Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 2
Obligatorio 1

Valentina Damasco 167467
Yuliana Gómez 186889

Docentes: Daniel Acevedo e Ignacio Valle
Repositorio: https://github.com/ORT-DA2/Damasco-Gomez

Entregado como requisito de la materia Diseño de Aplicaciones 2

15 de octubre de 2020

1. Descripción del diseño

1.1. Descripción general

El sistema desarrollado como requisito para la materia Diseño de Aplicaciones 2 es la implementación de una API REST para el Ministerio de Turismo que permite ayudar a turistas a poder encontrar el lugar que están buscando para vacacionar con las características que buscan. El sistema maneja usuarios turistas y administradores. Los turistas podrán buscar hospedajes para un punto turístico, realizar una reserva de un hospedaje y consultar el estado actual de una reserva. Los administradores tendrán permiso para agregar puntos turísticos, regiones y categorías, así como también filtrar aquellos hospedajes que ya tengan su capacidad completa para el período y cantidad de personas seleccionadas por el turista, cambiando la capacidad actual del hospedaje. Además podrán dar de alta un nuevo hospedaje y borrar uno existente para un punto turístico. Puede cambiar el estado de una reserva y realizar el mantenimiento de los administradores del sistema.

1.2. Errores conocidos

A continuación de describen los errores identificados para el sistema:

El sistema no maneja una clase exclusiva para las imágenes, la idea de esta clase era guardar el nombre de la imagen, la entidad de donde viene y el tipo de la misma. Por una cuestión de tiempos y de priorización de requerimientos se decidió que este requisito se implementará para una futura entrega.

El usuario tiene nombre como atributo. Este error fue detectado muy cerca de la fecha de la entrega con cual para que no impacte negativamente sobre la cobertura, tests, etc se arreglará para una próxima versión del sistema.

Diagrama general de paquetes

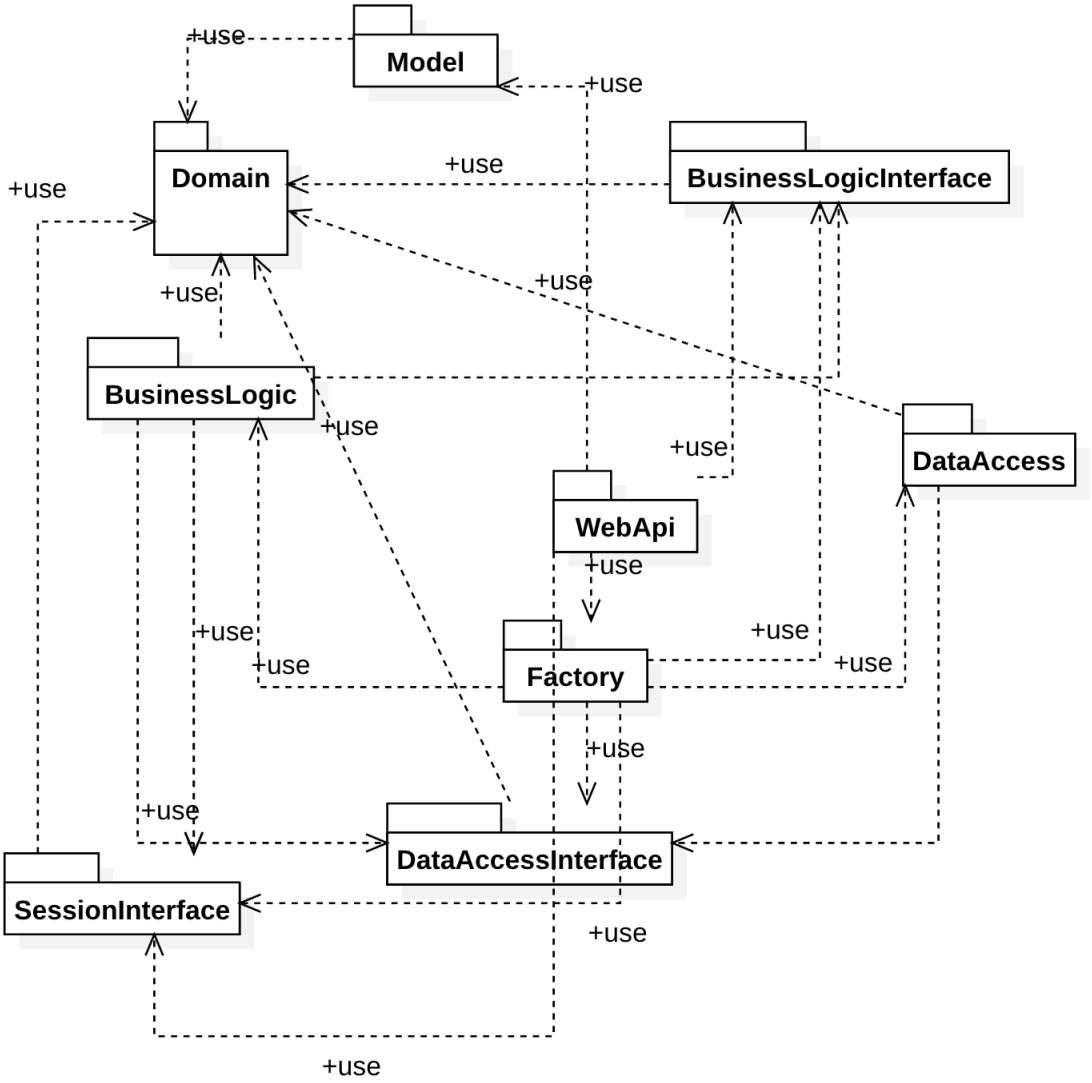

Figura 1.1: Package

Descripción de responsabilidades

En nuestro sistema se pueden distinguir 14 paquetes:

1. **WebApi** : Se encarga de definir los endpoint y procesar las request de los clientes. Aquí además tenemos dos paquetes más con distintas responsabilidades , uno que contiene los controladores y otro que contiene los filtros para el sistema.

2. **WebApiTest :** Contiene los test para los controladores de WebApi.

3. **SessionInterface :** Paquete que contiene la interfaz para la sesión del usuario.

4. **Migration:** Este paquete es el encargado de realizar las migraciones de mis entidades de dominio hacia la base de datos.

5. **Model:** Este paquete contiene los modelos de entrada y salida, separados en subpaquetes In y Out que se usan para la transferencia de datos entre la WebApi y el usuario.

6. **Factory:** Este paquete es el encargado de crear las dependencias en el contenedor de servicios.

7. **Domain:** Paquete que contiene las entidades del sistema.

8. **DomainTest :** Paquete que contiene las pruebas unitarias para las entidades del sistema.

9. **DataAccess:** Dentro de este paquetes tenemos 3 subpaquetes : Repositories, el cual contiene los repositorios de las entidades.Se encarga de implementar las interfaces existentes en DataAccessInterface
   Context, contiene una sesión con la base de datos , con las entidades guardadas y se utiliza para realizar las consultas y por último Migrations, que guarda las migraciones realizadas a la base y un historial de las mismas.

10. **DataAccessInterface:** Paquete de interfaces, define el contrato de los datos de la aplicación.

11. **DataAccess.Test:** Paquete que contiene las pruebas de las clases de DataAccess.

12. **BusinessLogic:** Este paquete define la lógica de negocio e implementa las interfaces definidas en BusinessLogicInterface.

13. **BusinessLogicInterface :** Paquete con interfaces , define el contrato de la lógica de negocio.

14. **BusinessLogicTest:**Contiene los test unitarios de las entidades de negocio.

Los paquetes fueron creados con el propósito de que nuestro sistema tenga bajo acoplamiento y una sola responsabilidad por paquete.

1.3. Paquete lógica de negocio

Para no romper con el Principio de Inversión de Dependencias de SOLID, que establece que paquetes de alto nivel no dependan de paquetes de bajo nivel creamos el paquete BusinessLogic. El mismo tiene la responsabilidad de de obtener , crear , actualizar, borrar y comparar contra la base de datos. Este paquete no se conecta directamente con DataAccess sino que es de interfaces. Logrando un código mas fácil de testear,abierto a la extensión y cerrado a la modificación

1.3.1. Diagramas de paquetes de interfaces

El siguiente diagrama de paquetes contiene a las interfaces que son implementadas por BusinessLogic. Cumpliendo con el Principio de Segregación de la Interfaz que establece SOLID.

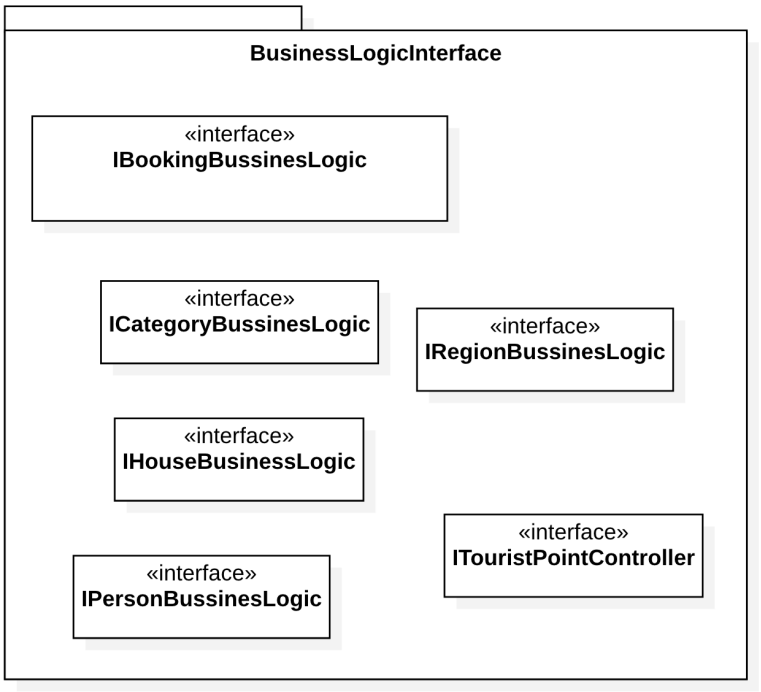

Figura 1.2: BusinessLogic Interface

1.3.2. Diagrama de clase

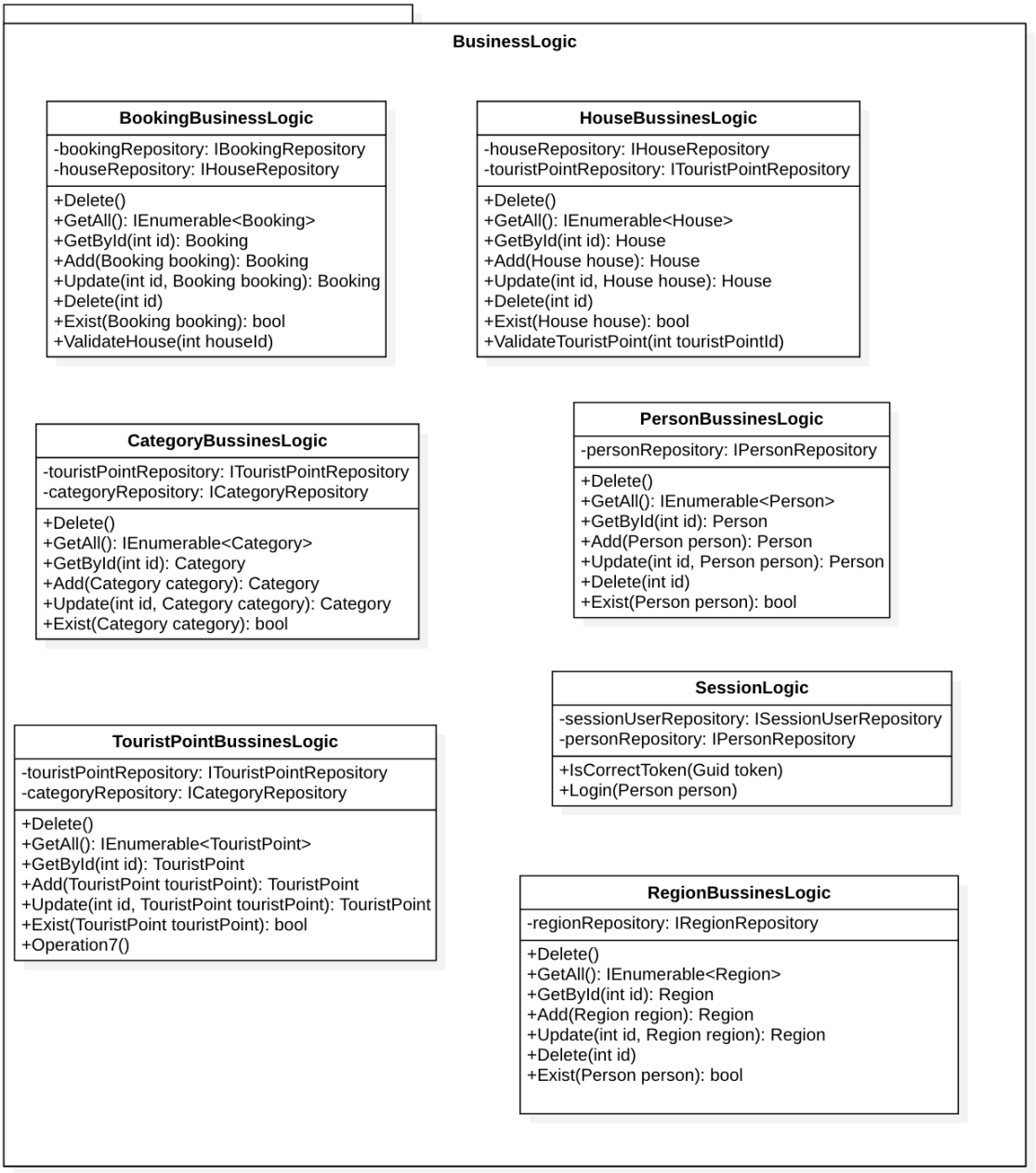

Figura 1.3: BusinessLogic

1.4. Paquete DataAccess

1.4.1. Paquete Context

Nuestro contexto llamado VidlyContext hereda de DbContext y es el encargado de representar la sesión de la base de datos. Sabemos que este nombre no es del todo nemotécnico pero generaba complejidad cambiarlo en el sistema después de haberlo llamado así. Utilizamos carga LazyLoading de Entity Framework porque tiene la ventaja de que a la hora de realizar consultas sobre una cierta entidad no se obtienen los objetos relacionados.

Diagrama de clase

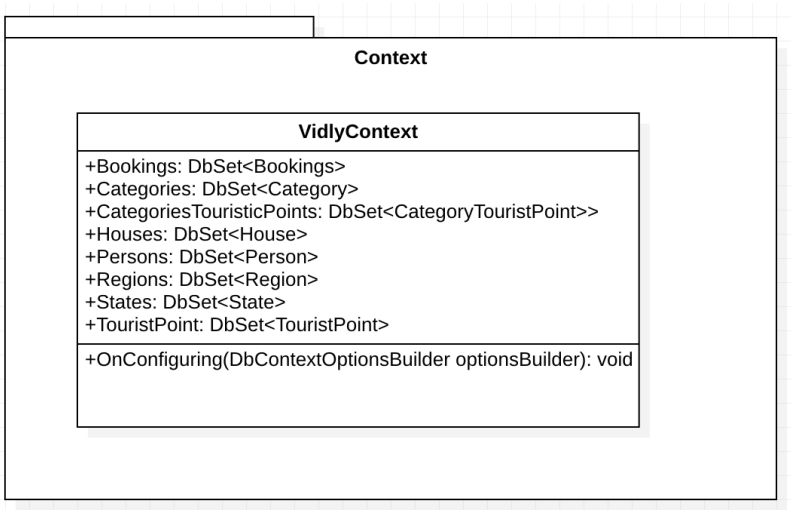

Figura 1.4: Context

1.4.2. Paquete de repositorios

Diagrama de clase

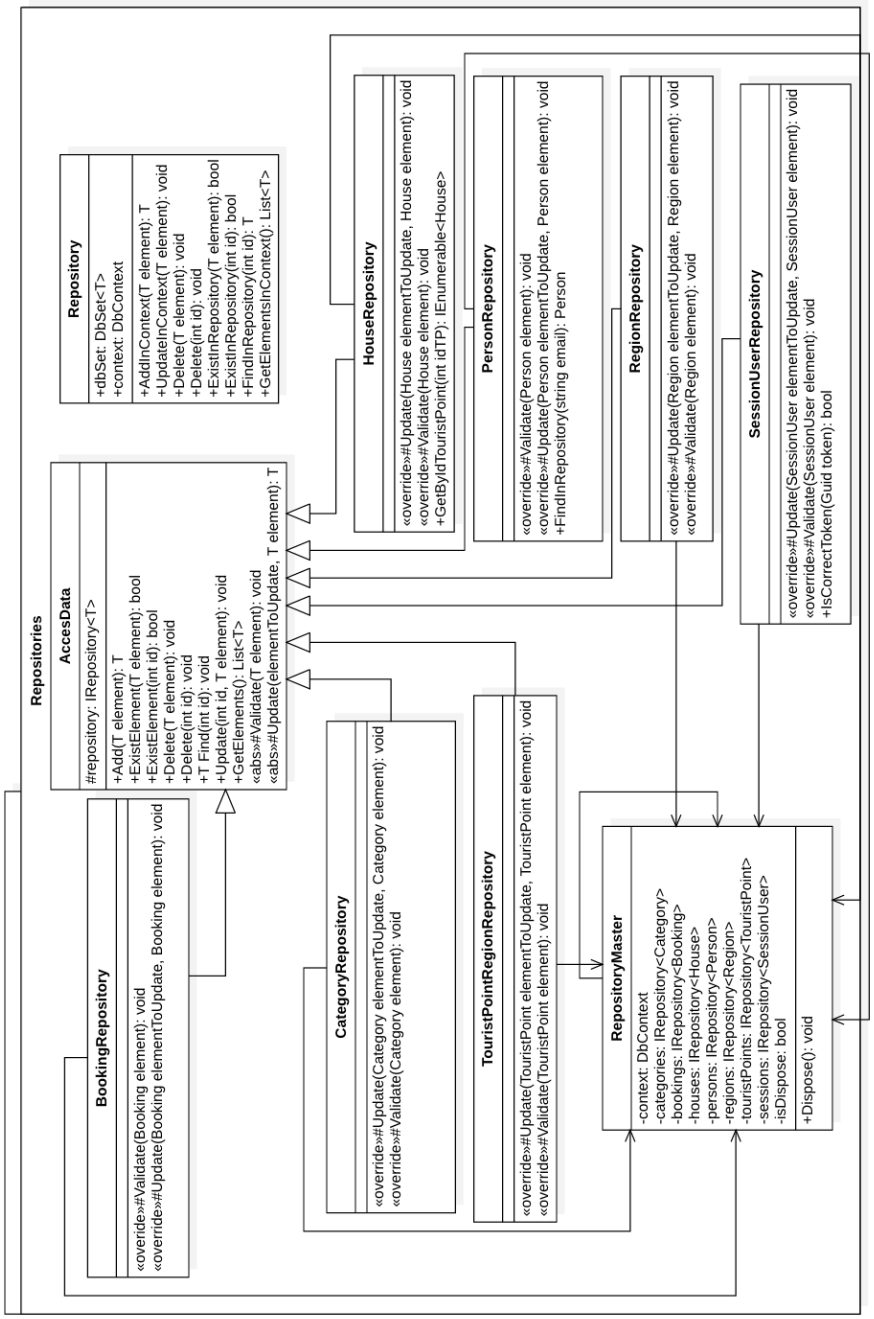

Figura 1.5: DataAccess

En cuanto al acceso a datos, cada repositorio hereda de una interfaz y de la clase genérica AccessData. Esta decisión fue tomada para que el acoplamiento sea de interfaz y no de implementación y además la clase genérica AccessData fue creada con el objetivo de reutilizar código y tiene la responsabilidad de manejar los repositorios. La clase Repository genérica fue creada para implementar las funciones de agregar,

actualizar, borrar, buscar y chequear existencia de los elementos del contexto de forma genérica.Y además Abre la conexión con el contexto. La clase RepositoryMaster conecta todos los repositorios. Funciona como singletón.Asimismo Tiene todos los repositorios.

Diagrama de paquetes DataAccessInteraface

Este paquete fue creado para crear acoplamiento de interfaz y no de implementación. DataAccess usa DataAccessInterface ya que el mismo se encarga de implementar las interfaces de este paquete. La interfaz IAccessData es una clase genérica de cúal heredan el resto de las interfaces. IRepository es el contrato de la clase Repository descrita anteriormente.

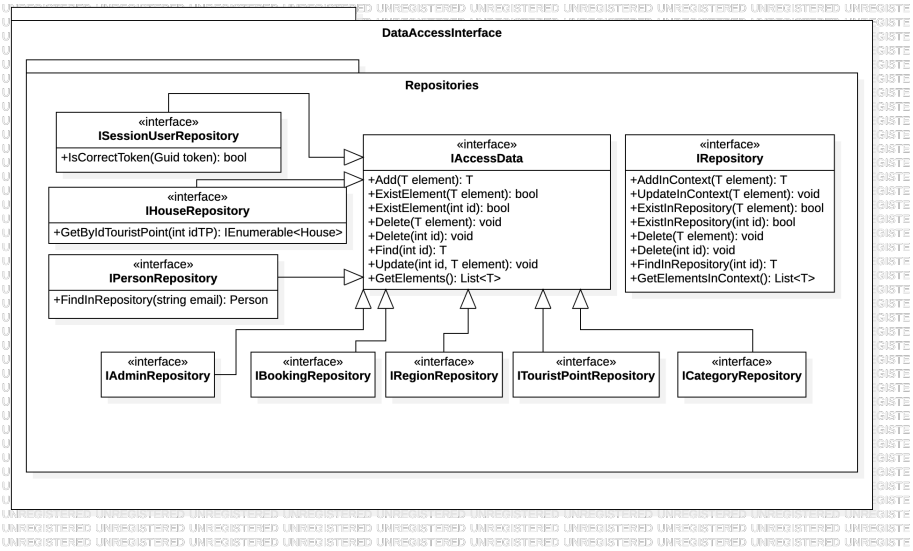

Figura 1.6: DataAccessInterface

1.5. Paquete de entidades del sistema

En el siguiente paquete se representan las entidades de nuestro sistema.

1.5.1. Diagrama de clases

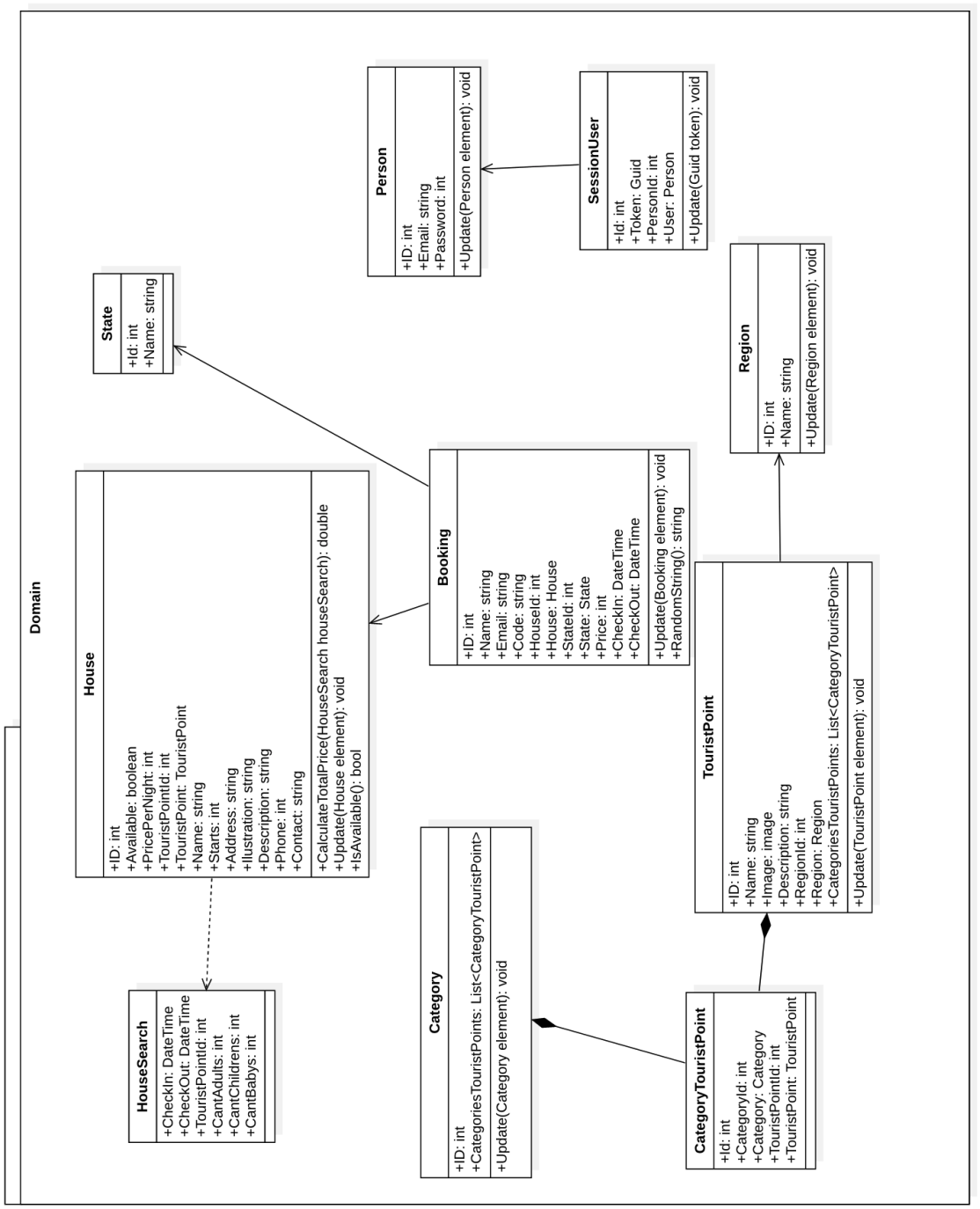

Figura 1.7: Domain

1.6. Paquete de servicios

Utilizamos un contenedor de servicios para inyectar las dependencias que usa nuestro sistema. Para que cuando alguien solicite el servicio inyectado aquí, el mismo crea las instancias y se lo pasa a quien lo solicita y además se encarga de descartar los elementos cuando no se utilicen más. De esta menar buscamos lograr que las clases solo se concentre en utilizar las dependencias que están en el contenedor y no en configurarlas. El paquete Factory se encarga de esto.

1.6.1. Diagrama de clases

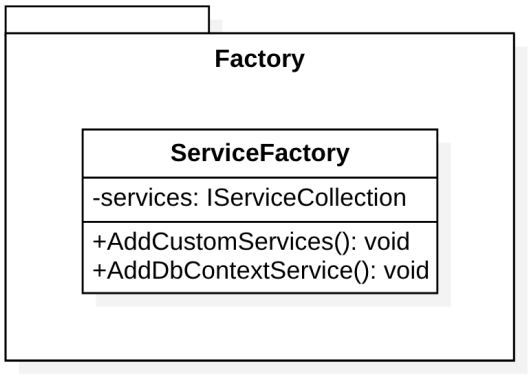

Figura 1.8: Factory

1.7. Paquete WebApi

Si bien WebApi contiene tanto el paquete Controllers como el paquete Filter, como son independientes para mantener la cohesión los representamos separados.

1.7.1. Paquete de Controladores

Los controladores de la WebApi heredan de VidlyControllerBase el cual es nuestro controlador base. Se tomó esta decisión de diseño de manera tal de evitar la duplicación de código innecesaria y que los controladores de la WebApi puedan manejar las requests HTTP de los clientes. Los controladores de WebApi solo se encarga de definir los endpoints y procesar las request de los clientes.

Diagrama de clases

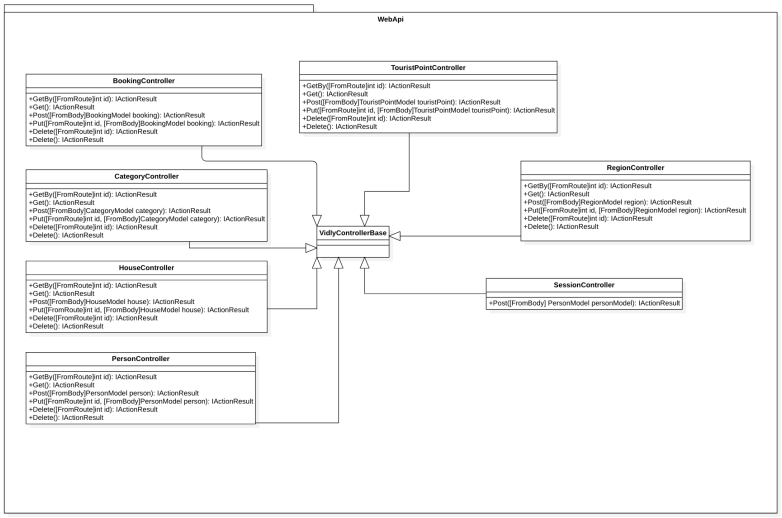

Figura 1.9: WebApi

1.7.2. Paquete de Filtros

En este paquete se muestran los filtros utlizando en nuestro sistema. Los mismos son dos : AuthorizationFilter y ExceptionFilter que heredan de Attribute, IAuthorizationFilter y IExceptionFilter respectivamente. Dichas interfaces y la clase Atribute son provistas por EF. El filtro de autorización maneja una ISessionLogic. Es decir la lógica de la sesión. Aquí se establece que el header de las requests deben contener la autorización correspondiente para realizar ciertas acciones tales como POST, PUT Y DELETE. Esta decisión fue tomada que el sistema sea seguro. Luego la clase ExceptionFilter es la encargada de catchear todas las excepciones del sistema. Si bien ambas clases están en el paquete de filtro al no estar vinculadas una con otra las representamos de forma separada para mantener la cohesión.

Diagrama de clases

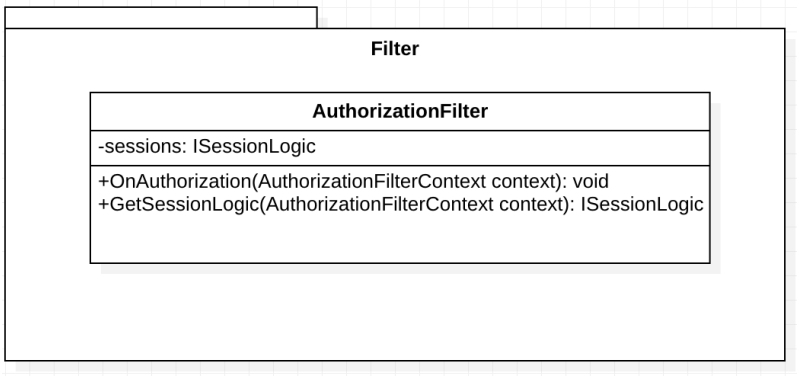

Figura 1.10: AuthorizationFilter

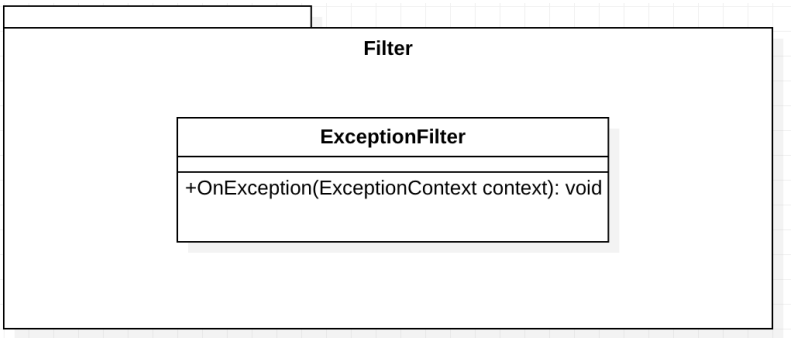

Figura 1.11: AuthorizationFilter

1.7.3. Paquete de Modelos

Para lograr seguridad en nuestro sistema utilizamos modelos de entrada y salida de datos entre el cliente y la WebApi. Estos objetos son Dummy. De esta manera no exponemos la información de nuestras entidades. Con la realización de estos modelos aplicamos el patrón Adapter. El impacto de cambio se vuelve menor , ya que si cambia Domain se vería afectado solo Model y no los controladores de WebApi.

Diagrama de clases

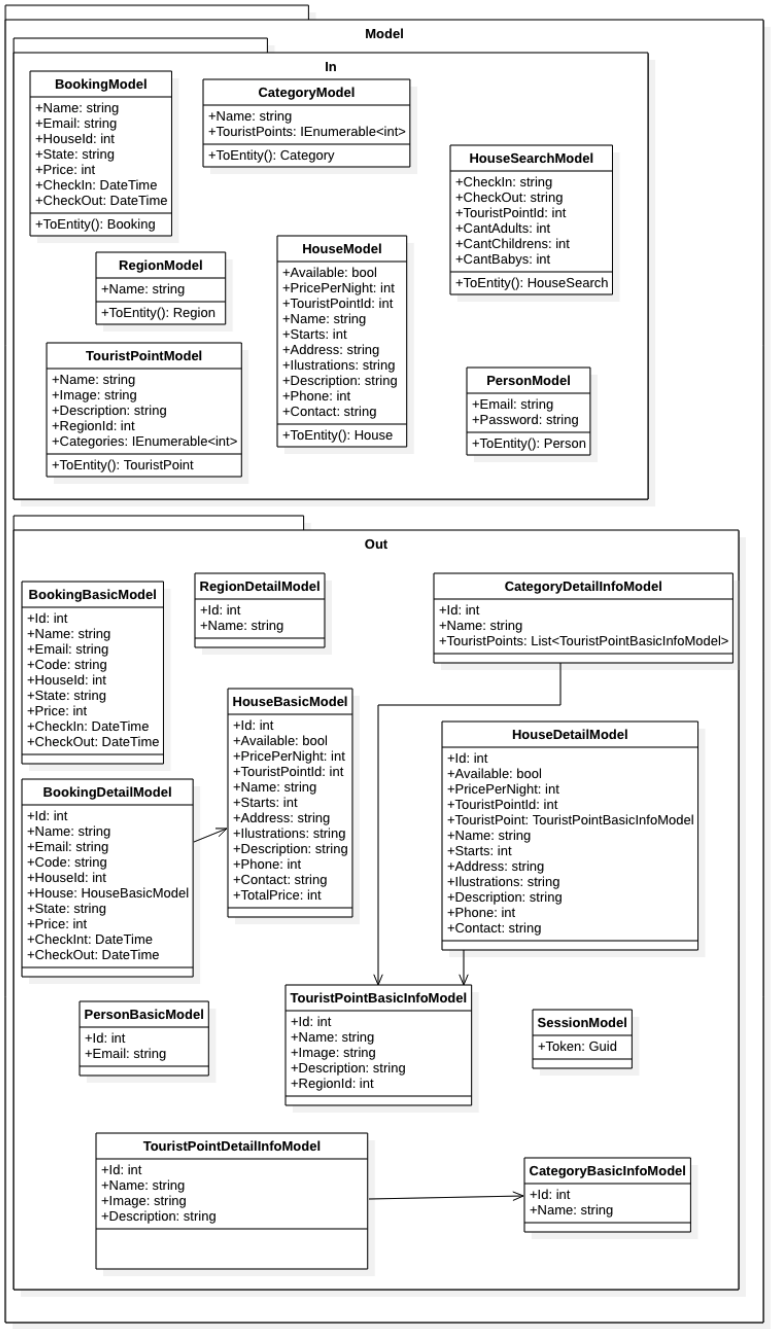

Figura 1.12: Model

Descripción y justificaciones de diseño

A continuación describiremos las decisiones de diseño que fuimos tomando a lo largo de este proyecto ajustándonos a las buenas prácticas de programación y a nuestros conocimientos para lograr un sistema flexible y abierto a la extensión.

- Utilizamos el patrón inyección de dependencia para desacoplar nuestra capa de negocio(BussniessLogic) de nuestro acceso a datos (DataAccess) mediante la utlización de interfaces.

- En lo que respecta WebApi la misma mantiene una dependencia a Busniness-LogicInterface, de manera que se acopla a una interfaz y no a una implementación.

- Asimismo hicimos uso del patrón Fachada que se puede ver por la implementación de nuestra api , que hace que el cliente que va a consumir nuestros recursos se deslinde de la implementación.

- Manejamos además autenticación de usuarios para lograr seguridad en nuestro sistema. Ciertas acciones como POST, PUT y DELETE requieren de autenticación de administrador para poder realizarse. Decidimos que el login del usuario lo maneje la lógica de la Sesión quién es el experto en la información este caso. Haciendo uso del patrón experto.

- Por otra parte determinamos resolver la relación N a N que tiene nuestro sistema entre categorías y puntos turísticos con una tercer clase Category-TouristPoint,está decisión fue tomada para acoplarnos a la tecnología Entity Framework.

- Las regiones, categorías y los estados de la reserva los mapeamos como entidades en el sistema con sus respectivas tabla en la base de datos pretendiendo ajustarnos lo mejor posible a un sistema extensible.

- Para la entidad hospedajes mapeada en clase House, como el boolenao de disponiblidad de la misma tiene una alta probabilidad de cambiar, decidimos crear un método IsAvailable de manera que cambie solo este método a futuro y descapolarnos del booleano, con este comportamiento buscamos que si la lógica cambia sea más fácil de ejecutar el cambio rompiendo lo menos posible.

- Para la función de calcular el precio total de un hospedeja en base a los datos que ingresa el usuario decidimos colarla en la clase experta en la información, es decir la clase del dominio House. Haciendo uso nuevamente del patrón experto.

- Los test fueron separados en paquetes diferenciados para cumplir SRP.

- Para el primer requerimiento funcional "Búsqueda de puntos turísticos por región y por categoría", decidimos no realizar un endpoint específico en el backend ya que esto será responsabilidad del frontend que lo resolverá más adelante realizando un filtrado.

- Para el manejo de excepciones decidimos crear una clase que tenga esta única responsabilidad. La misma es ExceptionFilter como se ha mencionado anteriormente.

1.8. Diagramas de secuencia y colaboración de las funcionalidades claves del sistema

RF1: Elegir un punto turístico y realizar una búsqueda de hospedajes

1.8.1. Diagrama de secuencia RF1

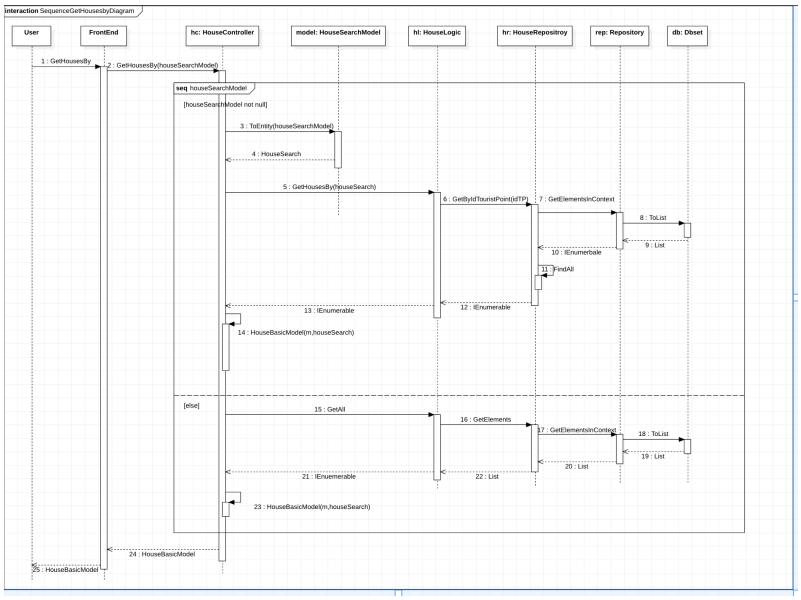

Figura 1.13:

1.8.2. Diagrama de comunicación RF1

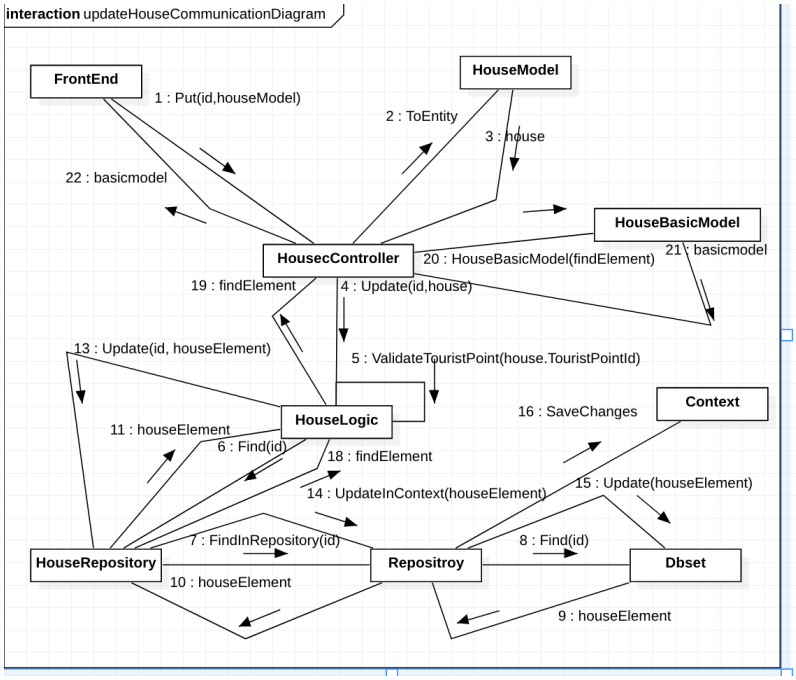

Figura 1.14:

RF2: Realizar una reserva de un hospedaje.

1.8.3. Diagrama de secuencia RF2

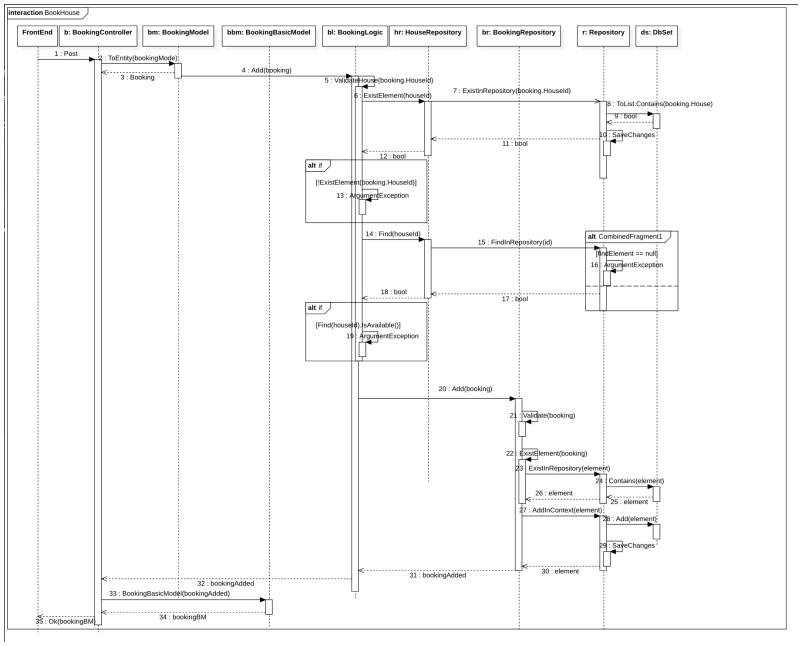

Figura 1.15:

1.8.4. Diagrama de comunicación RF2

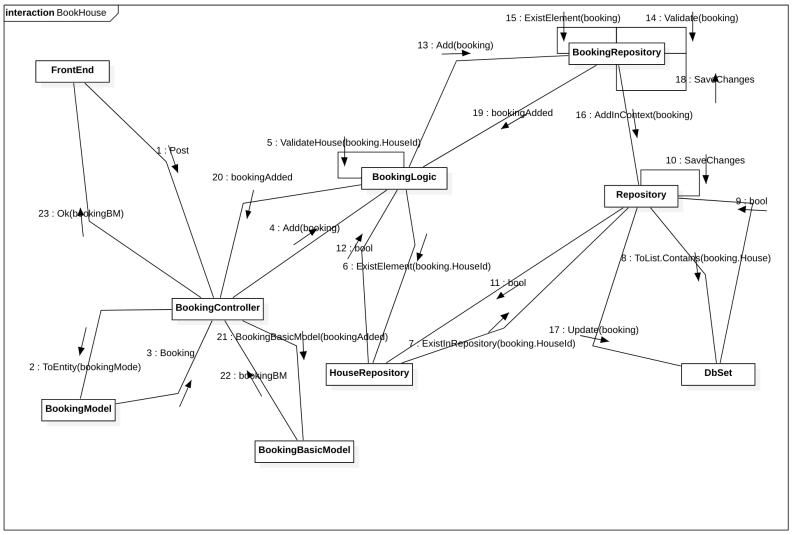

Figura 1.16:

RF3: Dar de alta un nuevo hospedaje para un punto turístico existente.

1.8.5. Diagrama de secuencia RF3

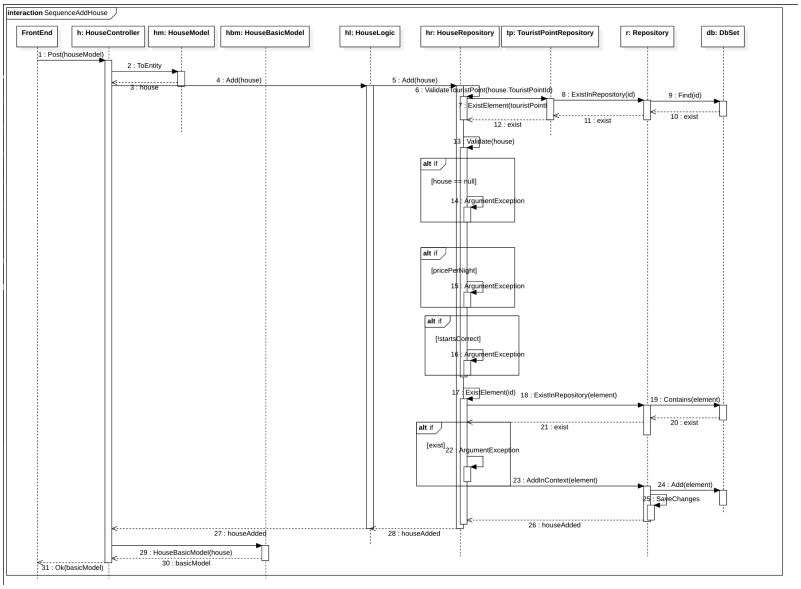

Figura 1.17:

1.8.6. Diagrama de comunicación RF3

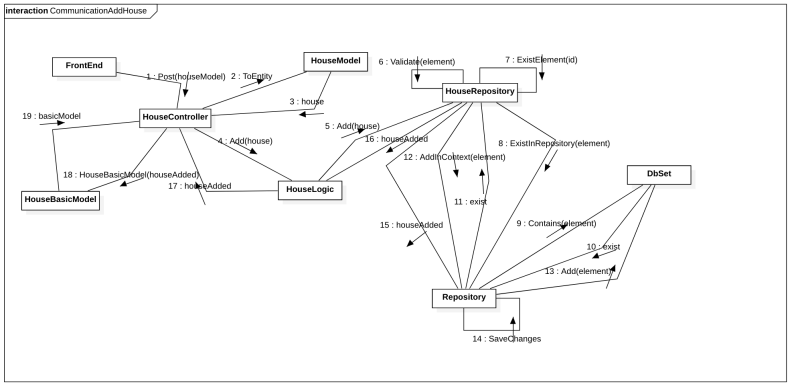

Figura 1.18:

RF4: Modificar la capacidad actual de un hospedaje.

1.8.7. Diagrama de secuencia RF4

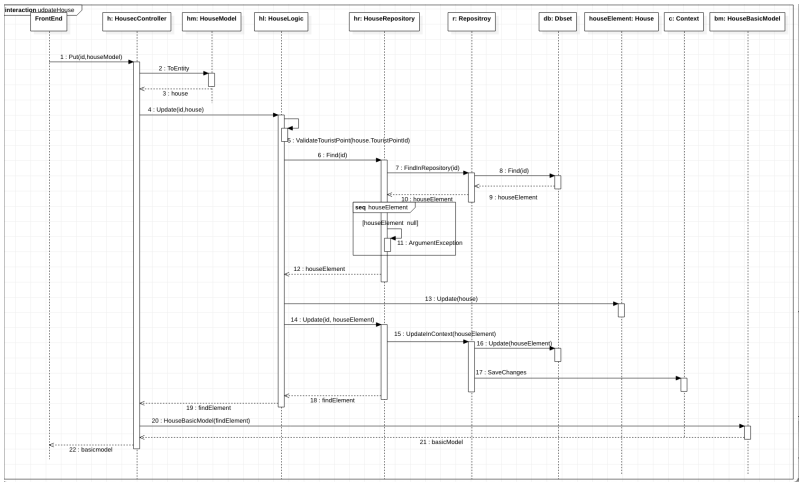

Figura 1.19:

1.8.8. Diagrama de comunicación RF4

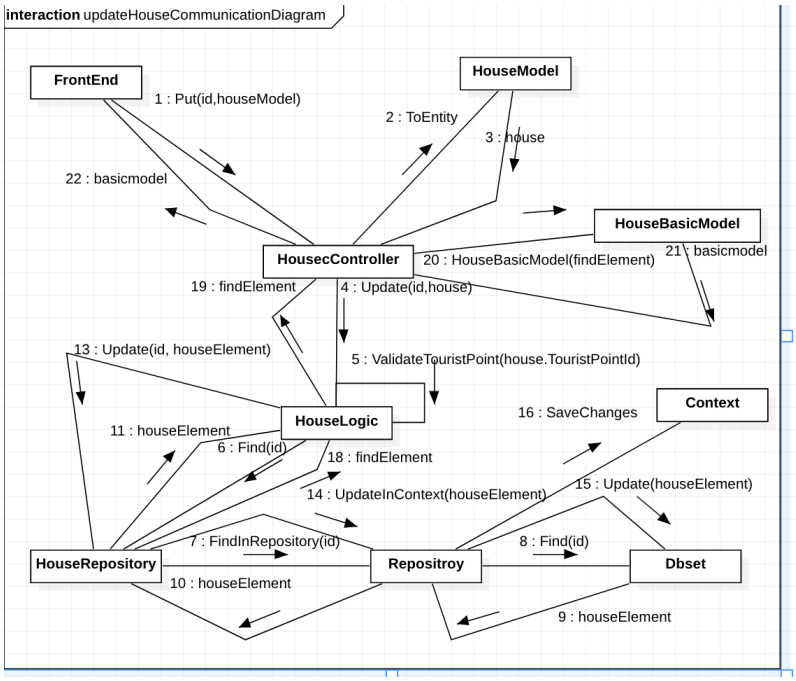

Figura 1.20:

RF5: Cambiar el estado de una reserva, indicando una descripción.

1.8.9. Diagrama de secuencia RF5

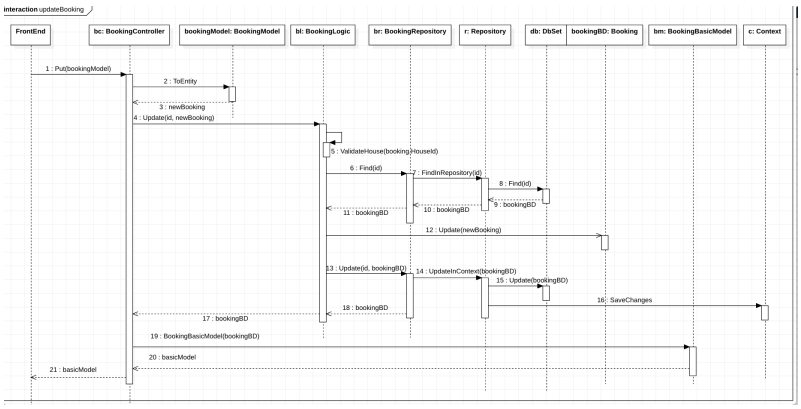

Figura 1.21:

1.8.10. Diagrama de comunicación RF5

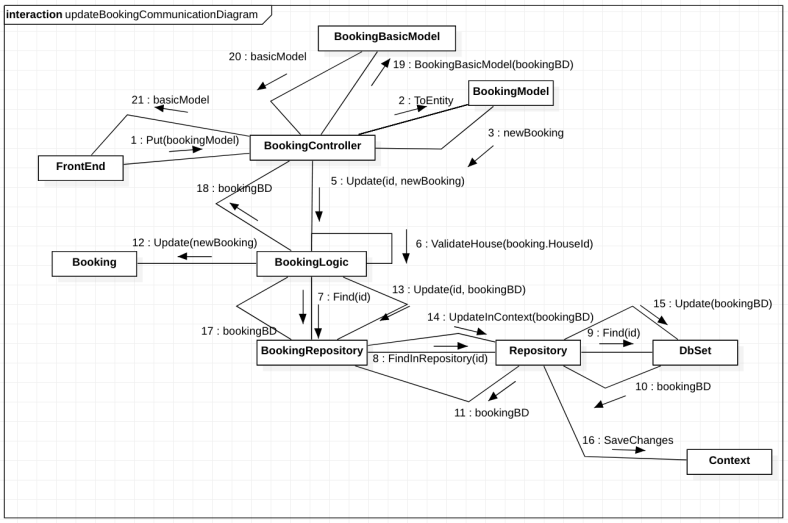

Figura 1.22:

RF6 : Búsqueda de puntos turísticos por región y por categoría:

1.8.11. Diagrama de secuencia RF6

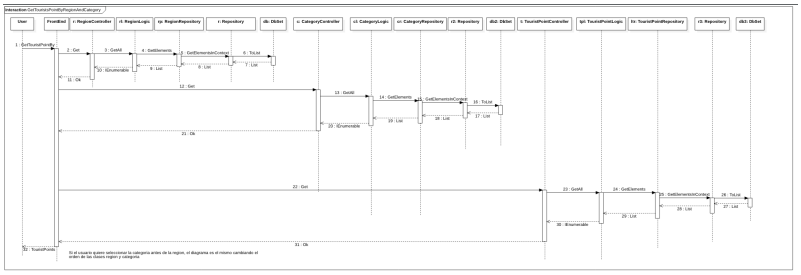

Figura 1.23:

1.9. Diagrama de componentes

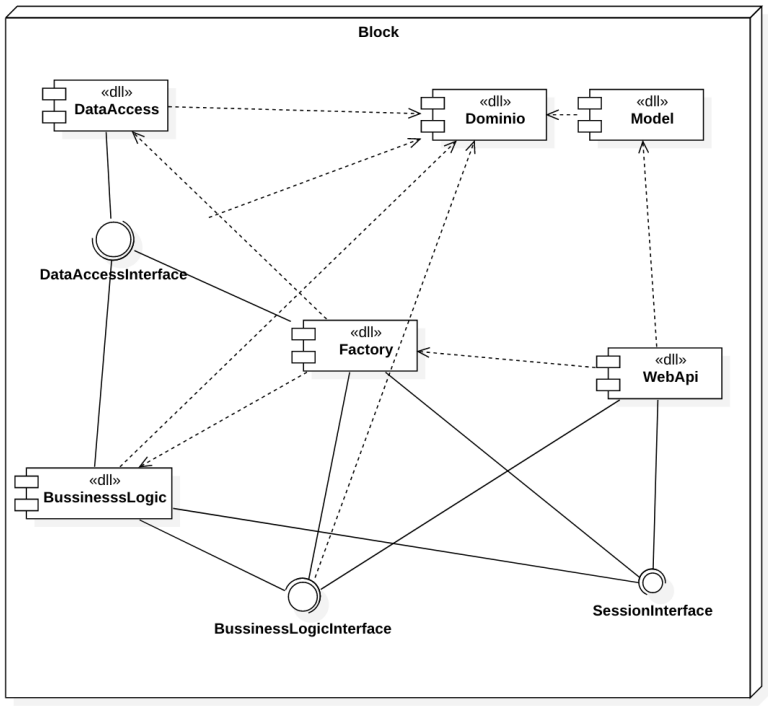

Figura 1.24: Diagrama de componentes

1.10. Diagrama de namespaces

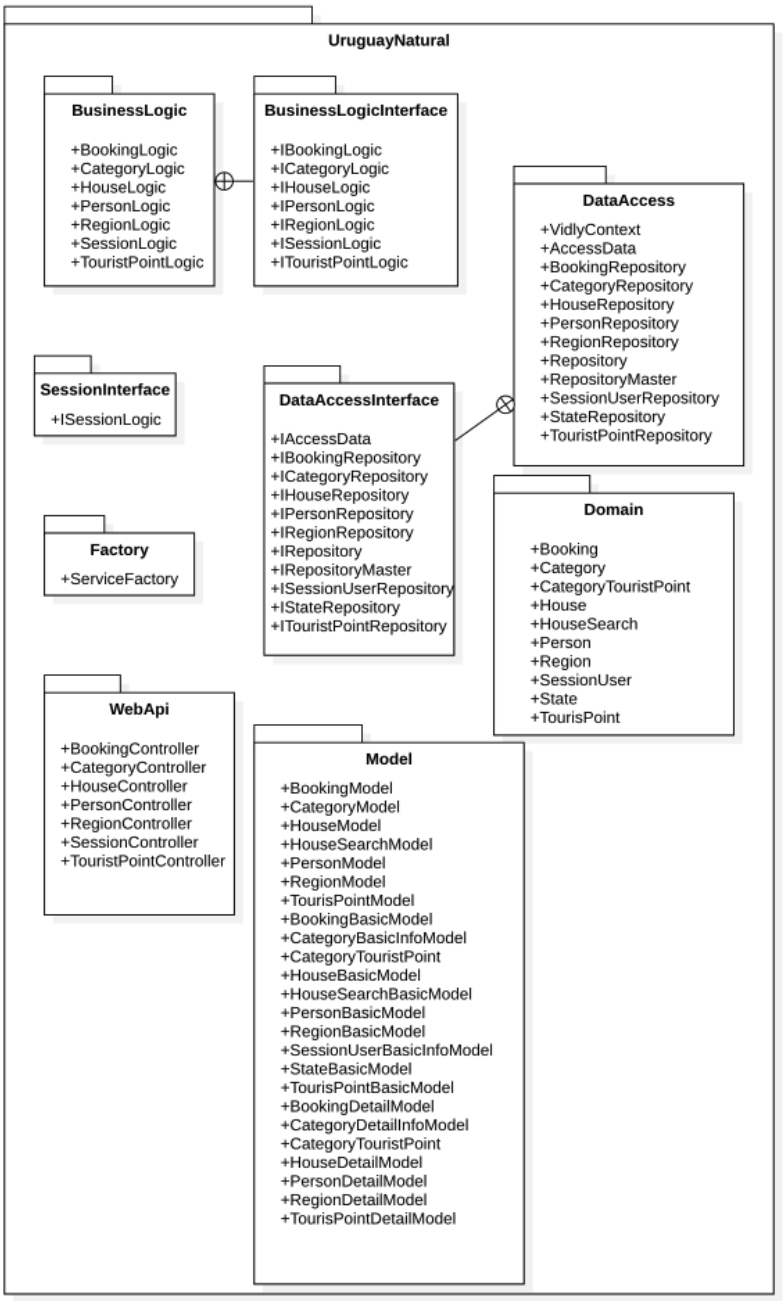

Figura 1.25: Diagrama de namespaces

1.11. Modelado de la tabla

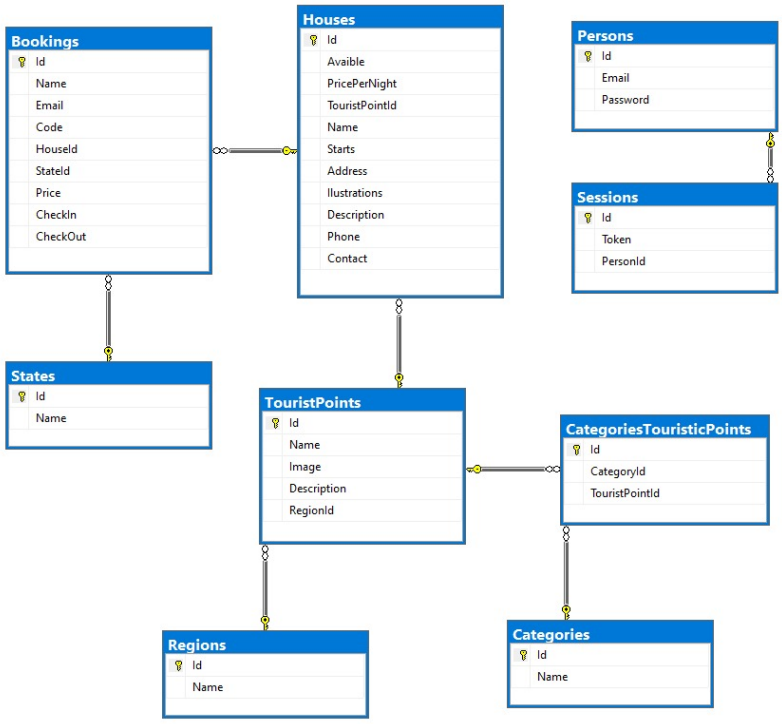

Figura 1.26: Modelado de las tablas