Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de aplicaciones II
Obligatorio II

Juan Pablo Barrios - 206432
Juan Ignacio Irabedra - 212375

Entregado como requisito de la materia Diseño de aplicaciones 2

Acceder al repo

26 de noviembre de 2020

Contents

1. Introducción:

A lo largo de este documento presentaremos el diseño de la aplicación que entregamos como obligatorio 2 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 funcionalidades que serán cubiertas en el apartado correspondiente.

El proyecto fue desarrollado usando C# como lenguaje para el back end, 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.

Por otro lado, el front end fue desarrollado como una single page application (SPA) usando AngularJS. A través de un router de Angular, la página es capaz de cambiar el template que muestra sin necesidad de recargar: esto hace a la experiencia de usuario, la cual es fluida. Como comentario, intentamos basarnos en las heurísticas de Nielsen para la SPA. Hicimos énfasis en la visibilidad del estado del sistema, heurística que consideramos clave: nuestra idea es que ante cualquier acción que el usuario realice, se le brinde algún tipo de feedback sobre lo que hizo.

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.

Estos puntos serán cubiertos inidivudalmente más adelante en el documento a los efectos de dar evidencia de la realización de los mismos.

- 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 ✓
- Creación de reportes (tipo A), que dadas fechas y un punto turístico, indica cuáles y cuántas reservas con ciertos estados existen para el período dado ✓
- Importar datos desde diversas fuentes resueltas en tiempo de ejecución ✓
- Escribir reseñas de hospedajes por reserva ✓

Todas las funcionalidades solicitadas por letra fueron cumplidas. Si bien existen bugs u omisiones que cubriremos a continuación.

2.1 Bugs conocidos:

Mencionaremos los bugs conocidos en el sistema.

- Al realizar la búsqueda de hospedajes, no se muestran las imágenes correspondientes a los hospedajes. Si bien se guardan, no se muestran al buscar.
- Al realizar la búsqueda de hospedajes, se muestran, para cada hospedaje, todas las reseñas existentes. No solo las correspondientes a ese hospedaje. Sin embargo, para cada una de estas reseñas se indica a qué hospedaje pertenece.

3. Descripción de diseño:

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.

3.1 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. Mencionaremos cómo funcionan algunos de los mecanismos clave de nuestra solución. También mostraremos un diagrama de estructura compuesta de algunos pocos objetos de nuestro dominio.

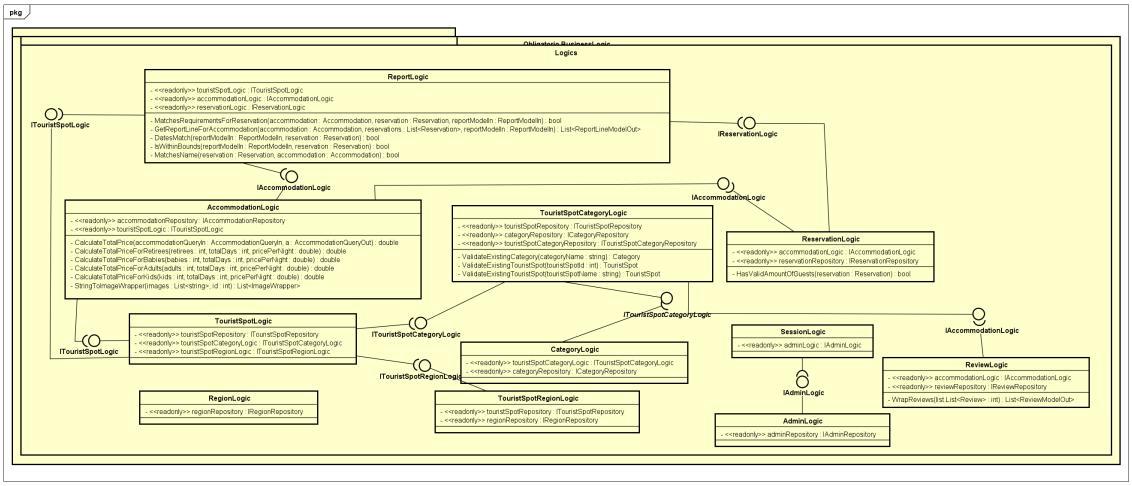


Figure 3.1: Diagrama de clases de BusinessLogic

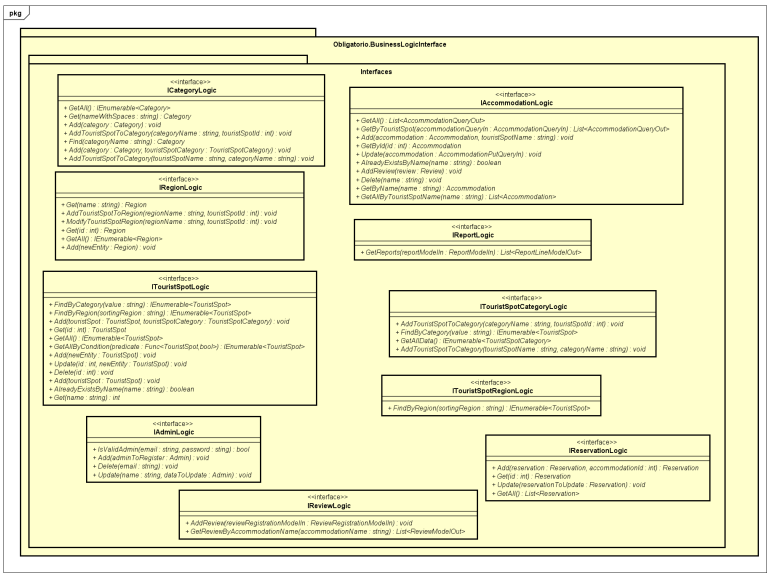


Figure 3.2: Diagrama de clases BusinessLogicInterface

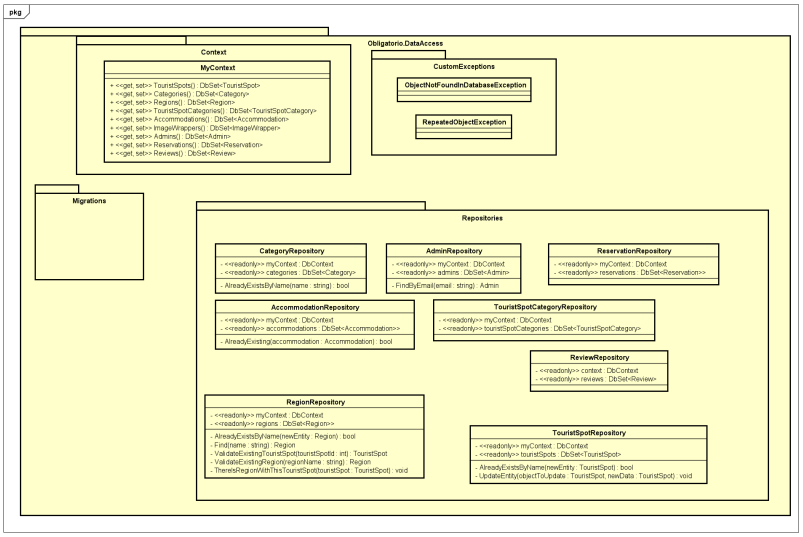


Figure 3.3: Diagrama de clases de DataAccess

Respecto a los paquetes, intentamos aplicar CCP. Las lógicas están en un paquete, los repositorios en otro, los contratos en otro, los controladores... diferentes elementos que agrupan comunes características están agrupados juntos, y estos, al tener un motivo de existencia análogo, están sujetos al mismo tipo de cambios. Por otro lado, las dependencias son acíclicas, la dirección del cambio es única.

Un buen ejemplo, como se ve en la figura 3.3, dentro de un paquete tenemos subpaquetes que agrupan elementos que responden al mismo tipo de cambios. Y todos estos subpaquetes son parte de uno más grande que los reune, ya que se usan juntos: las clases de los repositorios usan las excepciones locales y necesitan de la clase que hereda del DbContext. A su vez, necesitamos del DbContext para realizar migraciones, de donde se sigue que se está obedeciendo CRP.

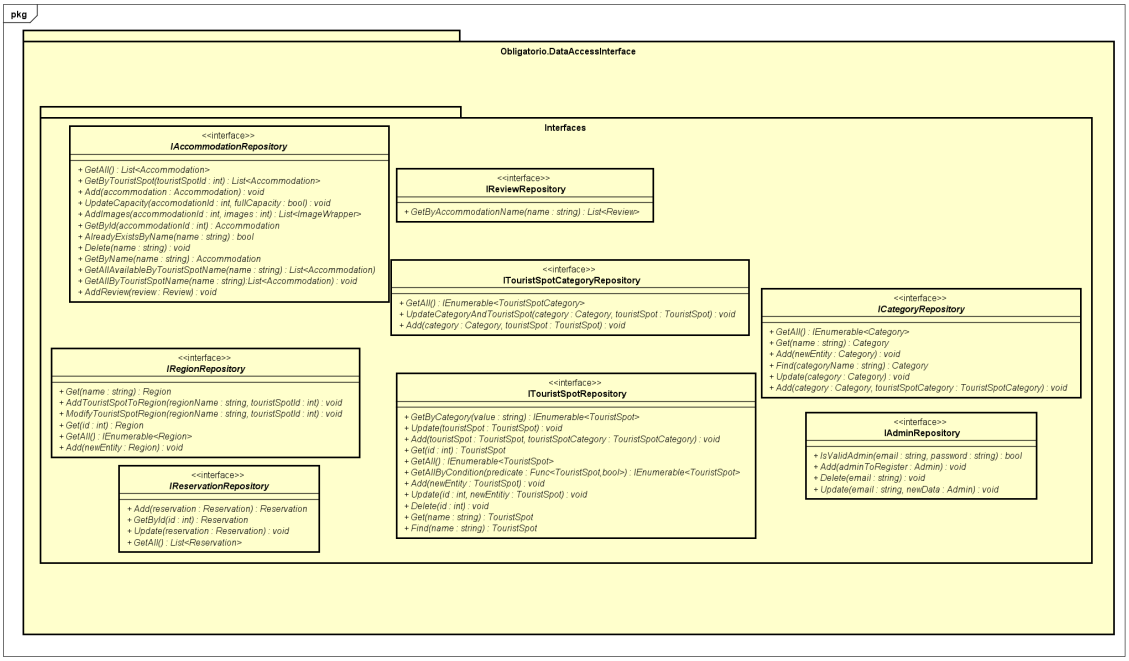


Figure 3.4: Diagrama de clases de DataAccessInterface

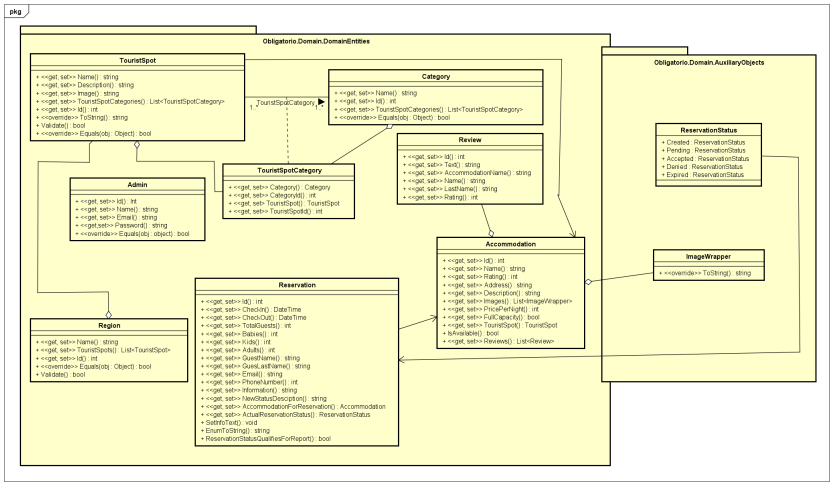


Figure 3.5: Diagrama de clases de Domain

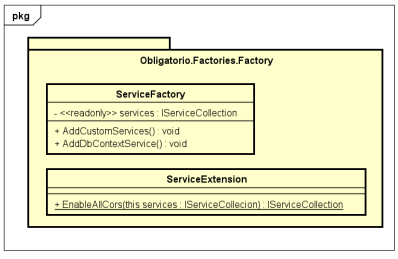


Figure 3.6: Diagrama de clases de Factory

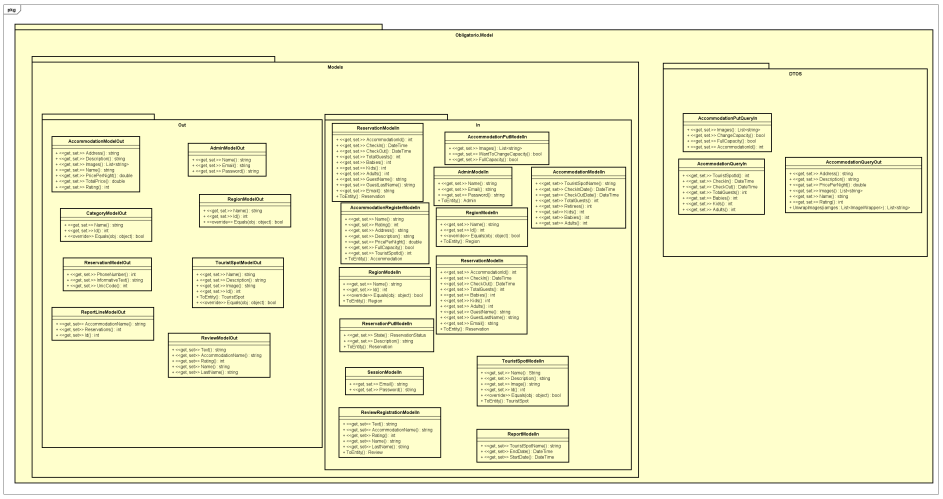


Figure 3.7: Diagrama de clases de Model

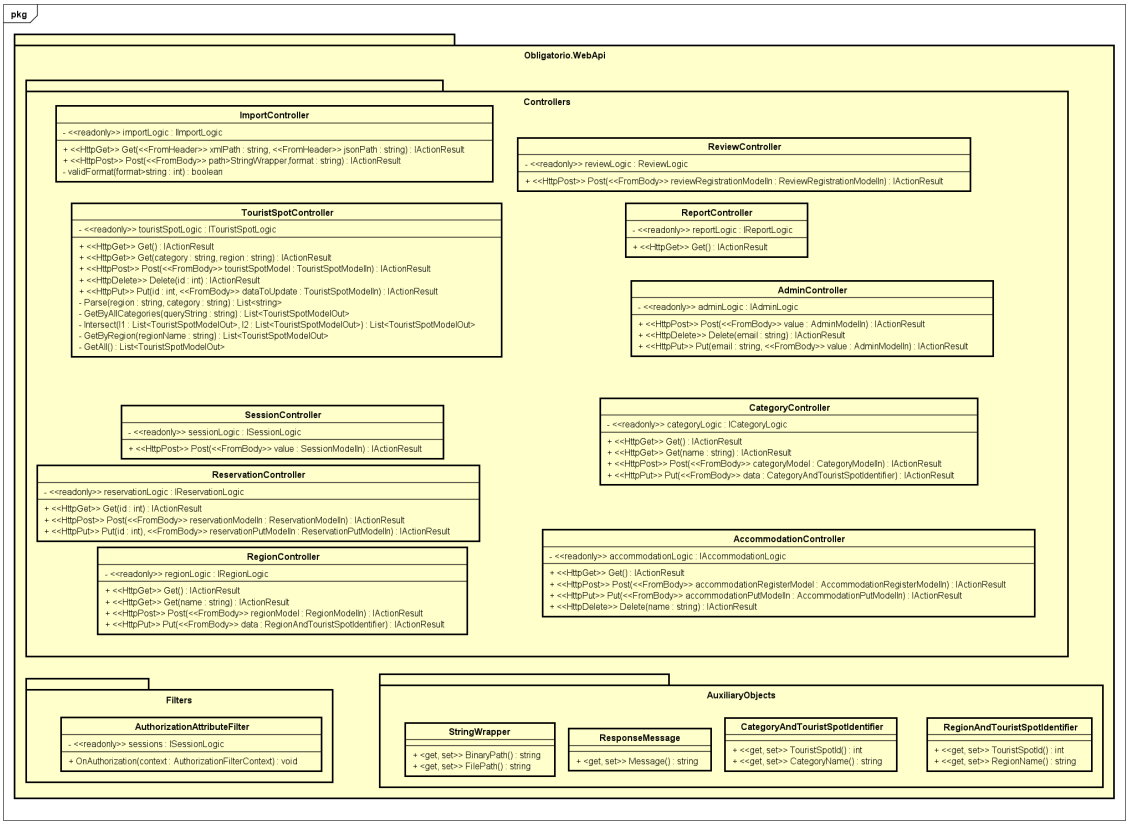


Figure 3.8: Diagrama de clases de WebApi

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

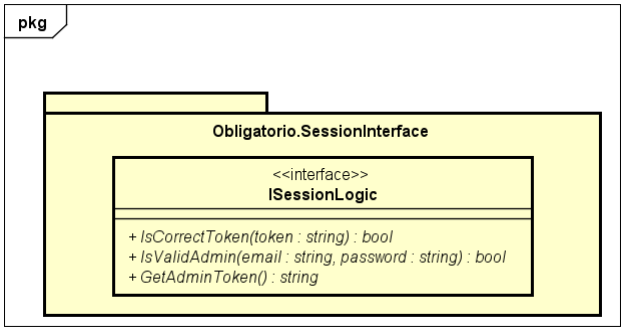


Figure 3.9: Diagrama de clases de SessionInterface

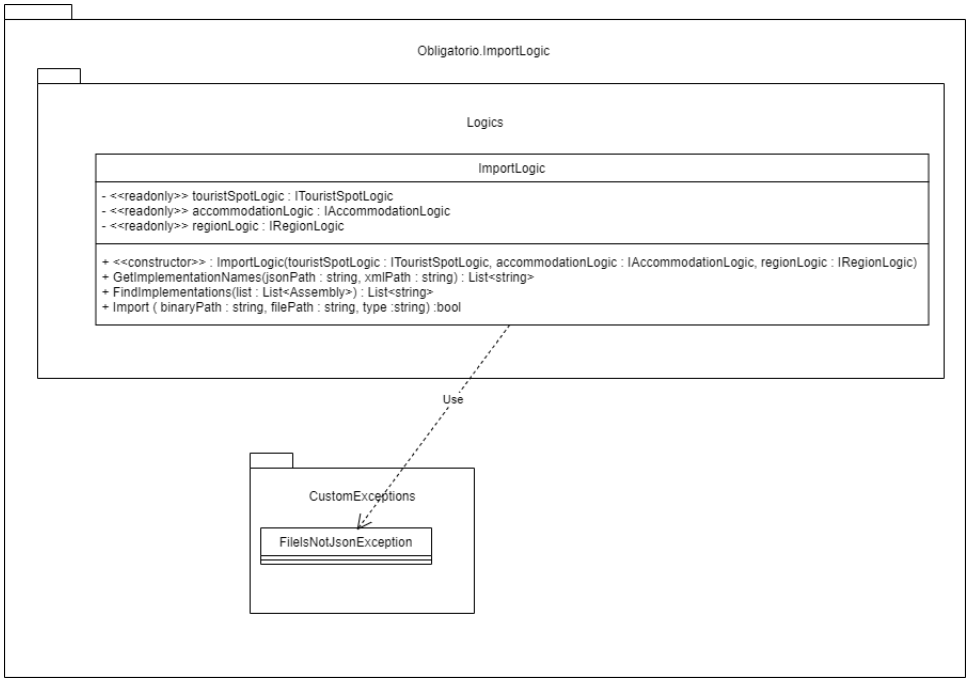


Figure 3.10: Diagrama de clases Obligatorio.ImportLogic

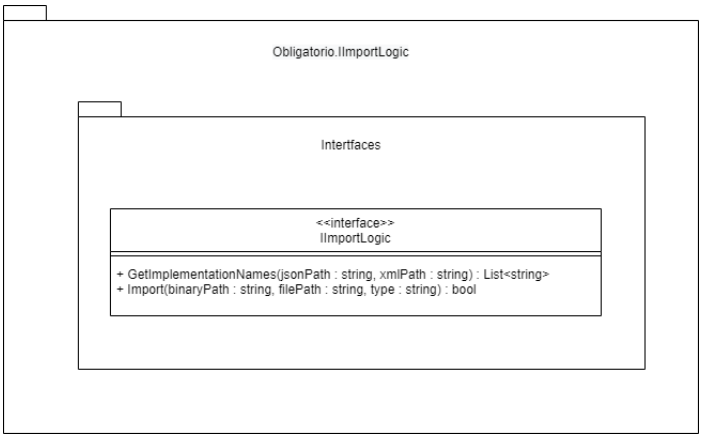


Figure 3.11: Diagrama de clases Obligatorio.ImportLogicInterface

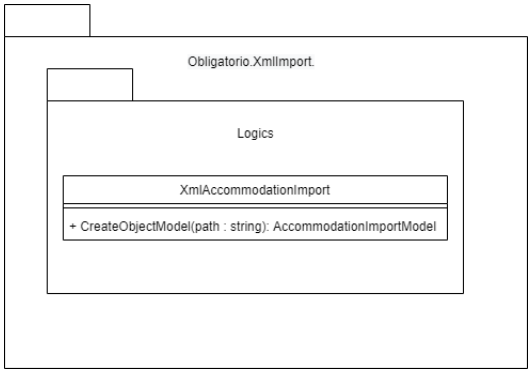


Figure 3.12: Diagrama de clases Obligatorio.XmlImport

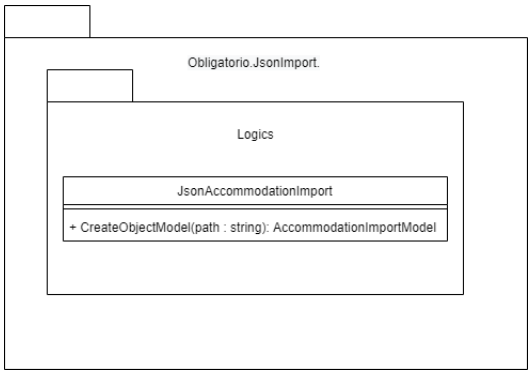


Figure 3.13: Diagrama de clases Obligatorio.JsonImport

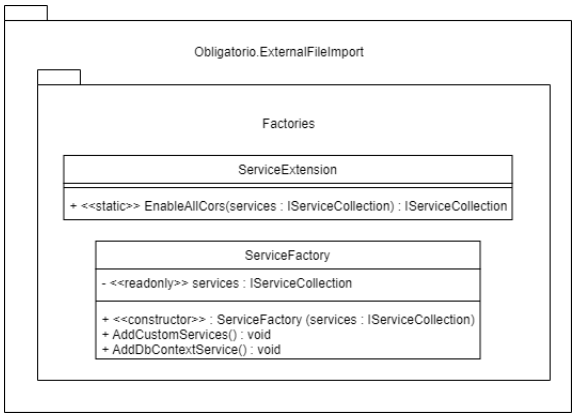


Figure 3.14: Diagrama de clases Obligatorio.ExternalFileImport

3.1.1 Búsqueda de puntos turísticos por región y por categoría:

Decidimos que la solución a este requerimiento empiece en el TouristSpotController. Ya que decidimos que esta clase se encargue de todas las funcionalidades que involucren directamente puntos turísticos, aunque terminen siendo varias. Esto se debe a que nos vemos limitados por la tecnología de una restfulApi.

Se recibe la configuración de la búsqueda mediante el endpoint Get del con-

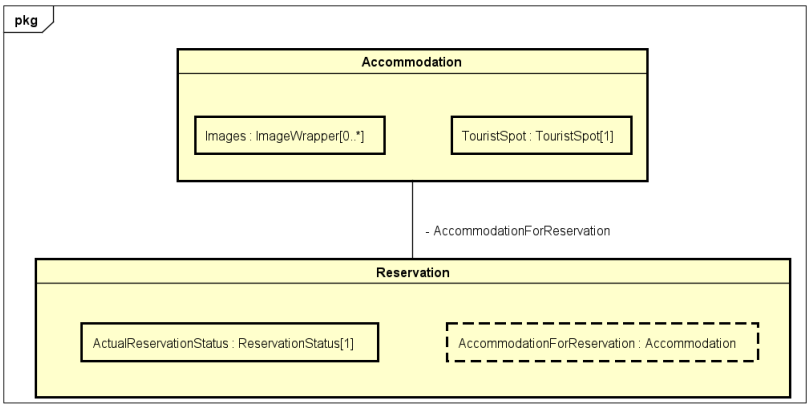


Figure 3.15: Diagrama de estructura compuesta reservation

trolador en el componente WebApi. Este se encarga de parsing del querystring, si aplica, y delega los siguientes pasos a otras capas. También es responsable de responder al cliente. Luego interviene TouristSpotLogic, del componente BusinessLogic. Esta dependencia no es directa, porque aplicamos DIP. TouristSpotLogic actúa como capa de indirección para este requerimiento, si bien también valida existencia de parámetros de búsqueda. TouristSpotLogic finalmente delega la responsabilidad de encontrar los puntos turísticos a TouristSpotRepository, en DataAccess. Detalles de implementación no se muestran en el diagrama de secuencia a continuación, el cual sintetiza lo narrado.

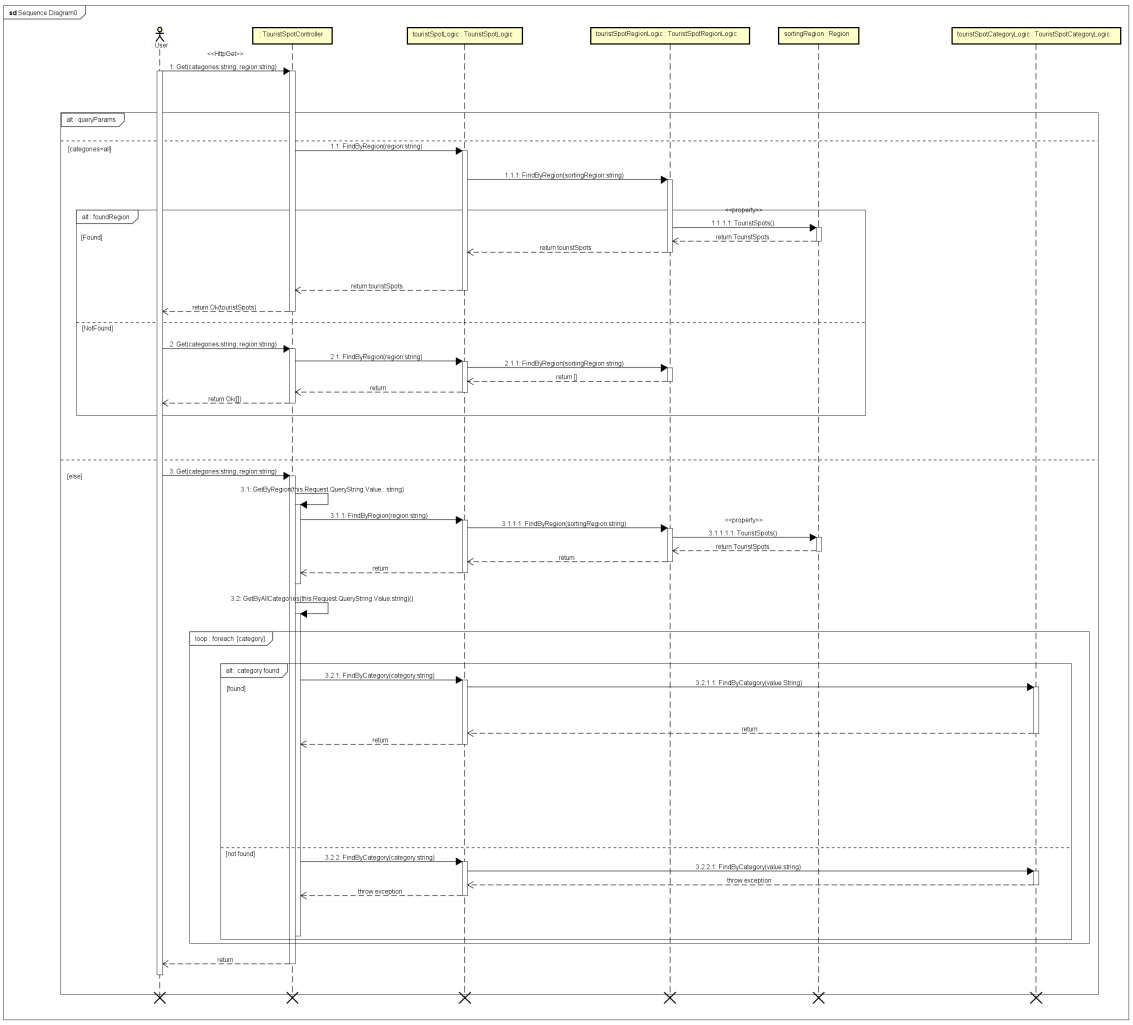


Figure 3.16: Diagrama de secuencia para búsqueda de punto turístico por región y categoría

3.1.2 Elegir un punto turístico y buscar hospedajes, realizar una reserva:

Desde el front end se configura el querystring que recibirá el verbo Get de AccommodationController. En caso de que el querystring es nulo, se devuelven todas la Accommodation que existen. Si no, se hace una búsqueda más compleja: la cual filtra según punto turístico y capacidad actual del hospedaje.

Para la búsqueda en caso de que se configuren parámetros de búsqueda: AccommodationController es responsable de manejar el querystring y calcular el costo. Se delega la búsqueda a AccommodationController y AccommodationRepository, quienes se encargan de filtrar por punto turístico y disponibilidad. Como indicado en el obligatorio 1, la disponibilidad de un hospedaje es manejado por el sistema: así que no se revisó que no exista reserva para dicha fecha en dicho hospedaje.

Para la creación de reservas, el usuario indica en el front end una de las Accommodation encontradas en la búsqueda recién mencionada, ingresa sus datos personales

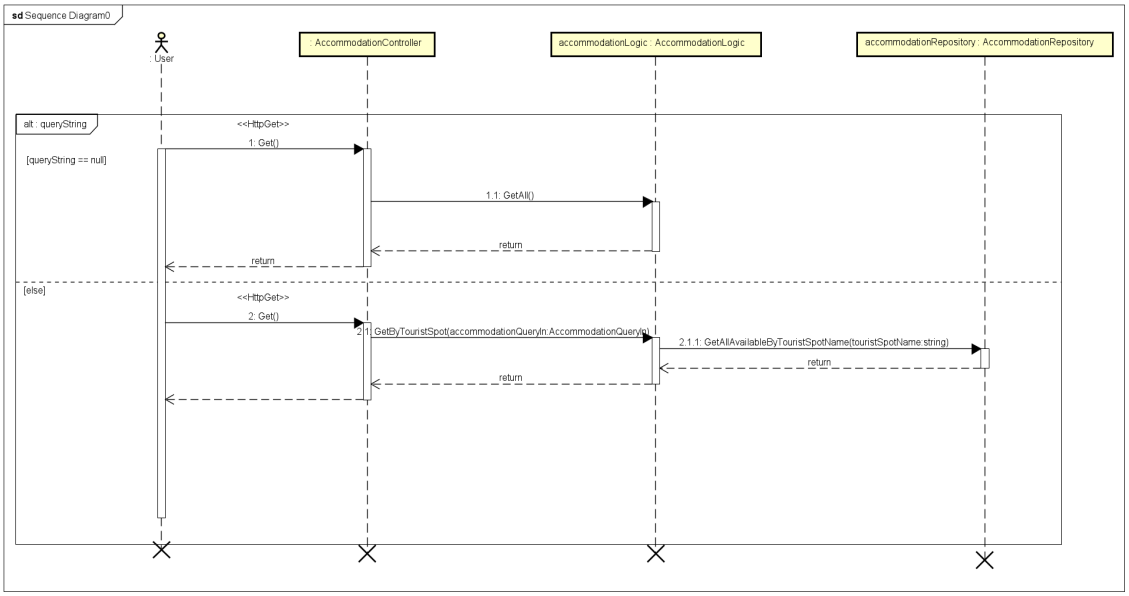


Figure 3.17: Diagrama de secuencia para búsqueda de hospedaje por punto turístico

y reserva. Esto se resuelve en WebApi, en ReservationController con el verbo post. Esta clase se encarga de delegar la creación a ReservationLogic y esta última se la delega a ReservationRepository. Estas clases trabajan en conjunto con AccommodationLogic y AccommodationRepository para validar que la Accommodation en la que se busca reservar exista: esto se hace a los efectos de que no se quiera reservar en un hospedaje que fue borrado mientras se hace la búsqueda.

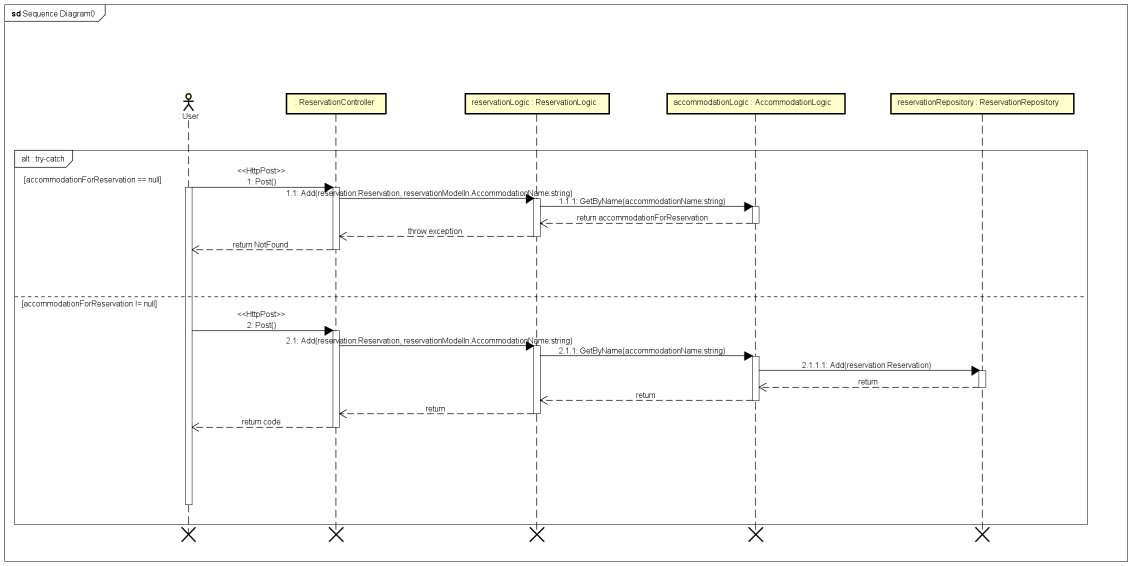


Figure 3.18: Diagrama de secuencia para reserva

3.1.3 Imports:

Este requerimiento involucró el uso de reflection. El usuario ingresa rutas donde se encuentran los .dll para usar. Si los encuentra, los lista y le permite elegir uno de ellos. Además, se le permite ingresar la ruta del archivo a importar. Solo se permite hacerlo uno a la vez. Cuando envía el formulario, se carga y avisa al usuario o lanza error.

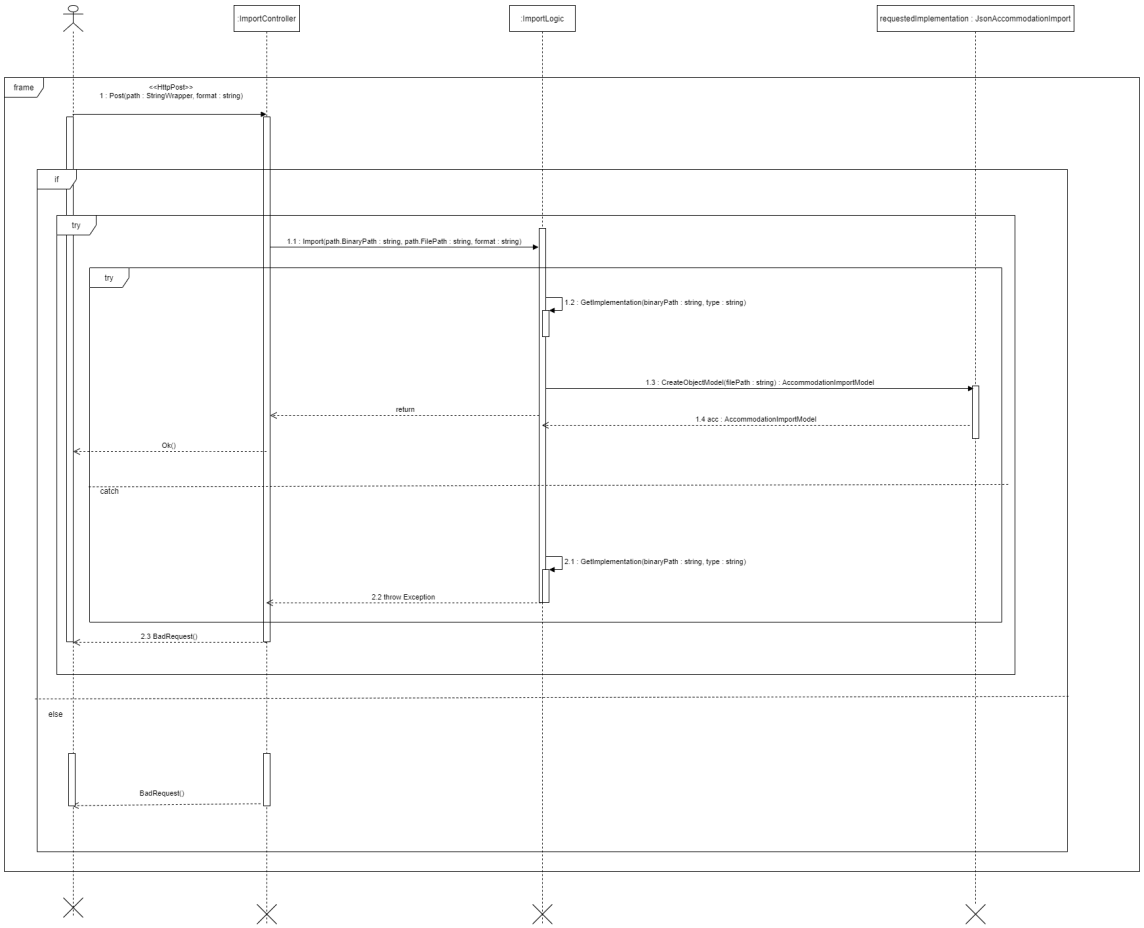


Figure 3.19: Diagrama de secuencia para reserva

A nivel de diseño, creamos un ImportController aplicando fabricación pura, ya que no teníamos candidatos a expertos en imports. También creamos proyectos ExternalFileImport, ImportLogic e ImportLogicInterface. ExternalFileImport se encarga de trabajar directamente con las librerías externas al sistema y los modelos. IImportLogic provee el contrato necesario para importar: llevar a cabo reflection, elegir en tiempo de ejecución cómo se va a importar, y delegar a las librerías externas la tarea de hacer el parse. Mostraremos un diagrama de secuencia para uno de los casos.

Para su funcionamiento, se debe tener .dlls que posean ya sea xml o json en su nombre. También, un archivo de texto .txt en formato json o xml con los siguientes datos: "Name":string, "Rating":[1...5], "Address":string, "Description":string, "TouristSpot":"Name":string, "Description":string, "Image":string , "Image":string, "PricePerNight":int, "FullCapacity":bool. An[alogo en formato Xml sin comillas para los nombres de las properties. Se adjuntan ejemplos de prueba en la carpeta del entregable.

3.2 Vista de procesos:

Consideramos que no tenemos elementos de valor para aportar en esta sección.

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. Como último comentario antes de desarrollar, queremos notar que elegimos nombres de packages generales para cualquier problema análogo, ya que, creemos que los componentes, o la gran mayoría de ellos, involucrados en nuestra solución podrían perfectamente usarse en proyectos que involucren turismo análogas. Nuestra aplicación ofrece servicios similares a los que una aplicación como Booking, Trivago, Airbnb, entre otros ofrecen.

- **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.
- **Obligatorio.ExternalFileImport:** se encarga concretamente de proveer interfaz para crear los objetos importados. Para ello utiliza
  - **Obligatorio.ExternalFileImport.Models** provee los modelos de los cuales se va a leer los objetos importados
  - **Obligatorio.ExternalFileImport.Import:** provee el contrato que las librerías externas deben cumplir para importar.
- **Obligatorio.ImportLogicInterface:** brinda interfaz para que el controlador de acople a ella para poder importar y así respetar DIP. Dentro vemos:
  - **Obligatorio.ImportLogicInterface.Interfaces:** aquí se ubica la interfaz mencionada.
- **Obligatorio.ImportLogic:** realiza la interfaz necesaria para que el componente WebApi se comunique con las librerías de terceros sin acoplarse a ellas. Aquí se encuentran:
  - **Obligatorio.ImportLogic.CustomExceptions:** brinda excepciones específicas para este paquete.
  - **Obligatorio.ImportLogic.Logics:** provee la lógica para resolver la decisión de qué binario usar para importar y validar paths tanto de los .dll como del archivo a importar.

- **Obligatorio.JsonImport y Obligatorio.XmlImport:** actúan cómo módulos que permiten importar datos al sistema.

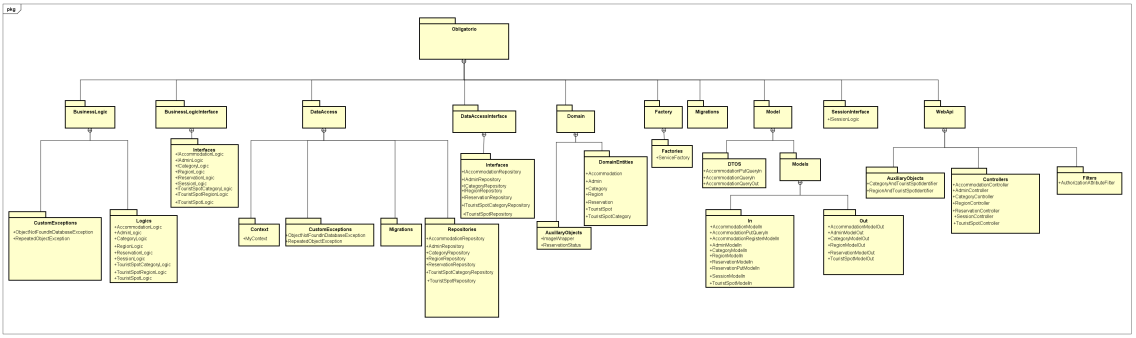


Figure 3.20: Diagrama de paquetes anidados

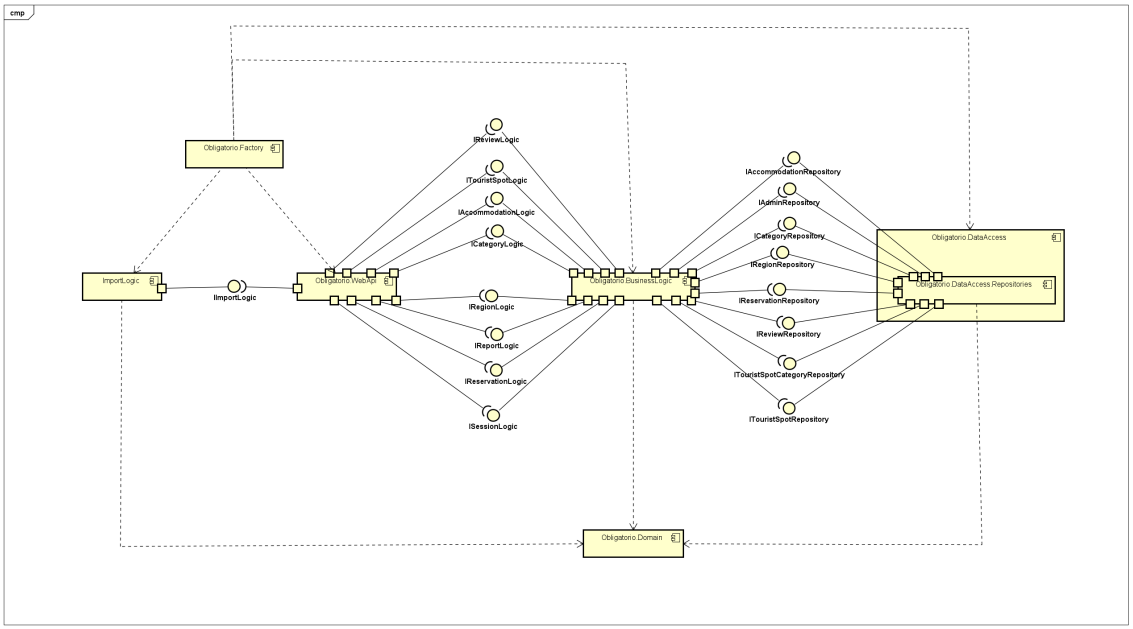


Figure 3.21: Diagrama de componentes

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.

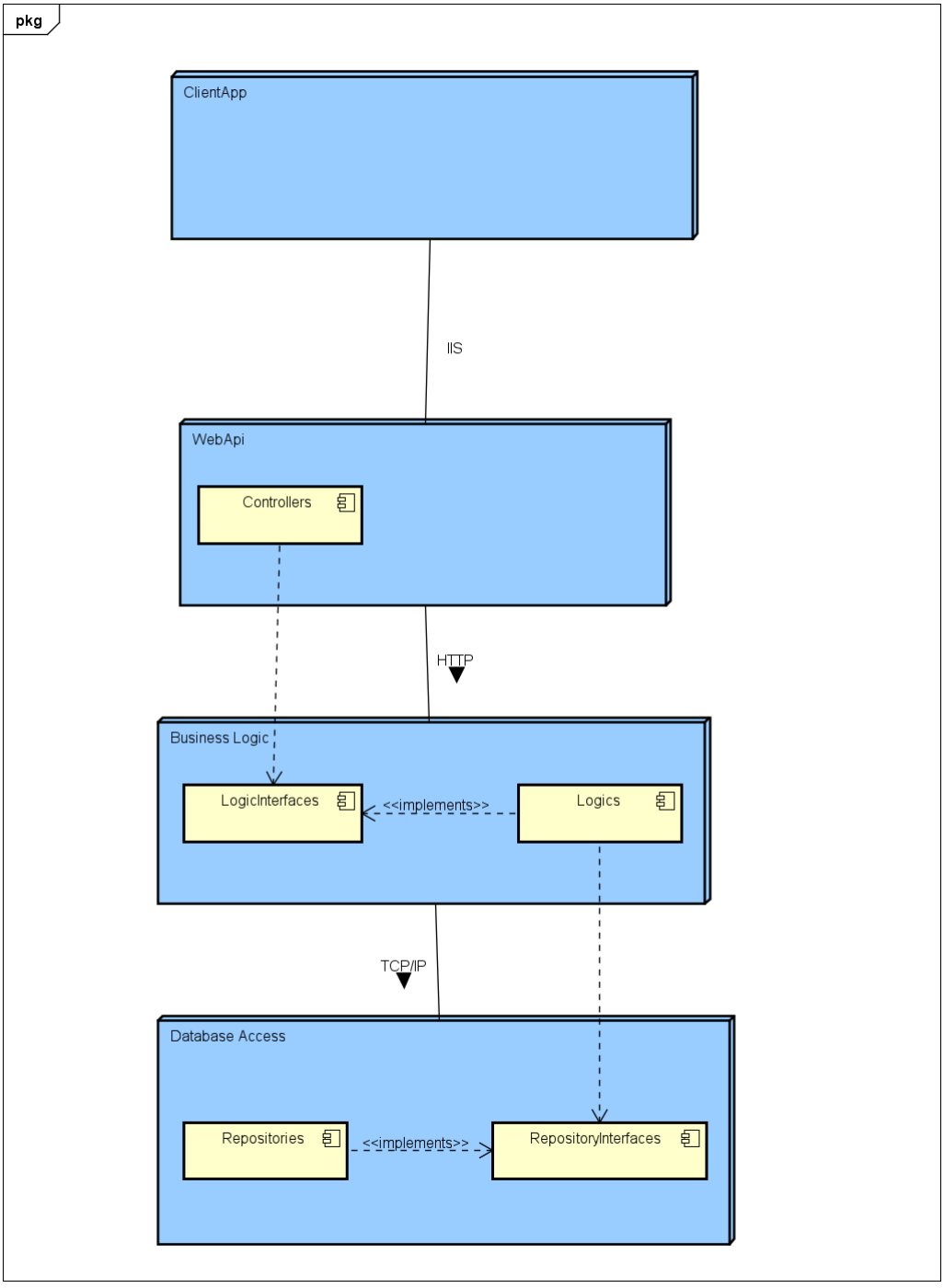


Figure 3.22: Diagrama de deploy

4. Representación de objetos en persistencia:

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

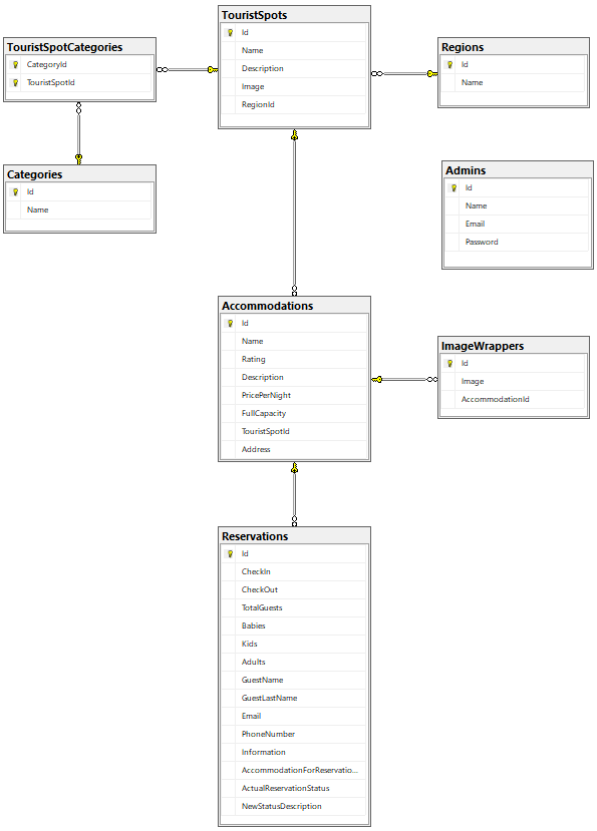


Figure 4.1: Modelo ER

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.

5.1 Mejoras

La primera mejora que realizamos fue en la clase TouristSpotController. En la entega anterior nuestro método Get rompía con clean code, para esta entrega decidimos eliminarlo y recontruirlo. En la entrega anterior todo el parsing del query string fue hecho a mano, para esta entrega aprendimos utilizar las herramientas que nos da c#. En esta versión del método intentamos respetar clean code lo más posible, es corta, recibe una cantidad adecuada de argumentos, no tiene efectos secundarios, etc.
La segunda mejora que realizamos fue en la clase AccommodationLogic, en esta clase modificamos el método CalculateTotalPrice, antiguamente realizábamos todas las operaciones en el mísmo método haciendo que sea muy poco entendible. Para esta entrega decidimos encapsular todas estas operaciones en métodos para poder realizar su mantenimiento más fácil y poder mantener un mismo nivel de abstracción en el método. Para esta funcionalidad en particular manejamos la posibilidad de realizarla utilizando el patrón Strategy, este sería un caso en el que si utilizamos el patrón el método sería abierto a el cambio y cerrado a la modifiación. Después de debatir sobre si era necesario o no aplicar el patran en este caso llegamos a la conclusión de que generar una estrategia para cálculos tan sencillos era hacer un sobrediseño y agregaría comlejidad innecesaria a la solución final.

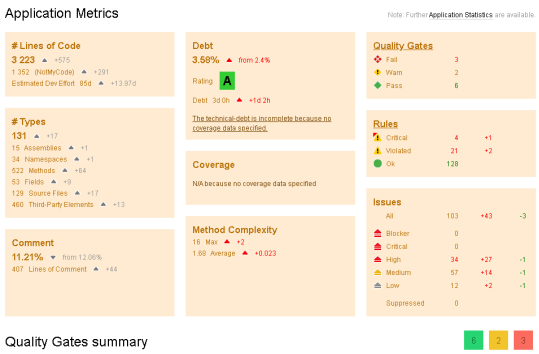
5.2 Análisis basado en métricas:

NDepend nos arrojó una mayoría de valores dentro de límties sanos. Como se ve en la figura siguiente, la mayoría de las métricas dan en valores verdes. La general debt está en un ranking muy bueno.

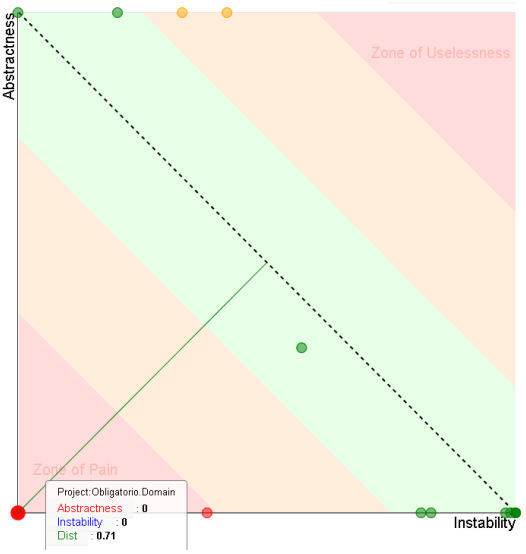
En términos de acoplamiento y dependencias, creemos que el obligatorio está en un muy buen estado. Extenderlo fue fácil, y los cambios que hubo que hacer no fueron dolorosos de hacer. Esto lo vinculamos con la gráfica de abstractness vs instability. Solo tenemos un package en la zona de dolor, y es el dominio. Esto tiene todo el sentido del mundo, ya que todo el obligatorio versa en torno a esto, y así creemos debe ser, porque es la realidad de la cual se abstrayó el problema. No tenemos ningún componente cerca de la zone of uselessness.

Notamos que hay dos assemblies con valores bajos de cohesión relacional: DataAccess y Model. DataAccess tiene muchas responsabilidades que podrían haberse segregado en diferentes assemblies, pero habría crecido mucho el tamaño de la solución. Decidimos dejarlas juntas por tener algo en común: persistencia. Model tiene muchas clases que se usan juntas y responden al mismo tipo de cambios: los modelos, por tanto, creemos que se podría mejorar, pero no fue una prioridad.

Notamos las reglas que Ndepend llama críticas que violamos: existe un par de packages mutuamente dependiente, este caso aislado viola ADP. Por otro lado, tenemos dos tipos con mismo nombre en diferente namespace (dos custom exceptions que definimos). Finalmente, nos dio una regla crítica violada en la api, la cual no atendimos mucho porque creemos que solucionarla no impacta a los efectos de la mantenibilidad del obligatorio. Por motivos de extensión, mantendremos este análisis breve.



(a) Resumen reporte NDepend

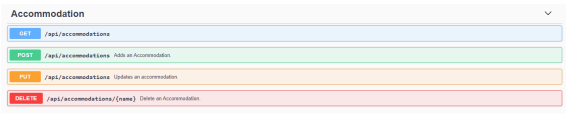


(b) Abstactness vs Instability

Figure 5.1: Elementos de NDepend

6. Anexos

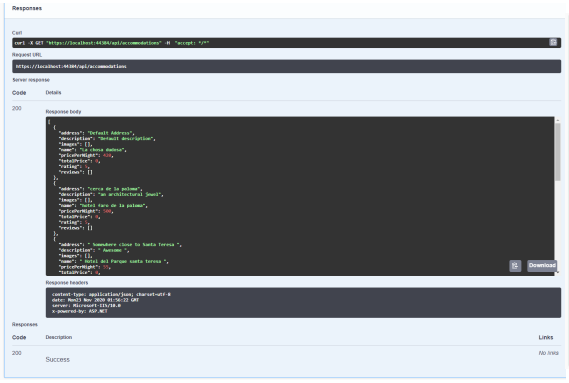
Respecto a TDD, no podemos evidenciar que lo aplicamos porque no lo hicimos. La api está documentada en swagger, con mejoras respecto a la entrega anterior.



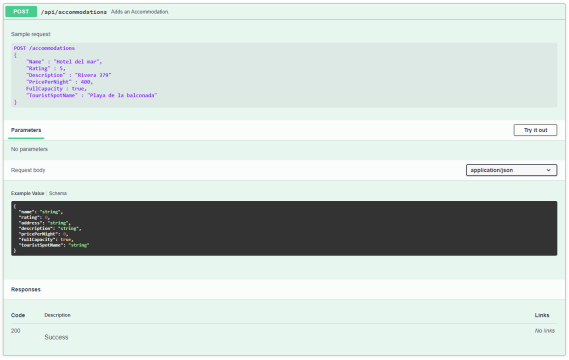
(a) Swagger Accommodation

(b) Delete

Figure 6.1: Elementos de NDepend

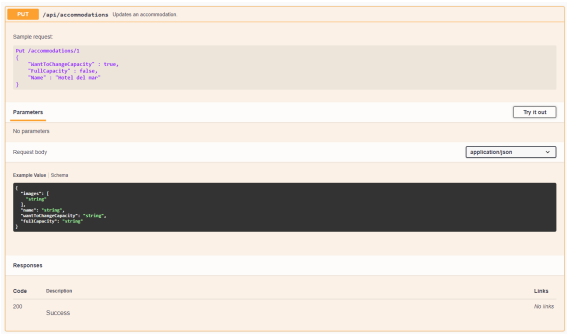


(a) Get

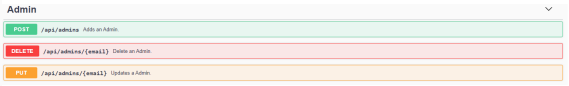


(b) Post

Figure 6.2: Elementos de NDepend



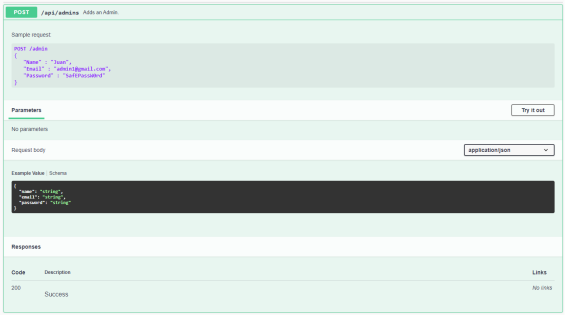
(a) Put



(b) Swagger Admin

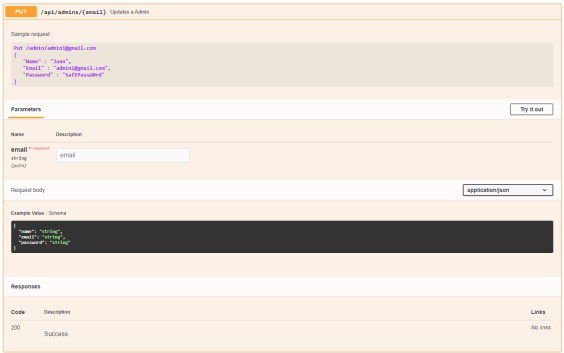
Figure 6.3: Elementos de NDepend

(a) Delete

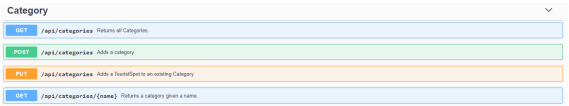


(b) Post

Figure 6.4: Elementos de NDepend

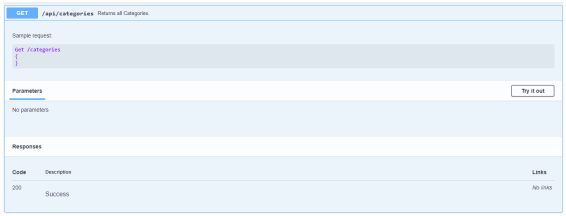


(a) Put

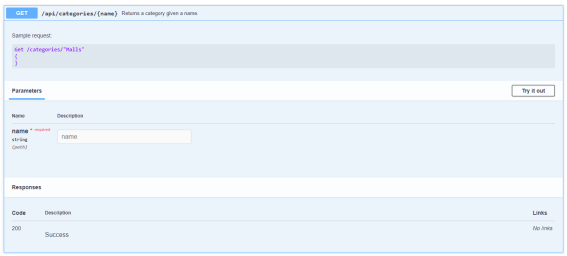


(b) Swagger Category

Figure 6.5: Elementos de NDepend

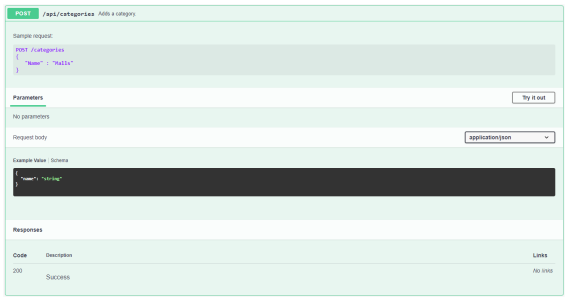


(a) Get



(b) Get

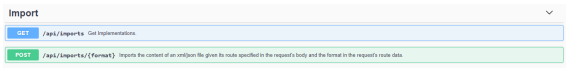
Figure 6.6: Elementos de NDepend



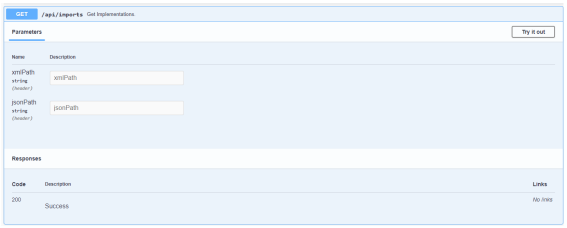
(a) Post

(b) Put

Figure 6.7: Elementos de NDepend

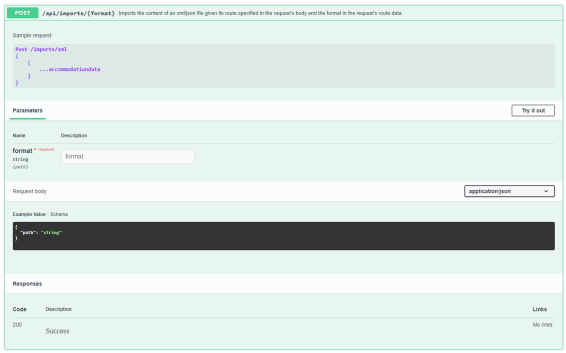


(a) Swagger Import

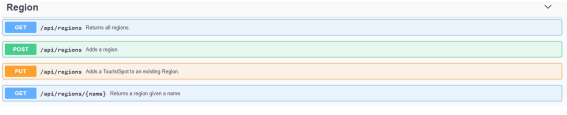


(b) Get

Figure 6.8: Elementos de NDepend

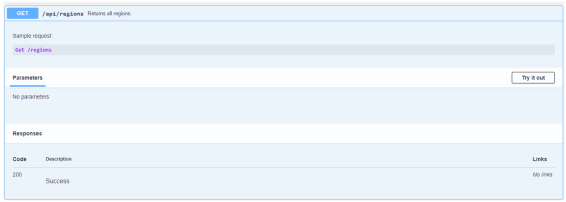


(a) Post

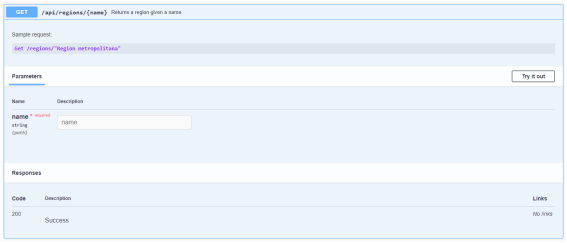


(b) Swagger Region

Figure 6.9: Elementos de NDepend

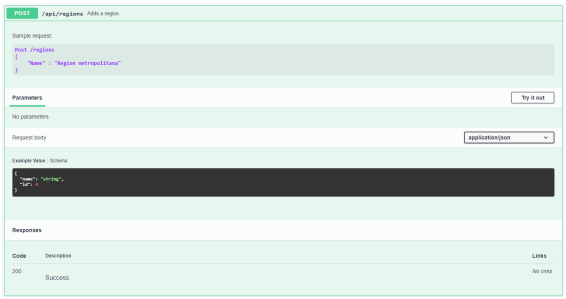


(a) Get

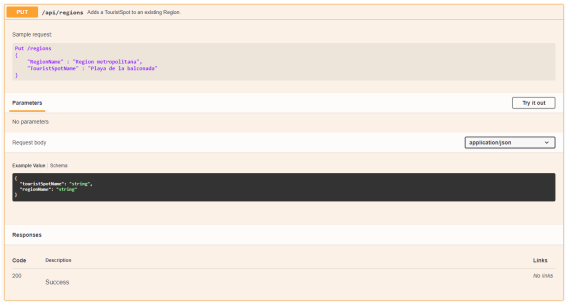


(b) Get

Figure 6.10: Elementos de NDepend

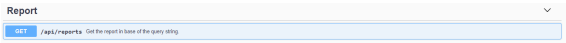


(a) Post

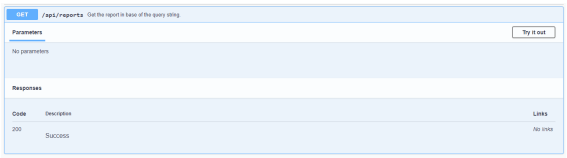


(b) Put

Figure 6.11: Elementos de NDepend

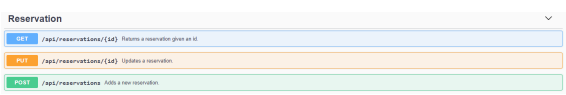


(a) Swagger Reporte

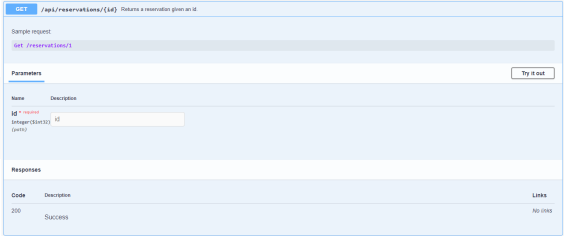


(b) Get

Figure 6.12: Elementos de NDepend

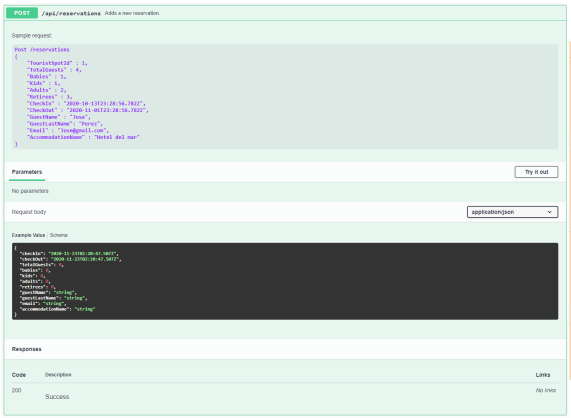


(a) Swagger Reservation



(b) Get

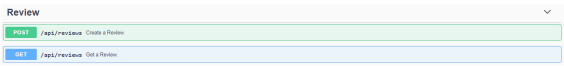
Figure 6.13: Elementos de NDepend



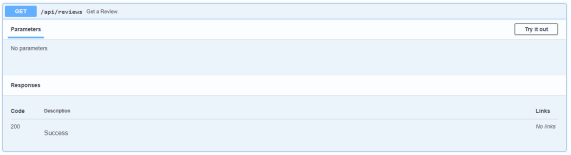
(a) Post

(b) Put

Figure 6.14: Elementos de NDepend

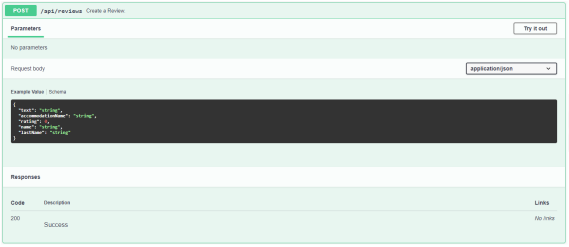


(a) Swagger Review

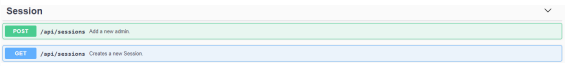


(b) Get

Figure 6.15: Elementos de NDepend

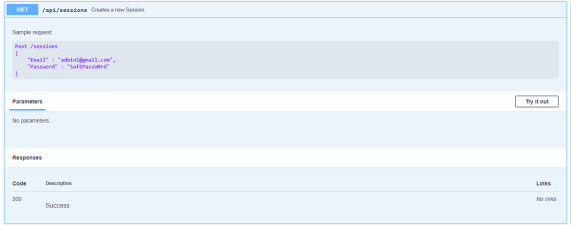


(a) Post

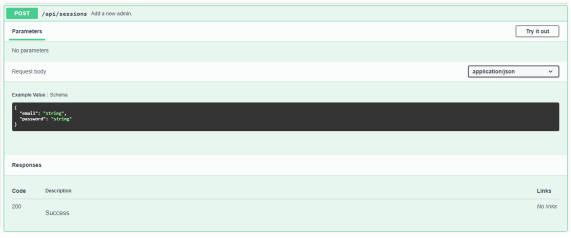


(b) Swagger Session

Figure 6.16: Elementos de NDepend

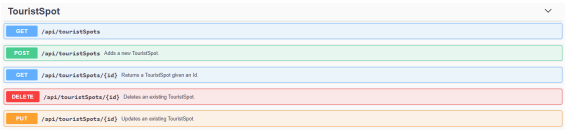


(a) Get

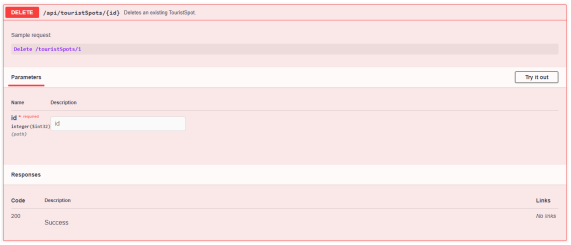


(b) Post

Figure 6.17: Elementos de NDepend

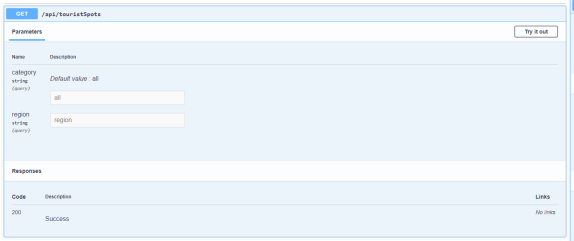


(a) Swagger TouristSpot

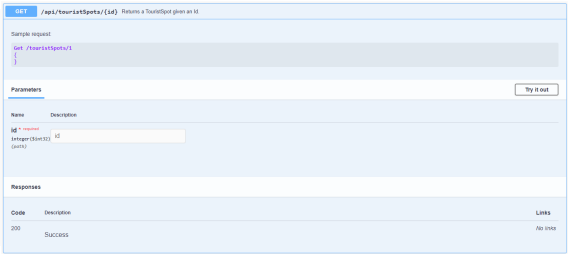


(b) Delete

Figure 6.18: Elementos de NDepend

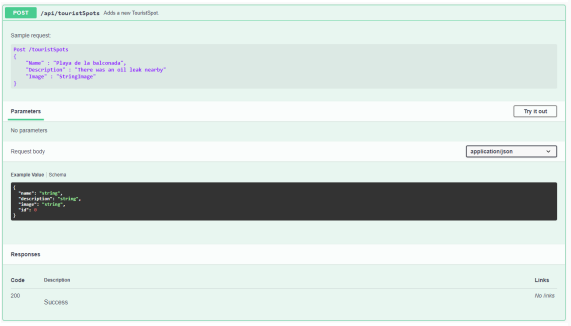


(a) Get

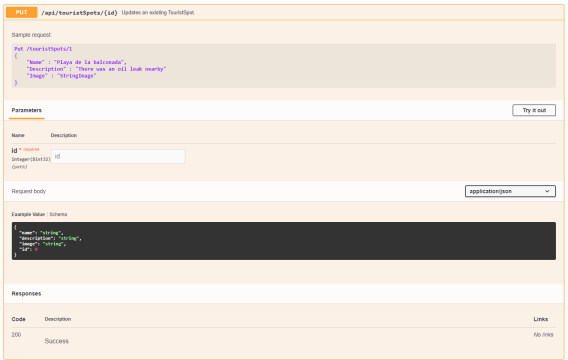


(b) Get

Figure 6.19: Elementos de NDepend



(a) Post



(b) Put

Figure 6.20: Elementos de NDepend

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.