Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de aplicaciones II
Obligatorio I
Descripción del diseño

Juan Pablo Barrios - 206432
Juan Ignacio Irabedra - 212375

Entregado como requisito de la materia Diseño de aplicaciones 2

15 de octubre de 2020

Contents

1. Introducción:

A lo largo de este documento presentaremos el diseño de la aplicación que entregamos como obligatorio 1 de diseño de aplicaciones 2, junto con las correspondientes justificaciones de todas aquellas decisiones que consideramos así lo ameritan.

El proyecto responde a la necesidad de actualizar el sitio web de la marca *Uruguay Natural* del Ministerio de Turismo, enfatizando la experiencia *end-to-end* con el usuario. El mismo ofrece las siguientes funcionalidades descritas en términos generales:

- Búsqueda de puntos turísticos por región y por categoría
- Elegir un punto turístico y realizar una búsqueda de hospedajes
- Dado un hospedaje, realizar una reserva
- Registrar administradores, los cuales deben poder:
  - Iniciar sesión con mail y contraseña
  - Dar de alta un punto turístico para una región existente
  - Alta y baja de hospedajes
  - Modificar capacidad actual de hospedaje
  - Cambiar estado de reserva, indicando descripción
  - Alta, baja y modificación de otros administradores

Estos puntos serán cubiertos inidivudalmente más adelante en el documento a los efectos de dar evidencia de la realización de los mismos o dar cuenta de la no implementación en caso que corresponda.

El proyecto fue desarrollado usando C# como lenguaje, siendo destinados a .NETCore3.1 las aplicaciones de consola, WebApi y proyectos de pruebas unitarias que usan MSTest. La persistencia se realiza mediante Entity Framework Core.

2. Requerimientos funcionales:

A continuación se listan los requerimientos que relevamos, seguidos de una cruz si no han sido implementados, un tick si efectivamente están funcionando.

- Búsqueda de puntos turísticos por región y por categoría ✓
- Elegir un punto turístico y realizar una búsqueda de hospedajes ✓
- Dado un hospedaje, realizar una reserva ✓
- Registrar administradores ✓, los cuales deben poder:
  - Iniciar sesión con mail y contraseña ✓
  - Dar de alta un punto turístico para una región existente ✓
  - Alta ✓y baja de hospedajes X
  - Modificar capacidad actual de hospedaje ✓
  - Cambiar estado de reserva, indicando descripción ✓
  - Alta ✓, baja X y modificación de otros administradores X

2.1 Bugs conocidos:

Mencionaremos los bugs conocidos en el sistema. Estos también se pueden ver en el video de pruebas funcionales.

- En el endpoint POST de Accommdation en AccommodationController, cuando invocamos el método, el retorno del CreatedAtRoute en el caso exitoso, se llama a una excepción no controlada la cual se puede ver en Postman. Sin embargo, efectivamente se registra la Accommodation pedida.
- En el endpoint GET de TouristSpot, al cual se le pueden enviar Categorias y una Region para buscar, la búsqueda es correcta en todos los casos excepto cuando solo se busca Region. Es decir, si no se le indica ninguna categoría, la búsqueda devuelve siempre vacío.

Esperamos poder corregir esto para la próxima entrega.

3. Descripción de diseño:

3.1 Vista lógica:

En esta sección introduciremos las clases involucradas en nuestra solución individualmente y un diagrama que introduzca las capas que definimos para poder dar preámbulo a la vista de procesos. También mostraremos un diagrama de estructura compuesta de algunos pocos objetos de nuestro dominio.

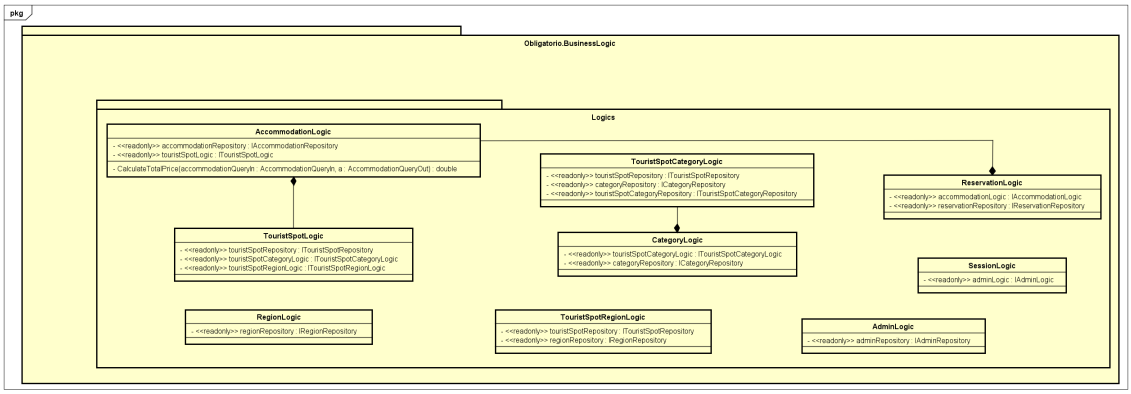

Figure 3.1: Diagrama de clases de BusinessLogic

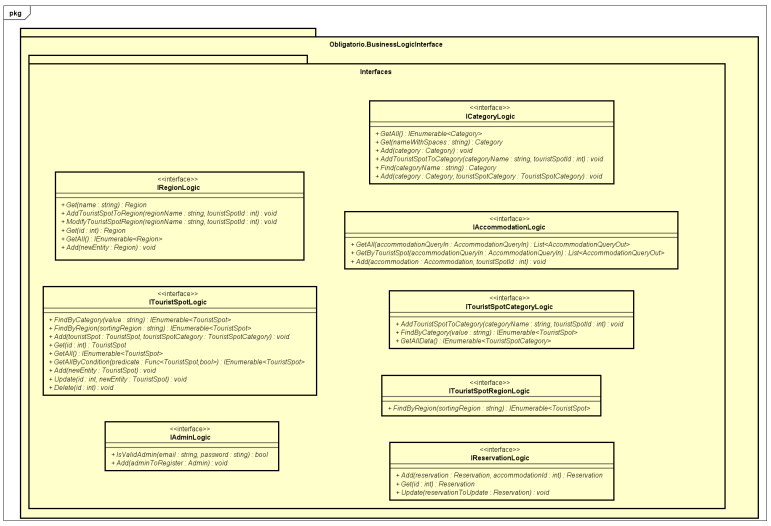

Figure 3.2: Diagrama de clases BusinessLogicInterface

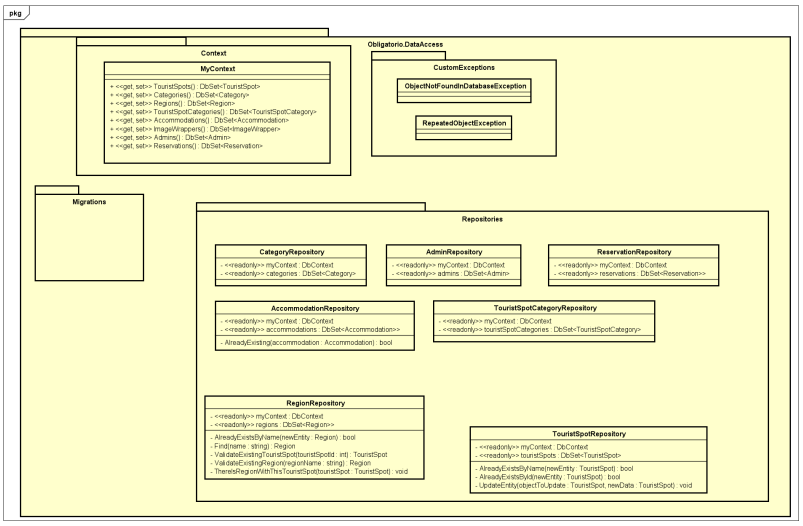

Figure 3.3: Diagrama de clases de DataAccess

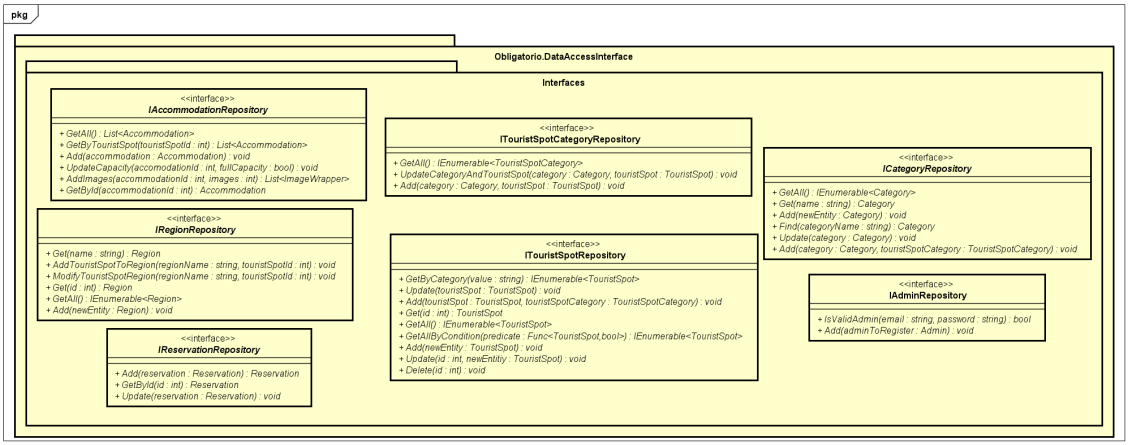

Figure 3.4: Diagrama de clases de DataAccessInterface

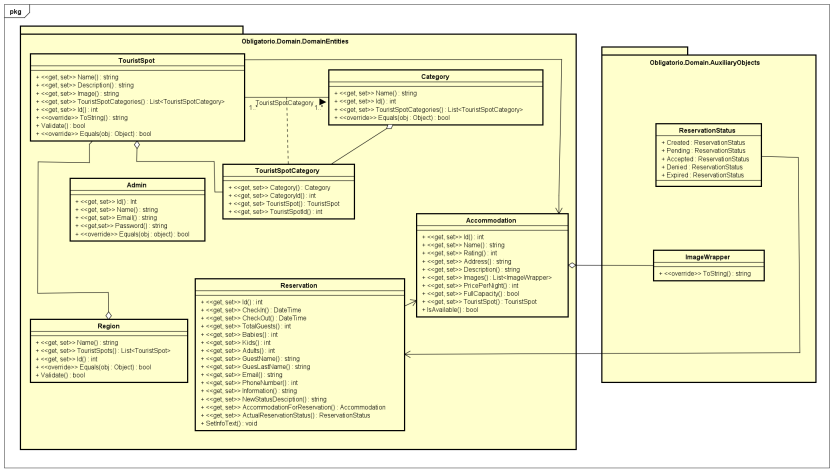

Figure 3.5: Diagrama de clases de Domain

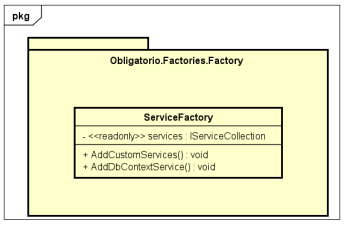

Figure 3.6: Diagrama de clases de Factory

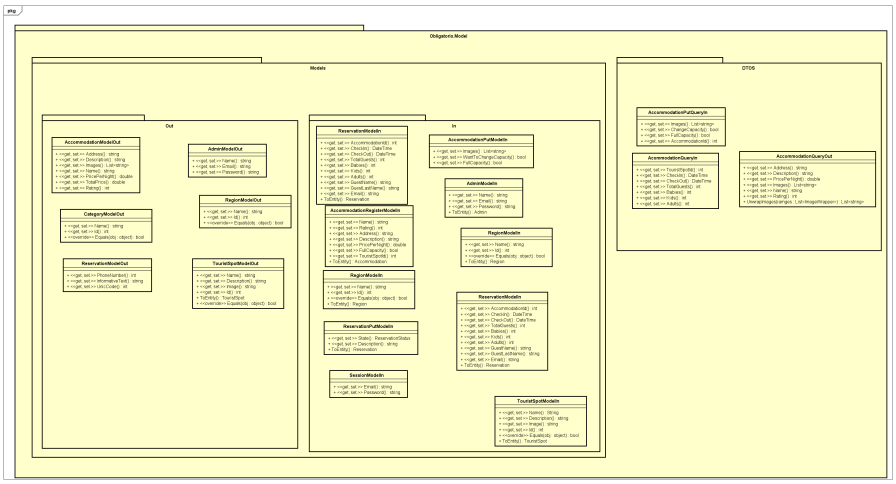

Figure 3.7: Diagrama de clases de Model

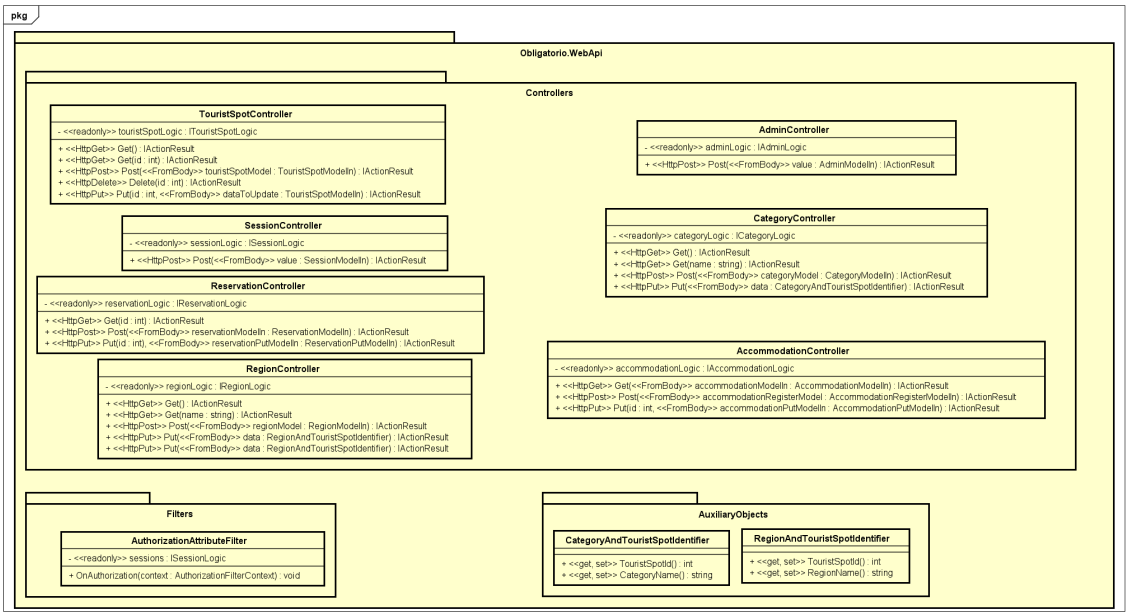

Figure 3.8: Diagrama de clases de WebApi

No vamos a ahondar en la justificación formal de las responsabilidades de paquetes ya que, hasta el día de hoy, hemos leído poco respecto a principios de paquetes. La idea fue dejar juntas clases que tengan razones de cambio parecidas, tengan niveles de abstracción similares y siguiendo lo que hemos visto en clase.

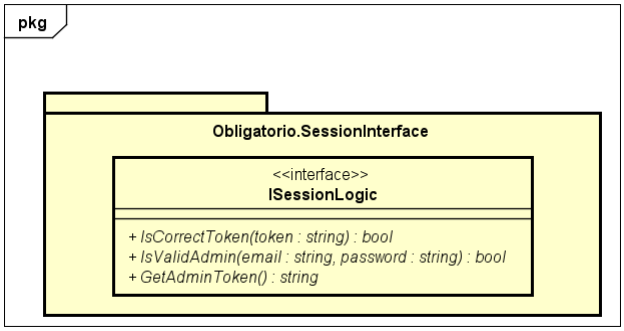

Figure 3.9: Diagrama de clases de SessionInterface

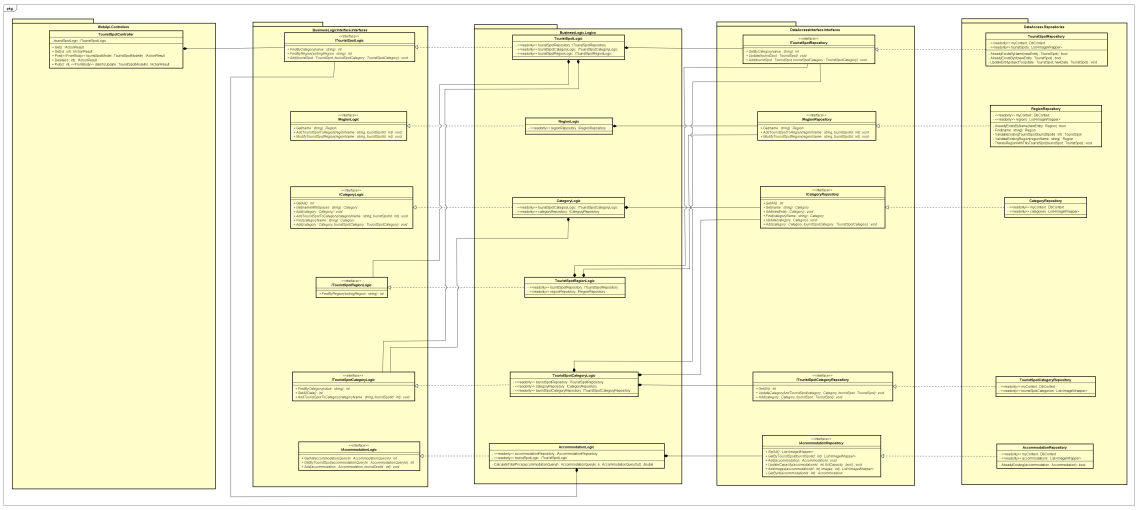

Figure 3.10: Diagrama de clases por capa

Ahora bien, elegimos dos objetos que colaboran entre ellos, *Reservation* y *Accommodation* para mostrar su estructura interna. Lo haremos mediante un diagrama de estructura compuesta:

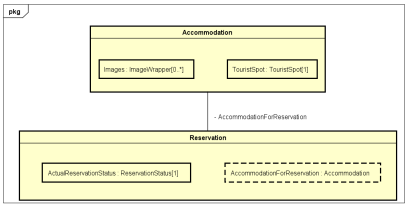

Figure 3.11: Diagrama de estructura compuesta

Las responsabilidades de las clases fueron asignadas mediante GRASP. También intentamos respetar principios SOLID. Creemos que hemos cumplido con lo mencionado, a excepción del SRP e ISP. Los contratos de nuestras clases son amplios, y al tener varios clientes, estos contratos de exponen de manera completa. Se podría haber reducido el contrato de cada clase mediante fabricación pura, pero no quisimos complejizar el diseño demasiado. En particular porque nos atrasamos mucho

acostumbrándonos a las nuevas tecnologías. También podríamos haber segregado múltiples interfaces para una misma clase, de modo que sus clientes puedan ver solamente lo que les interesa: pero ese refactor no pudo ser posible por tiempo.

3.2 Vista de procesos:

En esta sección introduciremos un par de diagramas de secuencia. El objetivo de estos diagramas es mencionar brevemente cómo se manejaron las excepciones y cómo funciona el mecanismo de persistencia a través de las capas que se pueden ver en el diagrama de clase por capas ubicado en la sección de vista lógica.

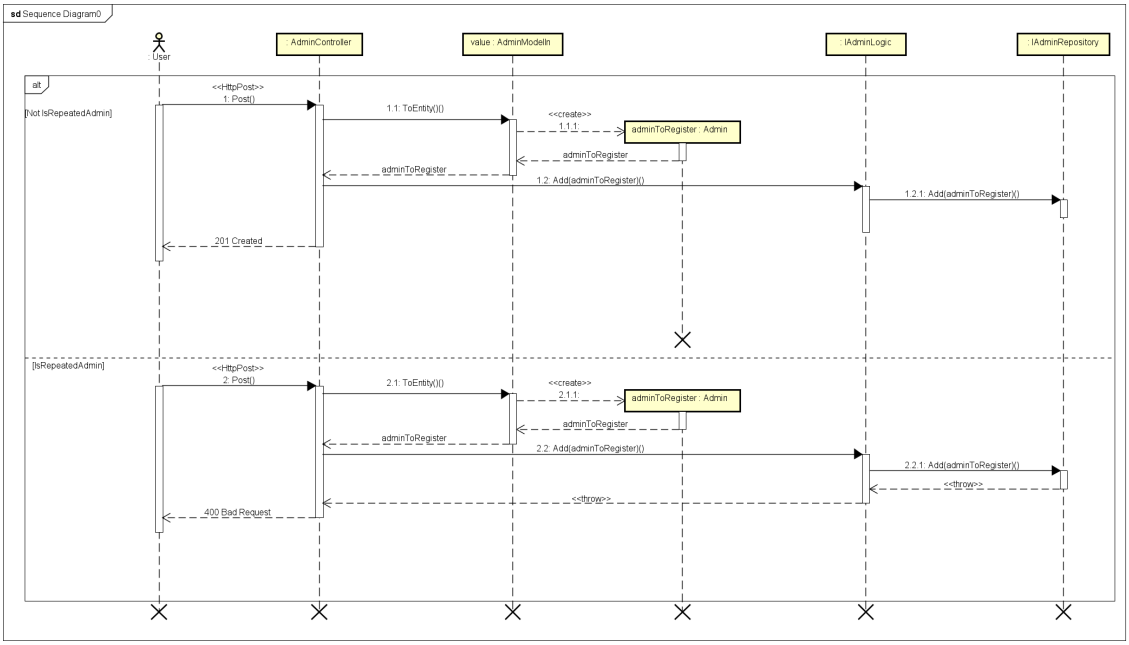

Figure 3.12: Secuencia de alta de Admin

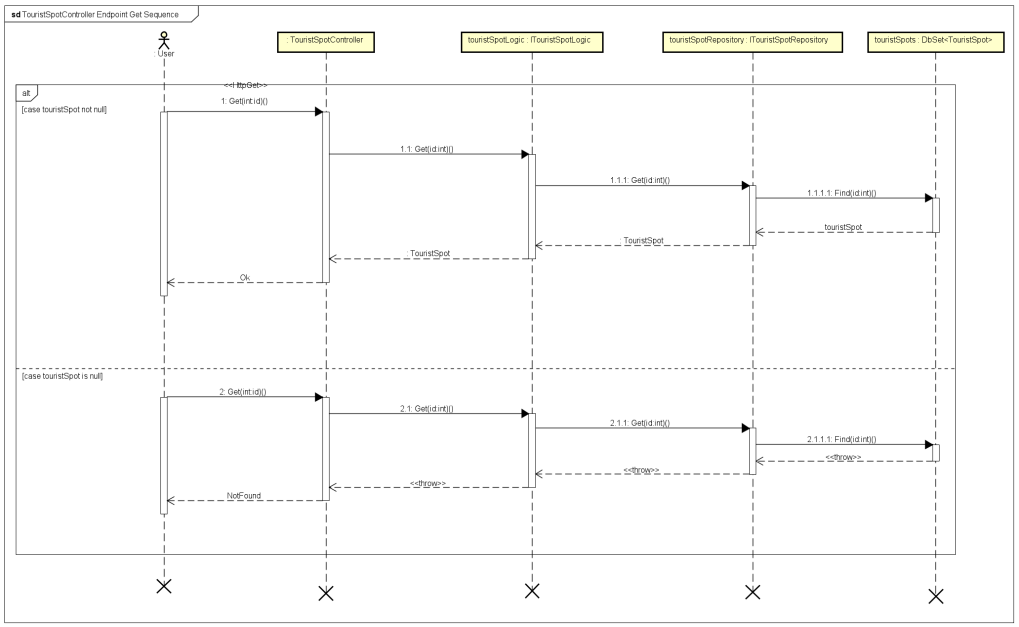

Figure 3.13: Secuencia de get de TouristSpot

3.3 Vista de implementación:

Mencionaremos brevemente las responsabilidades que tiene cada uno de los proyectos y paquetes relevantes para nuestra solución. También presentaremos el diagramas de paquetes y el diagrama de componentes.

- **Obligatorio.Domain:** Este proyecto se encarga de modelar, a nivel de objetos, el dominio de negocio con el que estamos trabajando. A su vez posee subpaquetes con las siguientes responsabilidades:
  - **Obligatorio.Domain.DomainEntities:** Este paquete posee las entidades propias del dominio de negocios. Estas responden a nuestra manera de abstraer el problema real y llevarlo a un diseño orientado a objetos. Dada la naturaleza de las clases en este paquete, que cada una modela una entidad real. Estas siguen una única razón de cambio: que la realidad cambie. Por eso están juntas en un paquete anidado.
  - **Obligatorio.Domain.AuxiliaryObjects:** Este paquete responde a limitaciones del ORM. A la hora de modelar la colección de imágenes que tiene un hospedaje, el ORM nos exigía mapearlas a nivel de contexto. Ya que la naturaleza de este objeto responde a una línea de cambio distinta a las entidades propiamente dicho, decidimos crear un paquete anidado para ellas.
- **Obligatorio.WebApi:** Este proyecto provee una Api que funciona como único punto de acceso a nuestra lógica de negocios (backend). Pensamos que va a funcionar como *fachada* para nuestro futuro frontend (que ahora es Postman). Veamos los subpaquetes que posee:
  - **Obligatorio.WebApi.Controllers:** Este paquete posee controladores. Los controladores responden a una necesidad de la tecnología que usamos, y están todos juntos, ya que deberían cambiar al mismo ritmo: todos ellos tienen dependencias hacia la lógica de negocios y el dominio, y no al revés; porque la tecnología de la Api es mucho más inestable que nuestras operaciones.
  - **Obligatorio.WebApi.AuxiliaryObjects:** El nombre del paquete no es el más agraciado, pero creemos que el nombre revela, dentro de su generalidad, el alcance del mismo. Posee clases que responden a necesidad de otros subpaquetes de la Api.
- **Obligatorio.Factory:** Este proyecto es el responsable del IoC Container. En él se registran los servicios usando la librería de Microsoft para inyección de dependencias. Posee un único paquete interno:
  - **Obligatorio.Factory.Factories:** El paquete se encarga de alojar la clase donde registramos manualmente los servicios. Esta clase es necesaria para la inyección de dependencias y la inversión de control. Decidimos agregarlo para que no todo se haga desde el Startup.cs.

- **Obligatorio.Model:** Este paquete existe respondiendo a la necesidad de separar nuestro modelo de negocios con lo que se expone al cliente de nuestra aplicación. Los DTOs deberían ser *lightweight* y no necesariamente contener todos los datos del modelo de negocios: por ejemplo, los navegation properties en persistencia. Entre ellos encontramos:
  - **Obligatorio.Model.Models.In:** Se encarga de proveer modelos de entrada al sistema y mecanismos para enlazarlos con objetos si corresponde.
  - **Obligatorio.Model.Models.In:** Se encarga de proveer modelos de salia al sistema y mecanismos para llevar objetos a su correspondiente modelo de salida.
  - **Obligatorio.Model.DTOS:** Se encarga de proveer clases que modelen formatos de queries que se le pueden enviar al sistema.
- **Obligatorio.Migrations:** Este proyecto se creó bajo la motivación del patrón *fabricación pura*, pero aplicado a paquetes. Creíamos que darle la responsabilidad de hacer las migraciones a otro paquete sería darle una responsabilidad que le bajaría la cohesión al paquete (el principal candidato era DataAccess, pero este ya tenía varias responsabilidades). De aquí salió este paquete.
- **Obligatorio.DataAccess:** Este proyecto se encarga de más de una cosa, pero delega las diferentes responsabilidades a subpaquetes, reuniéndolas según afinidad entre ellas. En términos generales se encarga de la persistencia y acceso a datos desde la lógica de negocios. Podemos encontrar los siguientes paquetes:
  - **Obligatorio.DataAccess.Context:** Este paquete se encarga de la configuración del modelo relacional.
  - **Obligatorio.DataAccess.Repositories:** Provee implementaciones de los servicios de acceso a datos para cada entidad que se persiste. Es responsable de suministrar a la lógica de negocios los objetos necesarios desde la base de datos para sus operaciones.
  - **Obligatorio.DataAccess.Migrations:** Se guarda el modelo incremental de la base de datos. Por limitaciones del ORM este paquete se crea en el proyecto donde está el contexto.
  - **Obligatorio.DataAccess.CustomExceptions:** Guarda excepciones definidas por nosotros con nombres específicos para nuestras necesidades dentro del proyecto actual.
- **Obligatorio.DataAccessInterface:** En este abstracto proyecto se proveen los contratos que cada repositorio debe ofrecer. Estos contratos se encuentran en:
  - **Obligatorio.DataAccessInterface.Interfaces:** Responsable de proveer los contratos para que sean implementados por los diferentes repositorios.

- **Obligatorio.BusinessLogic:** Este paquete se encarga de proveer a la WebApi con las operaciones que involucren a los elementos de la lógica de negocios que se persisten. También actúa como nivel de indirección entre la WebApi y la capa de acceso a datos y persistencia. Contiene:
  - **Obligatorio.BusinessLogic.Logics:** Es responsable de implementar los contratos especificados en el paquete de lógicas abstractas. Provee las operaciones que la WebApi necesita utilizando objetos que se persisten.
  - **Obligatorio.BusinessLogic.CustomExceptions:** Es responsable de proveer excepciones específicas para enfrentar defensivamente las solicitudes de la Api.
- **Obligatorio.BusinessLogicInterface:** Es responsable de especificar los contratos que las lógicas concretas deben implementar. Estas especificaciones se encuentran en:
  - **Obligatorio.BusinessLogicInterface.Interfaces:** Provee los contratos que las lógicas concretas deben ofrecer.

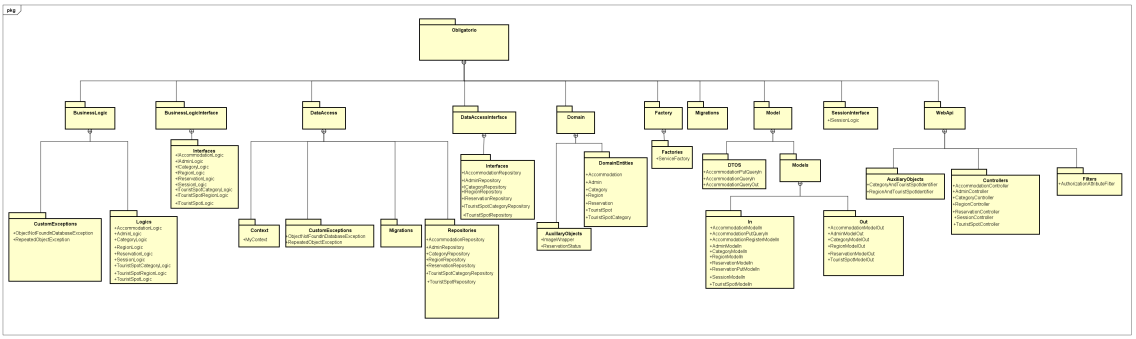

Figure 3.14: Diagrama de paquetes anidados

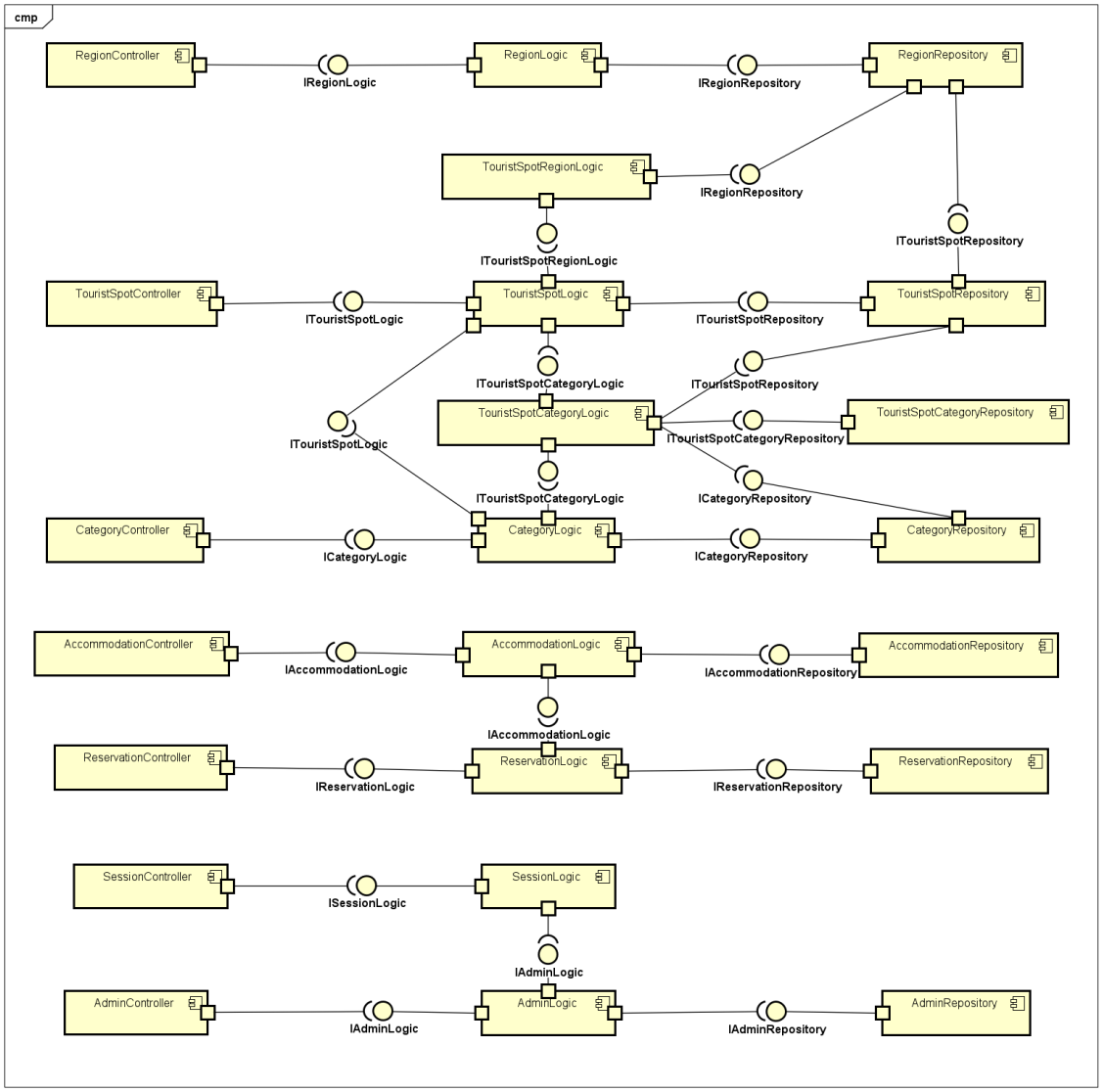

Figure 3.15: Diagrama de componentes

3.4 Vista física:

Mostraremos brevemente los componentes físicos involucradas en el despliegue del obligatorio mediante un diagrama de deploy o despliegue muy sintético. No haremos muchos comentarios respecto a este diagrama ya que creemos que no lo amerita.

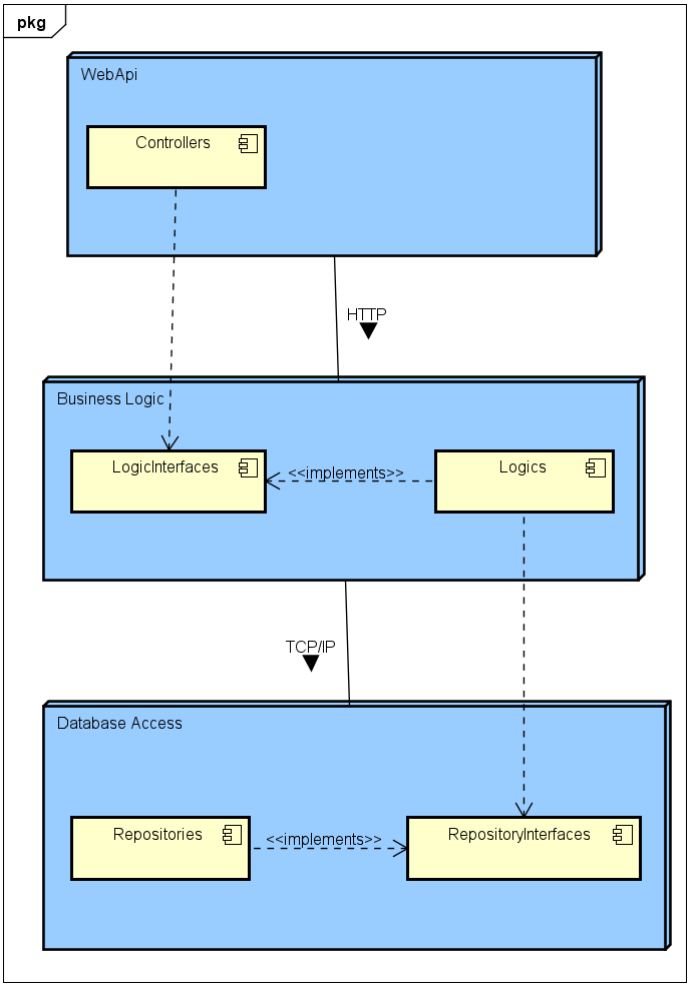

Figure 3.16: Diagrama de deploy

4. Model persistencia:

Adjuntamos un diagrama entidad relación del modelo utilizado en persistencia de datos:

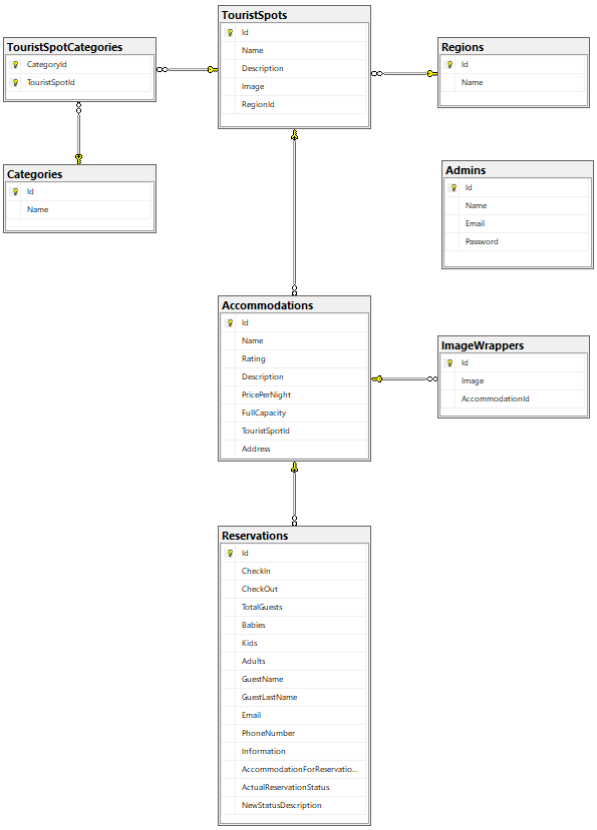

Figure 4.1: Modelo ER

5. Justificación de las decisiones de diseño:

Se hizo gran énfasis en DIP. La gran mayoría de relaciones de dependencias entre paquetes se hacen mediante interfaces. Esto nos garantiza estabilidad. De la mano con estabilidad, nuestros paquetes más *concretos* dependen de aquellos más *abstractos*.

Personalmente, seguimos las recomendaciones de M. Fowler en el artículo [1], que hace gran énfasis en cómo definir comportamiento en diseño orientado a objetos para diferenciarlo de la tradicional programación estructurada.

El mecanismo de inyección de dependencias intentó seguir SRP. El proyecto Factory tiene como razón de ser no cargar a la WebApi ni a DataAccess con la responsabilidad de registrar servicios para realizar DI. El IoC Container tiene su propia clase. Los servicios se registraron como scoped. Por motivos de performance optamos por no hacerlos singleton. Nunca manejamos la opción de transient.

El proyecto Model existe para que la WebApi trabaje con modelos y no con las entidades de dominio originales, las cuales tienen múltiples navegation properties, las cuales pueden ser costosas de transferir y muchas veces superfluas a los efectos de la Api.

A la hora de asignar responsabilidades, primaron los patrones *experto en información* y *bajo acoplamiento*. La idea fue asignar una responsabilidad a la clase que tiene la información para hacerlo, siempre y cuando esto no genere un acoplamiento alto o rápidamente solucionable a través de fabricación pura, sin sobrecomplejizar el diseño. Un ejemplo son las operaciones de inserción de entidades al contexto: los repositorios de cada entidad conocen el DbSet que el contexto tiene para esa entidad, ergo, son los indicados para realizar operaciones sobre la misma.

Bibliography

[1] Martin Fowler. (2013-9-5) Tell, dont ask. [Online]. Available: https://martinfowler.com/bliki/TellDontAsk.html

[2] "Universidad ORT Uruguay". (2020) "Letra del primer obligatorio de Diseño de Aplicaciones I, semestre impar 2020". [Online]. Available: "http://www.aulas.ort.edu.uy"

[3] R. C. Martin, *Clean Code: A handbook of agile software craftsmanship*, 1st ed. Prentice Hall, 2009.