DP1 2020-2021

Documento de Diseño del Sistema

Proyecto PetExtension

https://github.com/gii-is-DP1/dp1-2020-g2-08

Miembros :

- Miguel Durán González
- Manuel García Marchena
- Álvaro Gómez Pérez
- Pablo Mateos Angulo
- Isaac Muñiz Valverde
- Jorge Toledo Vega

Tutor: Irene Bedilia Estrada Torres

GRUPO G2-08

Versión 1

09/01/2021

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**
13/12/2020	V1	● Creación del documento	2
13/12/2020	V2	● Añadido diagrama de dominio/diseño ● Explicación de la aplicación del patrón caché	3
10/01/2021	V2.1	● Añadidos los diagramas de dominio/diseño y de capas del refugio	3
09/02/2020	V2.2	● Actualización diagrama shelter	4

Contents

Introducción

Al proyecto base de Petclinic, le vamos añadir una serie de funcionalidades, aportando valor al proyecto.
A la clínica le añadiremos una tienda de productos para mascotas, como comidas y accesorios donde comprarían los owners y pondrían valoraciones para que otros pudieran guiarse a la hora de comprar algo.
Queremos añadir un hotel para mascotas, en el que cada owner podría alojar a sus mascotas durante un período de tiempo reservando una habitación, teniendo también la opción de poner una valoración y opinión sobre la estancia, de forma que otros owners puedan ver como es el hotel.
Otra funcionalidad sería añadir un refugio en el que habría animales, en el que podríamos ver una serie de animales de cualquier tipo, dándonos la opción de adoptarlos para que pase a ser nuestra mascota.

Algunas de las funcionalidades más interesantes que hemos añadido al proyecto es, poder crear diferentes tipos de usuario dentro del sistema y que cada uno de estos tengan permisos exclusivos una vez inicien sesión, no es lo mismo un cliente que un owner; el cliente es referenciado a la tienda y el owner a los hoteles y a la clínica en sí. Otra cosa interesante es que los tres módulos que hemos implementado son totalmente independientes unos de otros favoreciendo así la cohesión del sistema y haciendo que si uno de los módulos falla, el resto sigue funcionando perfectamente.

Diagrama(s) UML:

Diagrama de Dominio/Diseño

En esta sección debe proporcionar un diagrama UML de clases que describa el modelo de dominio, recuerda que debe estar basado en el diagrama conceptual del documento de análisis de requisitos del sistema pero que debe:

- *Especificar la direccionalidad de las relaciones (a no ser que sean bidireccionales)*
- *Especificar la cardinalidad de las relaciones*
- *Especificar el tipo de los atributos*
- *Especificar las restricciones simples aplicadas a cada atributo de cada clase de domino*
- *Incluir las clases específicas de la tecnología usada, como por ejemplo BaseEntity, NamedEntity, etc.*
- *Incluir los validadores específicos creados para las distintas clases de dominio (indicando en su caso una relación de uso con el estereotipo <<validates>>.*

Un ejemplo de diagrama para los ejercicios planteados en los boletines de laboratorio sería (hemos omitido las generalizaciones hacia BaseEntity para simplificar el diagrama):

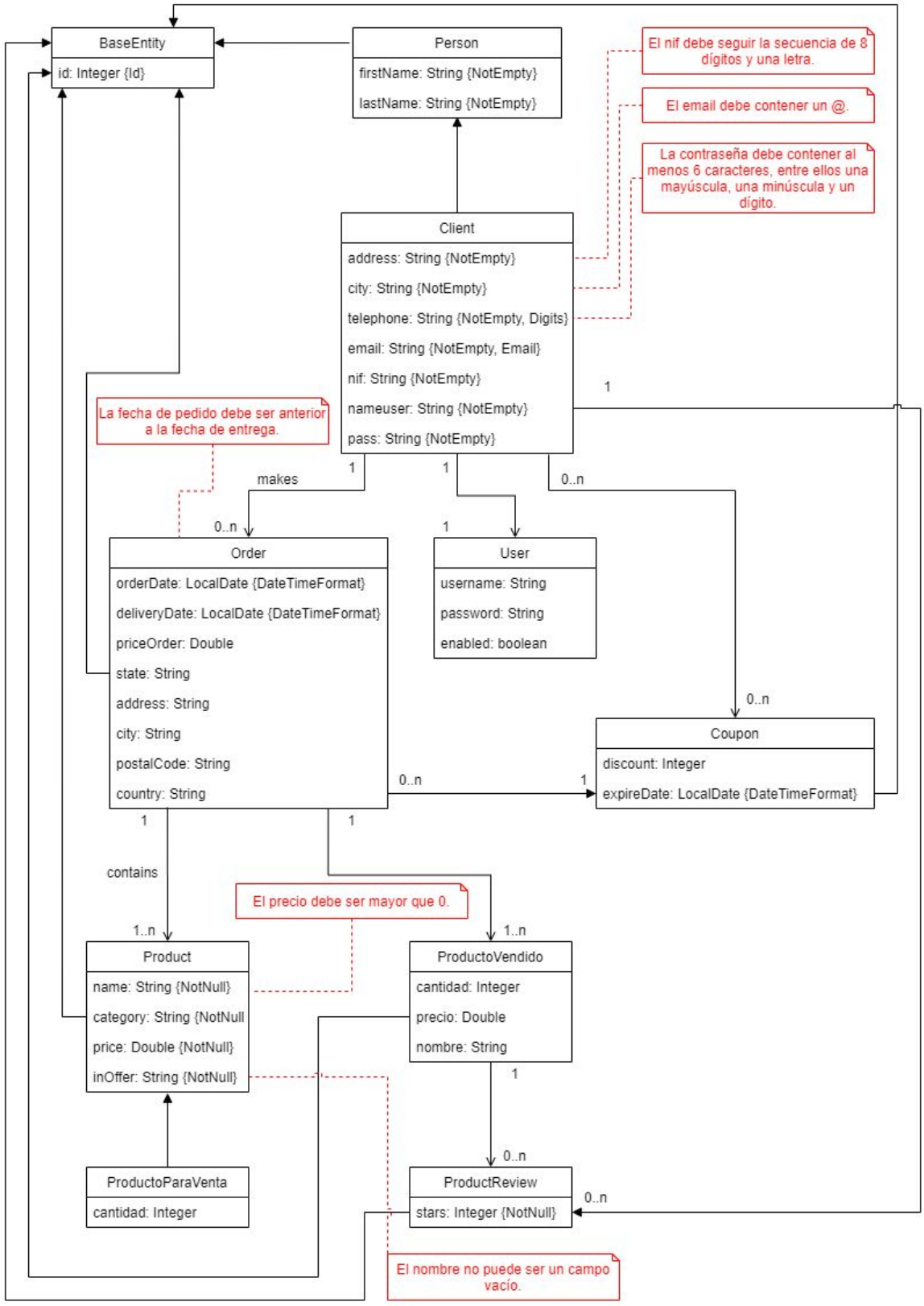

Figura 1. Diagrama de dominio/diseño tienda.

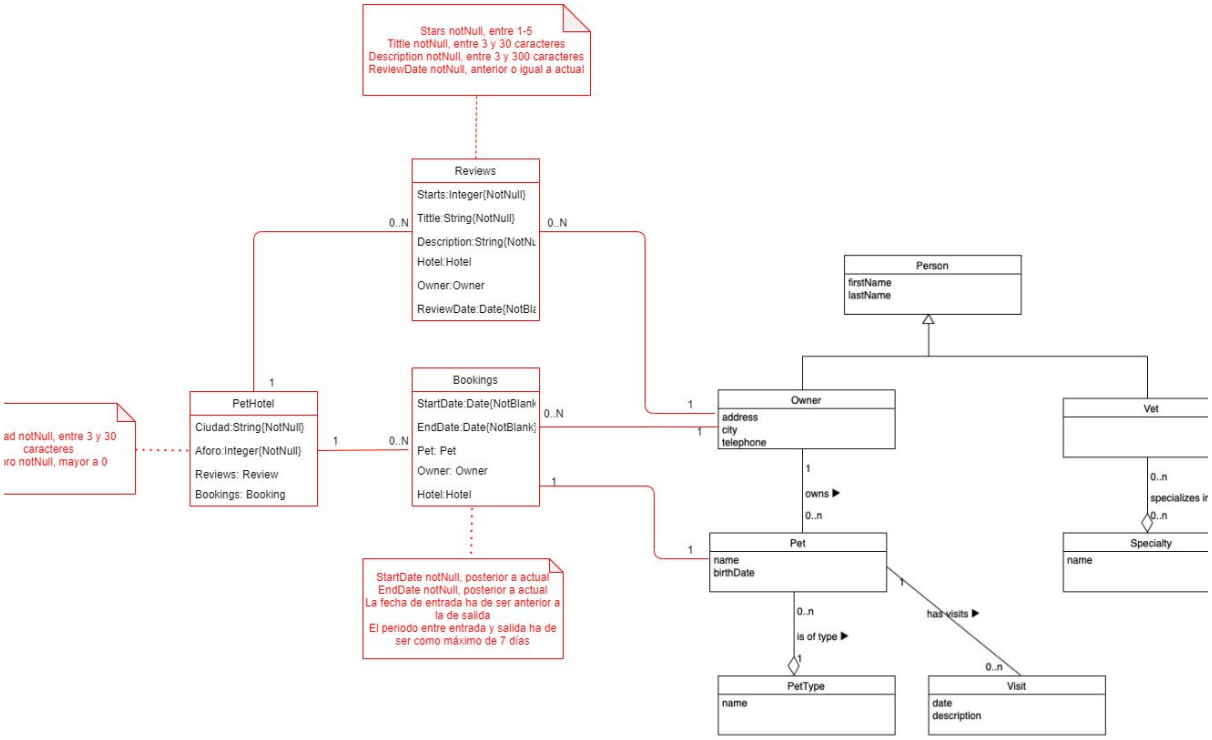

Figura 2. Diagrama de dominio/diseño hotel.

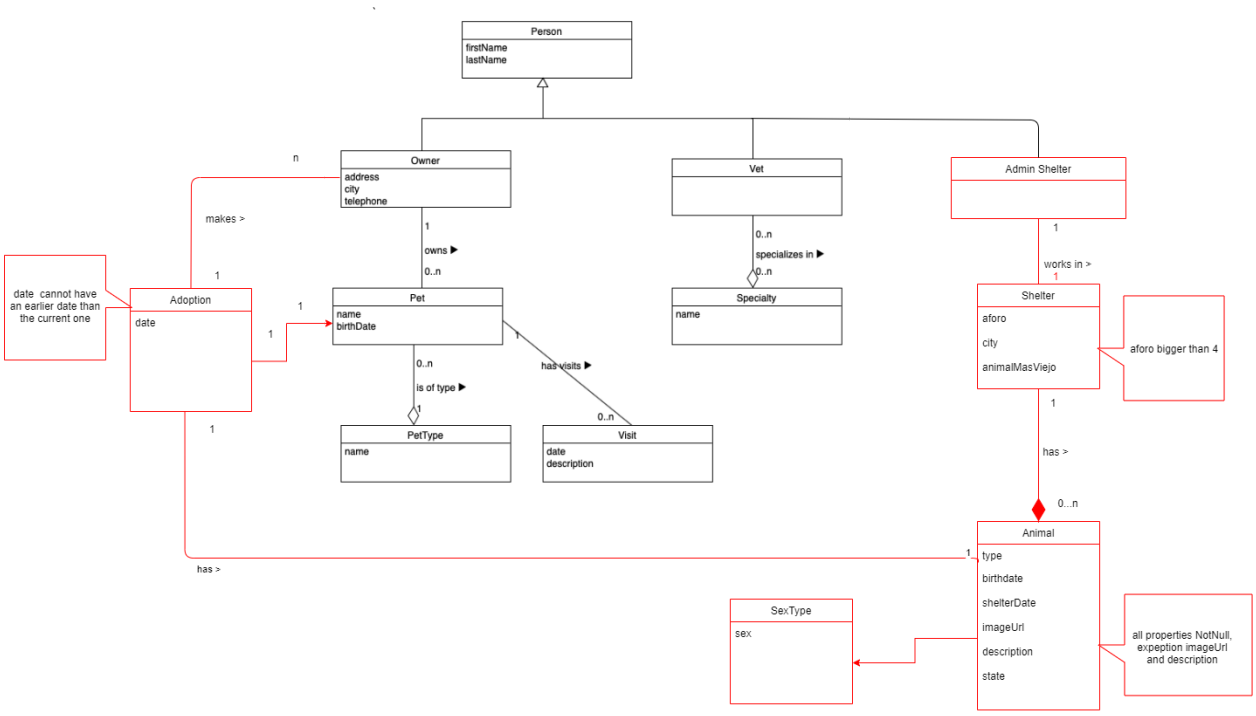

Figura 3. Diagrama de dominio/diseño refugio.

Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

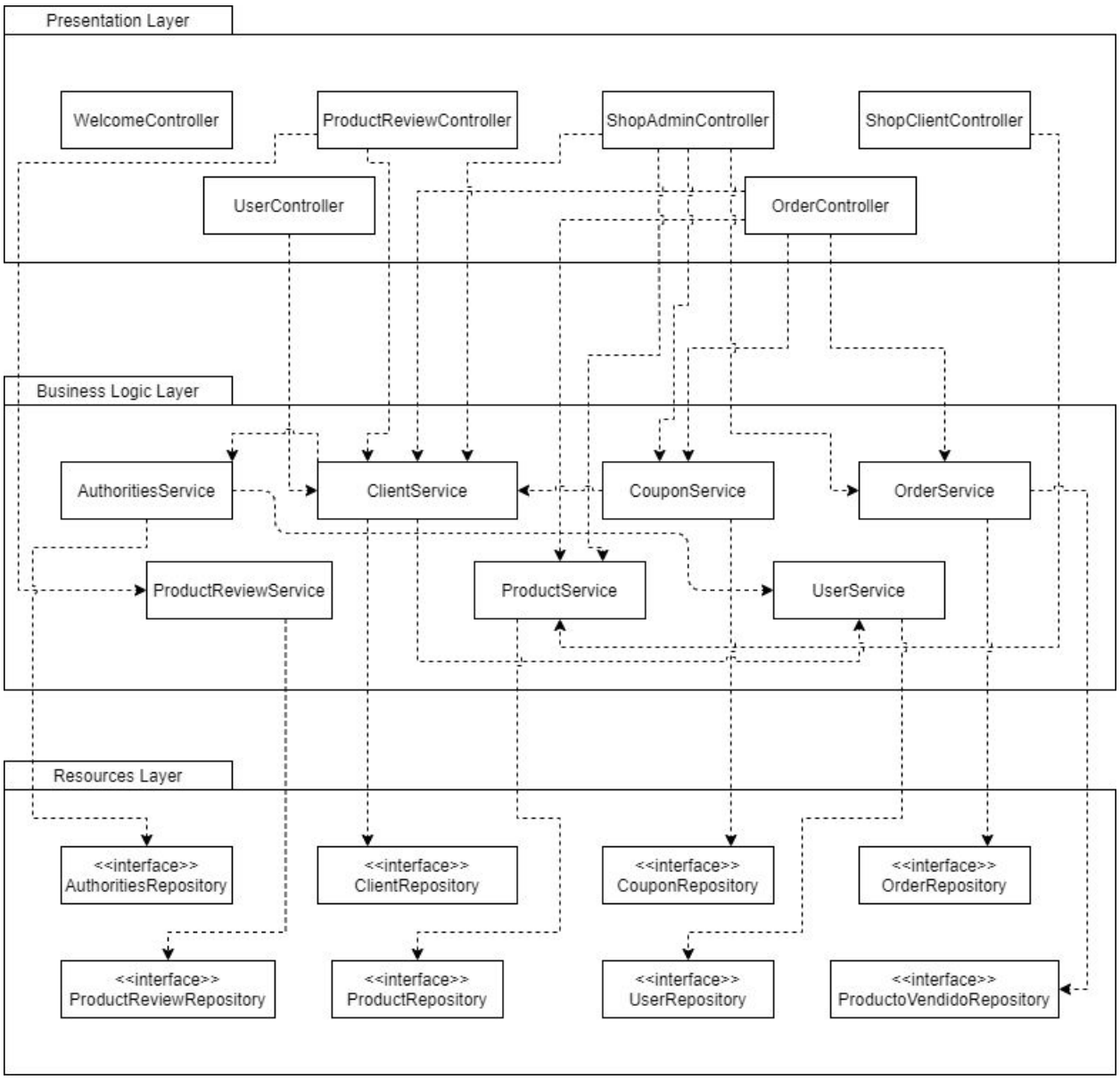

Figura 4. Diagrama de capas tienda.

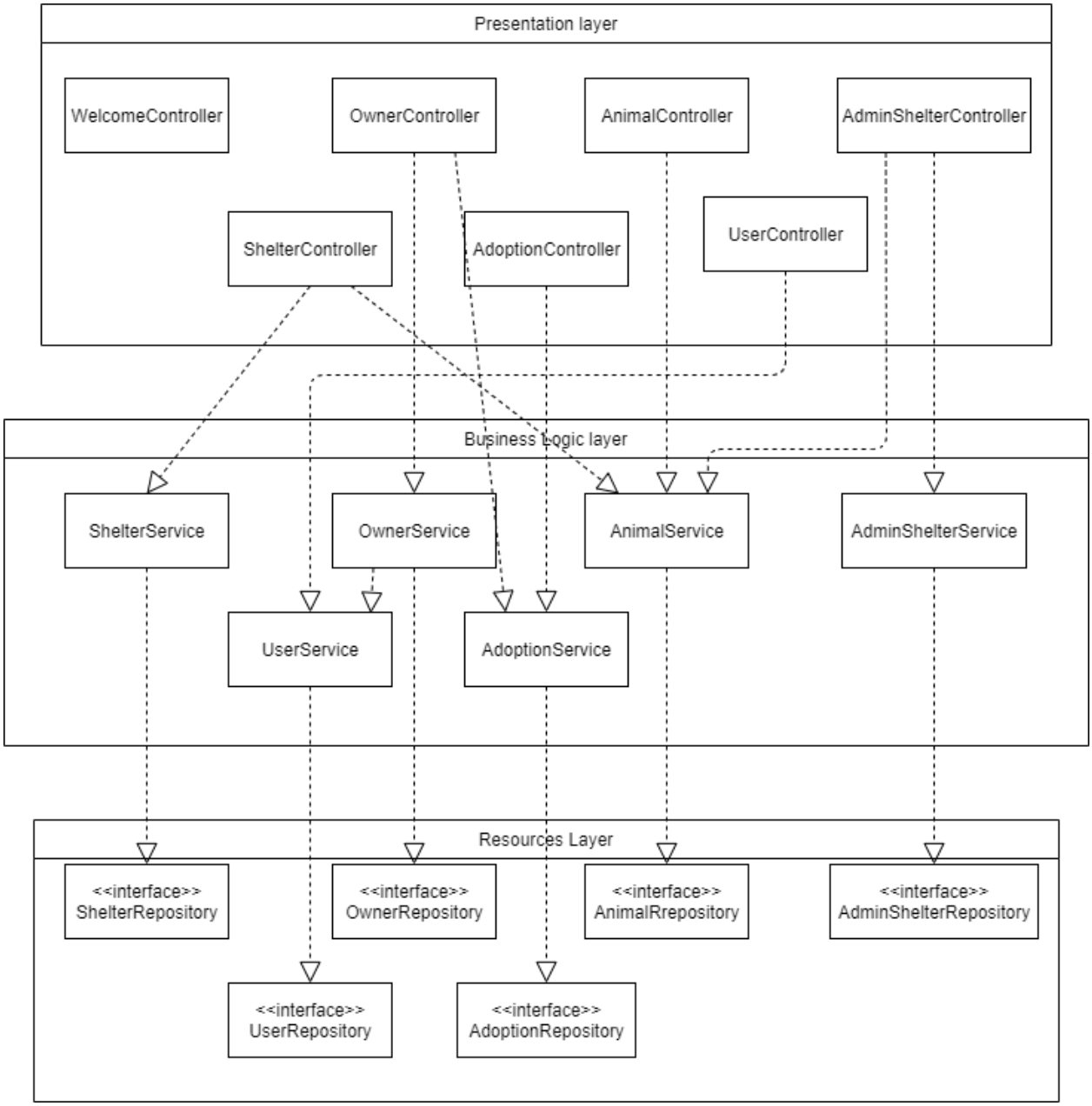

Figura 5. Diagrama de capas shelter.

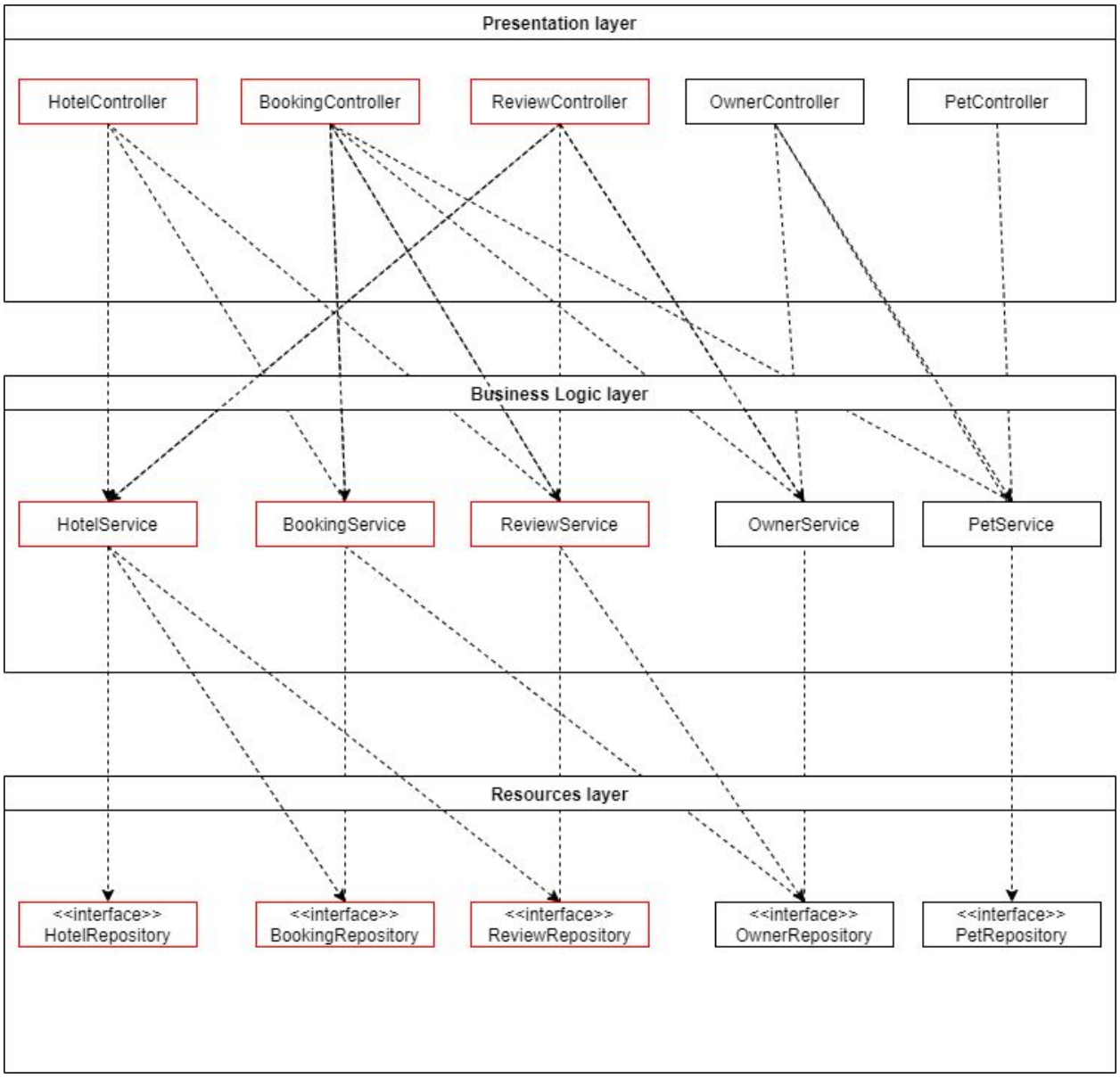

Figura 6. Diagrama de capas hotel.

Patrones de diseño y arquitectónicos aplicados

En esta sección se especifica el conjunto de patrones de diseño y arquitectónicos aplicados durante el proyecto. Para especificar la aplicación de cada patrón puede usar la siguiente plantilla:

Patrón: <MVC>

Tipo: Arquitectónico | de Diseño

Contexto de Aplicación

El patrón modelo vista controlador (MVC) se ha aplicado en todo el proyecto para dividir las vistas que verá el usuario en la interfaz gráfica, los controladores, que es donde los desarrolladores crean las

funcionalidades de la aplicación, y el modelo, que es donde se describen los tipos que se usarán en la aplicación(Nombre de atributos, restricciones y creación de tablas)

Clases o paquetes creados

Paquete models, para las entidades

Paquete Web, para los controladores

Paquete WEB-INF, para los archivos jsp(vistas)

Ventajas alcanzadas al aplicar el patrón

Este patrón tiene como ventaja separar la lógica de negocio y las interacciones del usuario, que facilita el manejo de errores. Otra gran ventaja es que se puede dividir fácilmente el trabajo entre los miembros del equipo de desarrolladores. Si necesitamos cambiar algo en la lógica de la aplicación, al estar separada de la interfaz de usuario, no habría que modificar ambas partes, ya que son independientes.

Decisiones de diseño

Decisión 1: Versionado de tablas críticas

Descripción del problema:

Cuando 2 personas están a la vez modificando una tabla de la base de datos, puede que haya conflictos ya que alguien puede obtener los datos sin tener la última versión de la base de datos, ya que la otra persona está haciendo modificaciones y no se van a ver reflejadas en la vista del otro usuario

Alternativas de solución evaluadas:

Para abordar este problema, se ha decidido hacer uso del versionado que nos proporciona Spring, de manera que usando la anotación @version en la entidad que queremos versionar, se creará un registro con un número entero que representará la versión más reciente de la tabla, que irá autoincrementando a medida que se actualicen los datos que contiene. En caso de que alguien quiera modificar algo de la tabla, se comprobará mediante ese número de versión que coincide con la última versión actualizada de la base de datos. Si la versión no coincide, dará un error y evitaremos el conflicto de los datos mencionados en la descripción del problema

Justificación de la solución adoptada

Esta solución se ha elegido ya que es la que nos enseñan en clase, además de ser algo sencillo de implementar ya que se añade solamente la anotación y una condición en el controlador que comprueba la versión de los datos de la tabla que se va a modificar

Decisión 2: Logs

Descripción del problema:

Queremos tener un registro del flujo que sigue la aplicación desde que se inicia hasta que se detiene, pasando por todas las funcionalidades que tiene.

Alternativas de solución evaluadas:

Para tener dicho registro, optamos por usar los log, para ello, añadimos al archivo xml las dependencias, para que simplemente poniendo en el controlador la anotación @Slf4j, podemos usar log.info(“Mensaje del log”). Cada vez que en controlador se lleve a cabo una función descrita en los requisitos del sistema, se añade un mensaje de log para que quede reflejado en el archivo que se aloja en la carpeta logs de nuestro proyecto, con el nombre spring-boot-logger.log.

Justificación de la solución adoptada

Al igual que el versionado, es una decisión que tiene una implementación sencilla y hemos visto en clase, por lo que hemos decidido usarla. Además, nos ayuda a ver con claridad el flujo que sigue la aplicación durante su ejecución, como si usarámos el modo debug constantemente.e