que se encargue de leerlo y lanzar las consultas a petición cuando queramos tener más datos para mostrar.*

#### Ventajas:

* Podemos reutilizar parte de los datos que ya tenemos especificados en (data.sql).
* No afecta al trabajo diario de desarrollo y pruebas de la aplicación.

#### Inconvenientes:

* Puede suponer saltarnos hasta cierto punto la división en capas si no creamos un servicio de carga de datos.
* Tenemos que buscar nosotros los datos reales adicionales.

*Alternativa 1.c: Crear un controlador que llame a un servicio de importación de datos, que a su vez invoca a un cliente REST de la API de datos oficiales de XXXX para traerse los datos, procesarlos y poder grabarlos desde el servicio de importación.*

#### Ventajas:

* No necesitamos inventarnos ni buscar nosotros los datos.
* Cumple 100% con la división en capas de la aplicación.
* No afecta al trabajo diario de desarrollo y pruebas de la aplicación.

#### Inconvenientes:

* Supone mucho más trabajo.
* Añade cierta complejidad al proyecto.

### Justificación de la solución adoptada

Tras haber visto en clase que la división en capas es fundamental y tras debatir en el grupo las alternativas expuestas anteriormente, hemos decidido que la opción que más se adecua a lo que queremos es la 1.a.

Hemos escogido esta opción ya que resulta muy fácil escribir una serie de datos adecuados a los que queremos probar, por lo que es una manera muy sencilla y rápida de obtener todos los datos necesarios para el desarrollo de la prueba.

## 

## Decisión 2: Mostrar los hoteles por la provincia

### Descripción del problema:

En nuestra aplicación, disponemos de un botón de búsqueda de hoteles, para la reserva de los mismos. Para facilitar la búsqueda, ya que normalmente un cliente ya sabe a dónde quiere ir de vacaciones, hemos visto conveniente crear un método en el que se puedan buscar los hoteles por la provincia elegida y así solo mostrar todos los hoteles disponibles para esa búsqueda.

Tras intentar crear un spin box o selector para el filtrado de hoteles por provincia, hemos visto imposible esa opción, ya que no utilizamos tecnología SPA. Vamos a exponer una serie de alternativas para poder llevar a cabo lo comentado anteriormente.

### Alternativas de solución evaluadas:

*Alternativa 2.a: Crear una lista por cada provincia en las vistas de forma que un jsp por cada provincia que muestre solo los hoteles de esa provincia.*

#### Ventajas:

* Los hoteles quedarían reducidos a sus provincias.

#### Inconvenientes:

* Supone mucho más trabajo.
* Añade muchísimos archivos casi idénticos, haciendo tedioso el crearlos.

*Alternativa 2.b:* Redirigir la lista de hoteles a otra lista con solo los hoteles que tengan la provincia seleccionada

#### Ventajas:

* Los hoteles quedarían reducidos a sus provincias.

#### Inconvenientes:

* Habría que desarrollar nuevos métodos en el controlador, repositorio y servicio para obtener los hoteles por la provincia.

### Justificación de la solución adoptada

Tras debatir con el grupo y haber probado las dos alternativas expuestas anteriormente, hemos llegado a la conclusión de que, como una historia de usuario nuestra es poder buscar por la provincia, la alternativa 2.b sería la más correcta, ya que nos parecía más sencillo implementarlo y más eficaz por la estructura que tiene nuestra aplicación, porque implementar un spinbox o selector nos estaba dando problemas ya que no podíamos obtener el valor seleccionado para hacer el filtro y de esta forma obtenemos el valor y reducimos la lista.

## 

## Decisión 3: Entidades comentarios

### Descripción del problema:

En nuestra aplicación, vemos fundamental que el cliente pueda dejar comentarios para así obtener un feedback de los hoteles y las actividades que están disponibles.

Principalmente implementamos el comentario como un atributo de las clases de hotel y actividad, pero nos daban una serie de errores que no pudimos solucionar.

A continuación, vamos a exponer una serie de alternativas para poder implementar bien los comentarios en nuestra aplicación.

### Alternativas de solución evaluadas:

*Alternativa 3.a: Crear entidades para los distintos tipos de comentarios que se pueden realizar.*

#### Ventajas:

* Facilidad de mantener y realizar cambios.

#### Inconvenientes:

* Añade muchísimos archivos casi idénticos, haciendo tedioso el crearlos.

*Alternativa 3.b: Crear un validador que te redirige al tipo de comentario*

#### Ventajas:

* No tiene archivos idénticos.

#### Inconvenientes:

* Difícil de desarrollar y mantener.

### Justificación de la solución adoptada

Nos hemos decantado por la alternativa 3.a porque nos parecía más sencillo de implementar y ordenar a la hora de buscar los archivos necesarios, en vez de hacer un validador que veíamos más complejo de implementar y nos iba a ocupar más tiempo del previsto.

## 

## Decisión 4: Solicitud de inscripción hotel

### Descripción del problema:

Realizar una solicitud para que un propietario de un hotel nos pueda enviar los datos de su hotel y que además se nos muestran las distintas solicitudes realizadas por todos los propietarios.

La primera vez que quisimos implementarlo decidimos probar con la entidad Hotel. Empezamos a crear métodos en el controlador y varios servicios, pero a la hora de realizar la solicitud de la inscripción se nos añadía directamente a la lista de hoteles ya inscritos.

### Alternativas de solución evaluadas:

*Alternativa 4.a: Usar la entidad Hotel ya creada y a partir de ahí realizar los formularios de inscripción.*

#### Ventajas:

* No hace falta crear más entidades
* Solo hace falta una vista para el formulario.

#### Inconvenientes:

* Se muestran todos los hoteles y las inscripciones juntos.
* Se necesitan crear atributos relacionados a las inscripciones dentro de la entidad Hotel para poderlos guardar en la base de datos, lo cual puede ser confuso por los nombres.
* El archivo es más grande, lo que provoca que sea más difícil encontrar los datos y errores.

*Alternativa 4.b: Crear una entidad llamada InscripcionHotel para realizar un formulario nuevo.*

#### Ventajas:

* Se puede crear el modelo a parte y no se relaciona directamente con los hoteles.
* Es más fácil encontrar errores y datos.

#### Inconvenientes:

* Datos casi idénticos a los hoteles, lo cual puede resultar un poco confuso.
* Se necesitan crear más vistas para el formulario y la lista de inscripciones.

### Justificación de la solución adoptada

Hemos decidido optar por la alternativa 4.b, porque nos parece más sencillo de implementar, puesto que tenemos una entidad solamente para las inscripciones y se encuentra separada de las demás entidades. A la hora de intentar implementar las inscripciones desde la entidad Hotel nos resultó muy tedioso el mostrar las inscripciones separadas de los hoteles, ya que salían en la misma vista y no conseguimos separarlas. Así, logramos crear una nueva lista en la que tenemos todas las solicitudes de una inscripción de hoteles.

## Decisión 5: Regla de negocio 3

### Descripción del problema:

Para afrontar nuestra regla de negocio número 3, principalmente habíamos pensado en el borrado del hotel en el cual su número de comentarios fuera mayor o igual que 5 y su valoración media sea menor que 2**,** pero generaba conflictos si un usuario ya había reservado un hotel que no cumpliese las características. Así conseguimos borrar los hoteles con una mala valoración para disponer de los mejores hoteles en nuestra app. Tras hacer varios cambios en el controlador, servicios y en el propio modelo ComentarioHotel no obtenemos los resultados obtenidos.

### Alternativas de solución evaluadas:

*Alternativa 5.a: Crear métodos que nos permiten no mostrar los hoteles que cumplan la propiedad comentada anteriormente.*

**Ventajas:**

### Facilidad de realizar y mantener.

* Cumple la funcionalidad sin tener que estar eliminado hoteles de la base de datos.

**Inconvenientes:**

### Realmente no se elimina el hotel y podría crear una carga de demasiados hoteles que no estamos usando.

### Justificación de la solución adoptada

Tras una reunión, se llegó a la conclusión de que era mucho más fácil y práctico que no mostrase el hotel en vez de eliminarlo de la base de datos, ya que en un futuro puede mejorar y sería el doble trabajo añadir de nuevo dicho hotel a la base de datos, obteniendo así el resultado que estábamos esperando.

Nos ha parecido una solución muy sencilla para resolver este problema y la hemos llevado a cabo.

## 

## Decisión 6: Elección del validador para la regla de negocio 2

### Descripción del problema:

Como medida de seguridad queremos establecer una serie de características en la contraseña. Estas son:

* Mínimo 8 caracteres.
* Mínimo una letra mayúscula.
* Mínimo un número.
* Mínimo una letra minúscula.

Hemos encontrado varias formas de poder implementar esta funcionalidad que se presenta a continuación.

### 

### Alternativas de solución evaluadas:

*Alternativa 6.a:* Establecer un patrón en el atributo password de la entidad User mediante la anotación @Pattern.

#### Ventajas:

* Fácil de implementar ya que está contenido en el propio framework.

#### Inconvenientes:

* No es reutilizable.

*Alternativa 6.b:* Crear una anotación personalizada para el atributo password de la entidad User.

#### Ventajas:

* Alta cohesión: establecer un validador propio para la regla de negocio.
* Se puede reutilizar el código para otras validaciones.
* Protección del almacenamiento de datos incorrectos en la base de datos.

#### Inconvenientes:

* Es más complejo tener un mensaje de error adecuado para los usuarios de la aplicación.

### Justificación de la solución adoptada

Hemos decidido optar por la alternativa 6.b ya que pensamos que es más conveniente realizar una validación propia para la contraseña, pudiendo reutilizar la anotación en futuras implementaciones de la aplicación. Conseguimos de esta manera que el código sea más limpio y fácil de utilizar.