



Obligatorio 2 - Diseño de aplicaciones 2 Diseño y documentación

Agustín Hernandorena (233361)
Joaquín Lamela (233375)

<https://github.com/ORT-DA2/233375-233361Obl>

Índice

Descripción general del trabajo	5
Diagrama de descomposición de los namespaces	6
Diagrama general de paquetes (namespaces)	8
Responsabilidades de cada paquete	9
Justificación del diseño para cada paquete	11
Importación de hospedajes	18
Dominio de la solución del problema	20
Mecanismo de acceso de datos	21
Estructura de las tablas en la base de datos	22
Manejo de excepciones	24
Diagramas de secuencia	26
Obtener puntos turísticos por región y categorías	26
Ingreso de una reserva	27
Generación de reporte de reservas en hospedajes	28
Diagrama de componentes	29
Justificación y explicación del diseño en base al uso de principios de diseño, patrones de diseño en paquetes y métricas.	31
Principio de clausura común (CCP)	31
CRP - Common Reuse Principle	31
Principio de dependencias acíclicas (ADP)	32
Principio de dependencias estables (SDP)	33
¿Cuales son las dependencias existentes? ¿Cumplen con el principio SDP?	
33	
Matriz de dependencia	33
Métricas de inestabilidad	34
Principio de abstracciones estables (SAP)	35
Abstracción vs Inestabilidad	36
Mejoras al diseño	39
Precio de la reserva	39
Verbos en URI	39
Endpoints, controller y tests innecesarios	39
Extensibilidad ante nuevos tipos de huéspedes	40
Manual de instalación	40
Deployment BackEnd	40
Deployment FrontEnd	44
Anexo	46
Mecanismo de borrado de elementos	46
Excepciones en el dominio	46
Excepciones en el acceso a datos	47
Excepciones en la importación	47
Cobertura de las pruebas	48
BusinessLogic	49
BusinessLogicException	49

BusinessLogicTest	50
DataAccess	50
DataAccessTest	51
Domain	51
DomainException	51
Filters	52
Models	52
Repository Exception	52
WebApi	52
WebApiTest	53
Importation	53
ImportationTest	53
ImporterException	54
ImporterJson	54
ImporterXml	54
API Rest	55
Principios REST	55
Interfaz uniforme	55
Peticiones sin estado	55
Cacheable	56
Separación de cliente y servidor	56
Sistema dividido en capas	56
Mecanismos de autenticación de request	57
Descripción general de códigos de error	58
Descripción de los resources de la API	61
Región controller	61
Category controller	61
Tourist Spot controller	62
Lodging controller	64
Search of lodgings controller	66
Reserve controller	67
User controller	68
Review controller	72
Importer controller	73
Report controller	75
Sección 1	76
Regions controller	76
Category controller	76
Tourist Spot controller	77
Lodging controller	77
SearchOfLodging controller	78
Reserve controller	78

User controller	79
Review controller	79
Import controller	80
Report controller	80

Descripción general del trabajo

El Ministerio de Turismo a través de su conocida marca “Uruguay Natural”, luego de varios meses de trabajo ha detectado algunos problemas con su sitio web actual, siendo el principal de ellos que el contenido es estático y demasiado genérico, por lo que es difícil poder atrapar los potenciales turistas sin ofrecerles una propuesta más atractiva y completa.

A raíz de esto, se nos solicita crear desde cero el sistema, que debe permitir que los posibles turistas tengan una experiencia end-to-end, es decir, deben poder desde explorar lugares turísticos, hasta evaluar y reservar paquetes turísticos para cada cliente.

A su vez, el cliente requiere que todos los datos del sistema sean persistidos en una base de datos. De esta manera, la siguiente vez que se ejecute la aplicación se comenzará con dichos datos cargados con el último estado guardado antes de cerrar la aplicación.

El objetivo principal de esta nueva versión es completar la interfaz de usuario. Para ello construimos una aplicación Angular (SPA) que implementa una lista de nuevos requerimientos y modificaciones de los requerimientos de la primera entrega que se describen a continuación.

- Generación de reporte

Los administradores pueden ahora ejecutar reportes sobre las reservas realizadas por los turistas.

El cliente presentó 2 posibles reportes a implementar, y nuestro equipo se inclinó por el “**Reporte A**”, el mismo permite:

Dado un cierto punto turístico, obtener un listado de hospedajes que tienen reservas para un cierto rango de fechas y ordenado por cantidad de reservas que cada hospedaje tiene.

Además para la generación del reporte se tienen en cuenta los siguientes aspectos:

- Únicamente se deben contabilizar aquellas reservas que no están en estado “expirada” o “rechazada”.
- El reporte sólo debe incluir aquellos hospedajes que tienen al menos una reserva que coincida con el período brindado.
- Si dos hospedajes tienen la misma cantidad de reservas, debe aparecer primero en el listado aquel hospedaje que fue dado de alta antes (el más antiguo).
- Además, si no existe ninguna reserva para ninguno de los hospedajes de un punto turístico, el reporte falla.

- Importación de hospedajes

Se agrega la funcionalidad que permite la opción de importar, desde diversas fuentes de datos y formatos los hospedajes, y si es necesario (en caso que no existiesen) sus puntos turísticos que el sistema aún no posee. El sistema soporta agregar nuevos hospedajes y sus puntos turísticos desde cualquier tipo de fuente, ya sea desde un .xml, desde un .csv, desde un .json, leyéndolo desde una base de datos, consumiendo un servicio, etc.

Para esta versión, se entregan los elementos compilados para poder importar los formatos: xml y json.

- **Nuevo tipo de huésped**

Para esta versión de la aplicación, se agrega un nuevo tipo de huésped denominado “jubilado” (70 años o más). El mismo posee 30 % de descuento del precio por día siempre y cuando esté acompañado de otro jubilado más (es decir, estén en múltiplos de dos).

- **Reseñas de un hospedaje**

Una vez una reserva fue realizada los usuarios pueden usar el sitio para realizar una reseña de su estadía en un hospedaje. Las reseñas pueden ser realizadas por cualquier usuario turista siempre y cuando estos provean un código de reserva válido independientemente del estado de la misma. La reseña consta simplemente de un texto, de una puntuación entre 1 y 5 y del nombre y apellido de la persona. El puntaje total de reseñas de un hospedaje se entiende como un promedio entre todas las reseñas realizadas por los turistas sobre dicho hospedaje. Si no se tienen reseñas para un hospedaje aún, cuando se realicen búsquedas para este hospedaje se indica que el hospedaje no posee puntuación (S/P). Las reseñas además se muestran en la página de un hospedaje (donde se realiza una reserva) mostrando el listado de las mismas que todos los turistas fueron realizando.

Diagrama de descomposición de los namespaces

En las siguientes figuras se muestran los diagramas de descomposición de los *namespaces* del proyecto (utilizando el conector nesting). Por cuestión de visibilidad se decidió dividir el diagrama en dos (aunque realmente es uno solo que comprende todo el sistema).

A los namespaces presentes en la primer versión del proyecto: *BusinessLogic*, *BusinessLogicException*, *BusinessLogicInterface*, *BusinessLogicTest*, *DataAccess*, *DataAccessInterface*, *DataAccessTest*, *Domain*, *DomainException*, *Filters*, *Model*, *RepositoryException*, *WebApi* y *WebApiTest*, se le suman con el fin de cumplir el requerimiento que permite la importación de hospedajes desde diferentes tipos de fuentes, los namespaces: *Importation*, *ImporterException*, *ImporterJson* e *ImporterXml*.

Este primer diagrama mostrado tiene como objetivo mostrar la organización y jerarquía de paquetes (sin mostrar las dependencias, que se van a mostrar con otro diagrama que se presentará más adelante en el documento).

Se puede observar dentro de cada paquete, las entidades e interfaces que lo conforman. Podemos ver que en estos diagramas se utiliza el conector nesting, el cual es una notación gráfica para expresar la contención o anidamiento de elementos dentro de otros elementos. En este caso, lo utilizamos para mostrar de forma apropiada el anidamiento de paquetes en el diagrama de paquetes de nuestra solución.

En la figura (1) del diagrama de descomposición que se muestra a continuación, no se puede apreciar ningún conector nesting debido a que en la parte que se capturó no hay ningún paquete que se encuentre anidado a otro.

Sin embargo, en la figura (2), se pueden apreciar conectores nesting ya que tenemos presencia de paquetes anidados, podemos apreciar como dentro del paquete *WebApi* tenemos a los paquetes: *Filters*, *Model* y *Controllers*.

Asimismo, se puede apreciar que dentro del paquete *Model*, tenemos los paquetes: *ForRequest*, *ForResponse* y *ForResponseAndRequest*. El propósito de los mismos se explicará en detalle en una sección más adelante en este documento.

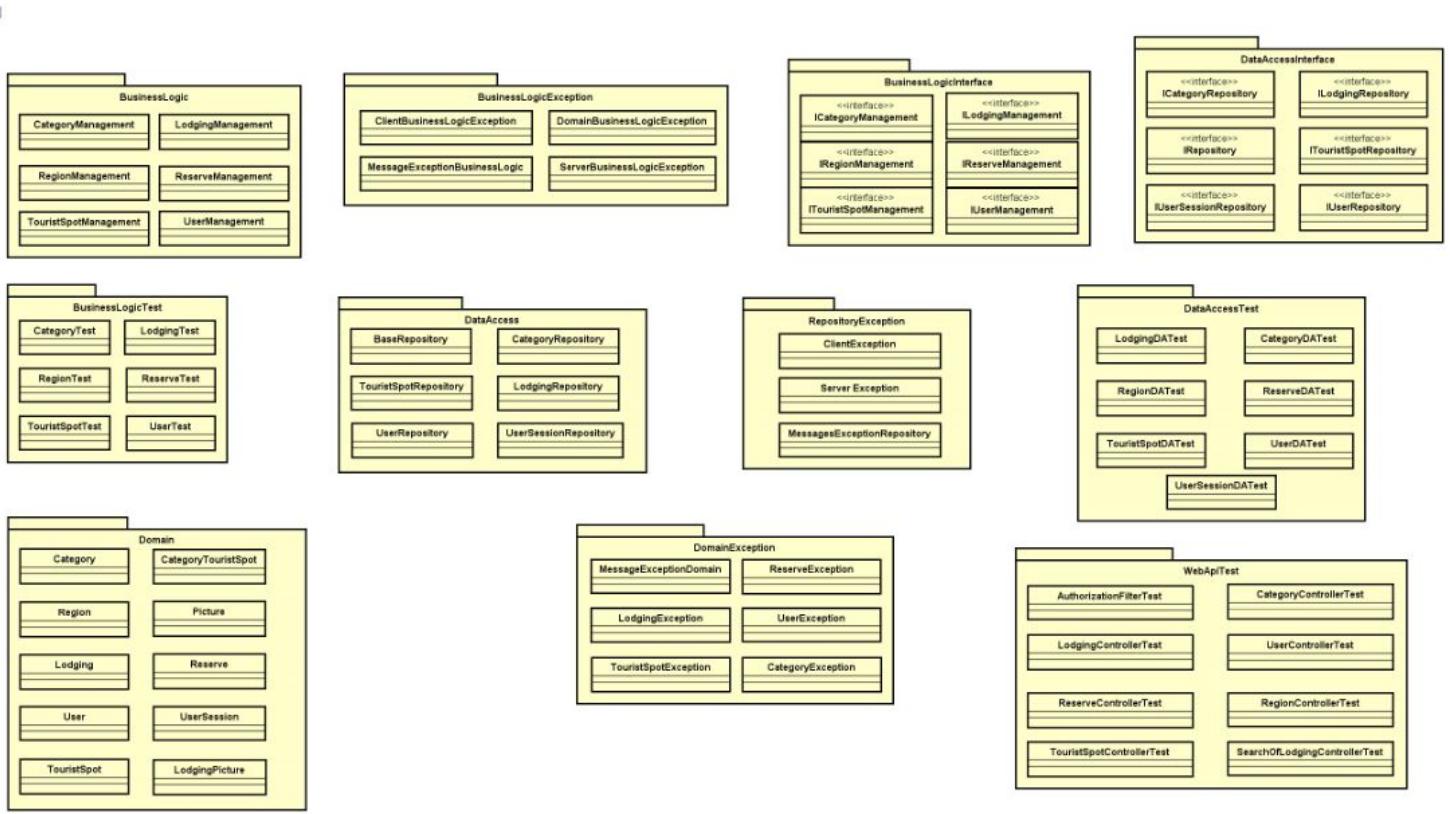


Diagrama de descomposición de los namespaces del proyecto (parte 1).

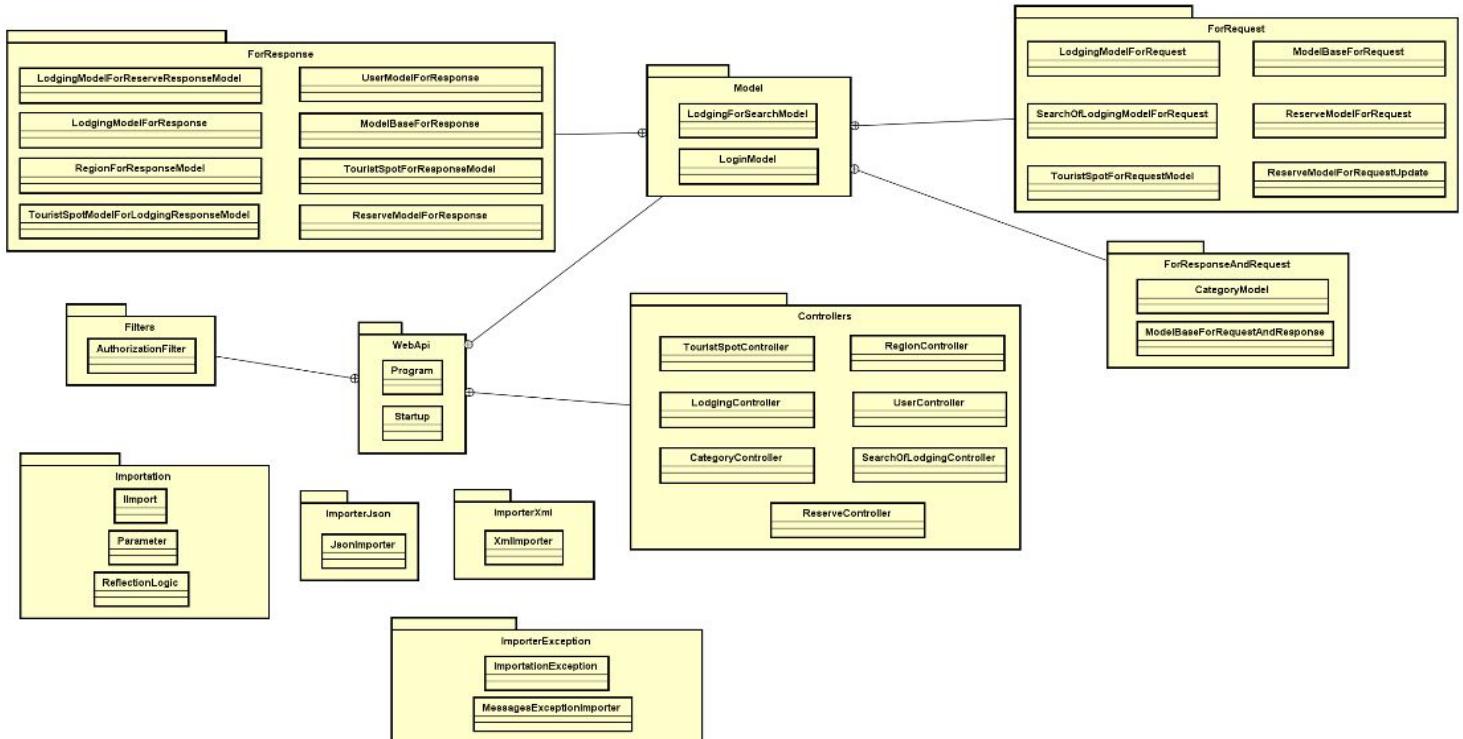
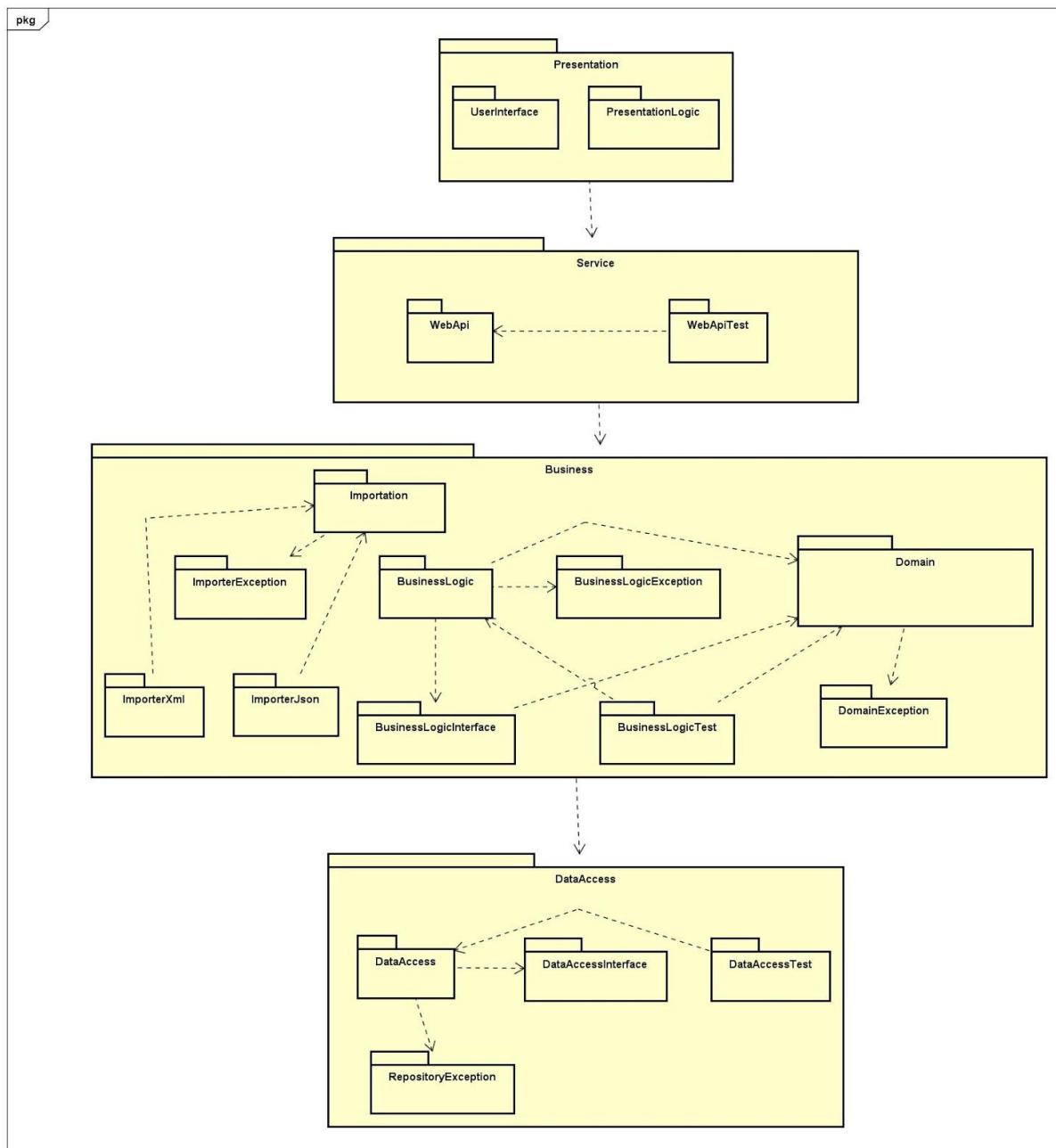


Diagrama de descomposición de los namespaces del proyecto (parte 2).

Este diagrama no muestra aspectos como la dependencia existente entre los diferentes paquetes ni tampoco nos da una visión general de cómo está organizada nuestra aplicación en cuanto a capas, así que para mostrar estos aspectos, procedimos a realizar un diagrama general de paquetes (namespaces) mostrando los paquetes organizados por layers y sus dependencias.

Diagrama general de paquetes (namespaces)



A las tres capas diagramadas en la primera versión, se le suma la *Presentation Layer*.

Esta división de la arquitectura en capas trae grandes ventajas como la separación de responsabilidades entre los componentes. Los componentes dentro de una capa específica tratan solo con la lógica que pertenece a esa capa.

Cada una de las capas cumple con un propósito específico en la aplicación.

En nuestro caso la capa *Service*, está compuesta por el paquete *WebApi* que contiene el paquete *Controllers*, *Models*, y *Filters*, y su responsabilidad es la de manejar las solicitudes HTTP entrantes y enviar la respuesta a quien llama (en este caso la respuesta será hacia la capa de *presentación*).

La capa *Business* es la responsable de ejecutar las reglas de negocio específicas asociadas con la solicitud.

La capa *DataAccess* es la responsable de manejar todo lo relativo al almacenamiento de los datos.

La capa *Presentation* la cual es la responsable de manejar todo lo relativo a la interfaz de usuario y realizar solicitudes a la capa de servicios. En particular, contiene los componentes de interfaz de usuario lanzados en esta versión, construidos por medio del framework Angular. Podemos notar que esta capa está compuesta por dos paquetes, uno de ellos es la interfaz de usuario en sí, mientras que el otro paquete existente es la lógica de la capa de presentación qué son cada uno de los componentes existentes dentro de la interfaz de usuario, con la respectiva lógica que estos llevan.

Responsabilidades de cada paquete

Centrándonos en las responsabilidades que tiene cada uno de los paquetes, comenzaremos por el paquete *WebApi*. Dentro de este paquete tenemos al paquete llamado *Controllers*, cuya responsabilidad es la de manejar las solicitudes HTTP entrantes y enviar la respuesta a quien llama (para esta versión de la aplicación, la respuesta será enviada a la UI). Este paquete, depende del paquete *IBusinessLogic*, ya que debe conocer los servicios que ofrece las reglas de negocio para poder manejar las solicitudes entrantes y poder brindar una respuesta.

También, dentro de *WebApi* tenemos el paquete *Models*, el cual contiene DTOs (*Data Transfer Object*).

Por último, dentro de este paquete, tenemos al paquete *Filters*, los cuales nos permiten ejecutar código antes o después de determinadas fases en el procesamiento de una solicitud HTTP. Es decir, nos permite interferir una determinada solicitud antes o después de que llegó a nuestro Controller. En el caso de nuestra solución, se utilizó el filtro de autorización (*Authorization Filter*), con el fin de poder ejercer un determinado control sobre aquellas operaciones que requieren estar logueado como administrador para poder ejecutarlas.

En cuanto al paquete *BusinessLogicInterface*, el mismo tiene la responsabilidad de exponer los servicios de las reglas de negocio, y permitir que los *Controllers* de la *WebApi* puedan hacer uso de los mismos sin conocer la implementación particular de las reglas de negocio.

Luego, tenemos el paquete *BusinessLogic*, que básicamente está compuesto por clases que implementan las interfaces expuestas en el paquete *BusinessLogicInterface*, contiene la implementación de las reglas de negocio de nuestro sistema. A su vez, este paquete, depende

del paquete *Domain* ya que debemos conocer las entidades del dominio de nuestro sistema para poder implementar las reglas de negocio, así como también de *BusinessLogicException*, ya que por medio del mismo es posible lanzar excepciones y detener la ejecución del sistema ante un comportamiento inesperado.

Podemos notar además, que el paquete *BusinessLogic* depende del paquete *IDataAccessInterface*, esta dependencia se basa en que es necesario que cada clase de las reglas de negocio contengan un objeto de la persistencia, que permita guardar y obtener objetos en el almacenamiento persistente.

Luego, tenemos que el paquete *DataAccess* depende del paquete *IDataAccessInterface*, debido a que en *DataAccess* hay clases que requieren de las interfaces definidas en *IDataAccessInterface*.

Entonces, tenemos que, un modelo de alto nivel (*BusinessLogic*) no depende de un módulo de bajo nivel (*DataAccess*), sino que ambos dependen de una abstracción bien definida como *DataAccessInterface*. Esta forma de implementación nos hace concluir que nuestra solución cumple con DIP (*Dependency inversion principle*).

Esta implementación tiene algunas ventajas que impactan en la mantenibilidad y extensibilidad del sistema, ya que si el día de mañana, cambia la forma en que se almacena la información, simplemente habrá que agregar un nuevo objeto que implemente la interfaz *IRepository* (y por tanto las operaciones), pero no será necesario realizar modificaciones a nivel de las clases de las reglas de negocio, ya que estas únicamente dependen de la abstracción, y no de implementaciones concretas.

Centrándonos en la responsabilidad del paquete *DataAccess*, el mismo se encarga del guardado de los datos del sistema. Es decir cómo se guardan y se obtienen los datos ya ingresados. Básicamente, una capa de persistencia encapsula el comportamiento necesario para mantener los objetos, es decir: leer, escribir y borrar objetos en el almacenamiento persistente.

Dentro del paquete *DataAccess* observamos que el mismo depende de las excepciones de repositorio (*Repository Exception*), esto debido a que cuando uno quiere acceder a datos que se encuentran alojados en algún almacenamiento persistente, se puede producir fallas a la hora de obtenerlos. De forma que debemos pensar que existe la posibilidad de fallas en el almacenamiento. A partir de esto, debemos diseñar una solución que esté preparada para afrontar estos flujos alternativos, y en particular, a considerar la manera en que en estos casos se debe cortar la ejecución del sistema, y mostrar al usuario un mensaje que indique el error que está ocurriendo.

Es por esto, que el paquete *DataAccess* tiene una dependencia de las excepciones de repositorio (*Repository Exception*), es decir, sabe cuándo lanzar una excepción y qué mensaje mostrar gracias a la información que obtiene de *Repository Exception*.

En cuanto al paquete *Domain*, el mismo contiene las entidades que conforman el dominio del problema a resolver, y depende del paquete *DomainException* quien le permite lanzar excepciones generalmente cuando se desea crear una entidad del dominio que no cumple con las características mencionadas en el contrato (p. ej: campos vacíos, máximo o mínimo de caracteres para un cierto campo, entre otros).

En esta versión se agregó el paquete Importation el cual posee la lógica de Reflection necesaria para realizar la importación de hospedajes desde diferentes fuentes. La lógica que se encuentra dentro del paquete, tuvo que implementarse de forma tal que fuera extensible en un futuro. Esto ya que no se conocen cuales son todos las posibles formas de importación de hospedajes, ya que el cliente deseaba que un tercero pudiera extender nuestro sistema agregando nuevas formas de importación propias.

Es por esto que la aplicación provee una interfaz mediante la cual, el tercero puede implementarla para poder tener más modelos de importación posibles.

Durante la importación de los archivos, pueden ocurrir diversos errores como: que la ruta del archivo sea inválida, que el dll no corresponda con el tipo de la interfaz, entre otros. En ese sentido decidimos crear un paquete que maneje estas excepciones: *ImporterException*, y que permita enviar mensajes a la UI acerca del problema que está ocurriendo.

Además, se pidió para esta versión que el sistema sea capaz de poder importar hospedajes desde un archivo JSON o XML, y para contemplar esto, tenemos dos paquetes: ImporterXml e ImporterJson, que contienen la lógica necesaria para importar estos dos tipos de archivo (deserializar el json o xml en un objeto).

Justificación del diseño para cada paquete

Dentro del paquete *WebApi*, tenemos el paquete *Controllers*, cuyo diagrama de clases se presenta a continuación. A las clases presentes en la versión anterior: *CategoryController*, *RegionController*, *LodgingController*, *TouristSpotController*, *ReserveController*, *SearchOfLodgingController* y *UserController*, se le suma *ImporterController*, *ReviewController* y *ReportController*. Cada una de estas clases se encarga de manejar las *requests* HTTP.

La clase *CategoryController* posee operaciones relativas a las categorías: permite obtener una dado un identificador y obtenerlas todas.

La clase *RegionController*, tiene operaciones relacionadas con las regiones, y dado que el sistema no requiere el mantenimiento de las mismas, únicamente ofrece las operaciones de obtener todas las regiones, y la de obtener una región dado un identificador.

La entidad *LodgingController*, tiene operaciones relacionadas con los hospedajes, y básicamente, cuenta con las operaciones CRUD relativas a ellos (create, read, update y delete).

La clase *TouristSpotController*, tiene operaciones relacionadas a los puntos turísticos, entre las que se encuentran: obtenerlos todos, obtener uno dado un identificador, obtener aquellos que se encuentren en una región dado un identificador y que tengan las categorías dadas en una lista de identificadores de categorías, y por último dar de alta uno en el sistema.

La clase *SearchOfLodgingController*, tiene una operación vinculada a la búsqueda de hospedajes por determinadas características, es decir la operación permite obtener los hospedajes que se encuentren disponibles dado un *DTO* que contiene los datos de la búsqueda: fecha de check in, fecha de check out, cantidad de huéspedes (adultos, niños, bebés y jubilados), y el punto turístico previamente seleccionado.

La clase *ReserveOfController*, contiene operaciones relacionadas a las reservas de hospedajes, entre las que se incluyen: obtener una mediante un identificador, dar de alta una en el sistema

dado un modelo que contiene: el hospedaje a reservar, fecha de check in, fecha de check out, cantidad de huéspedes (adultos, niños, bebés y jubilados), nombre, apellido y dirección de correo electrónico de quien reserva.

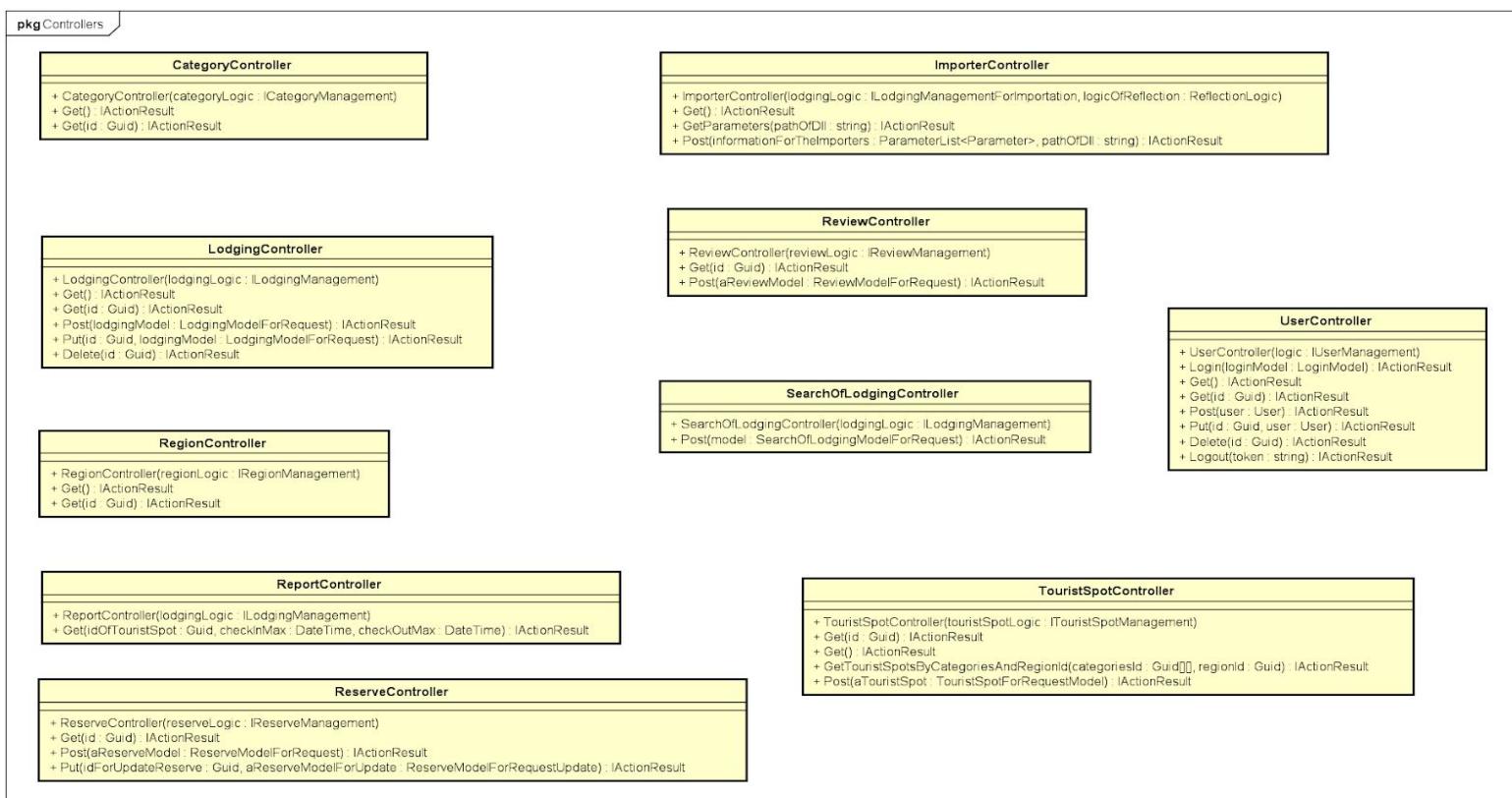
Este controlador también provee la operación de actualización de una reserva existente en el sistema, pudiendo actualizar la descripción y el estado de la misma.

El *UserController* maneja todas las operaciones referidas a los administradores del sistema, entre las que se encuentran las operaciones CRUD, y además las operaciones de *Login* y *Logout*.

También encontramos el *ReviewController* que contiene las operaciones relacionadas a obtener un review mediante un identificador y la creación de un review para un hospedaje determinado, donde los datos del usuario son cargados automáticamente por medio del identificador de reserva otorgado (ya que para poder hacer una reseña al hospedaje sé debe haber tenido una reserva). Solamente se puede hacer una reseña por reserva existente.

Luego tenemos el *ReportController* el cual tiene un único método que permite dado un rango de fechas (check-in y check-out), generar el reporte de hospedajes con las características que se mencionan al comienzo de este documento.

Finalmente, se agregó el *ImporterController* el cual permite obtener los nombres de los importadores existentes (mediante la operación get), obtener los parámetros que el importador requiere para funcionar (mediante la operación getParameters), y por último la operación que permite realizar en sí la importación del archivo, mediante la lista de parámetros requeridos y la ruta donde se encuentra el dll que se quiere utilizar.



Otro de los paquetes incluidos dentro del paquete *WebApi*, es el paquete *Models*.

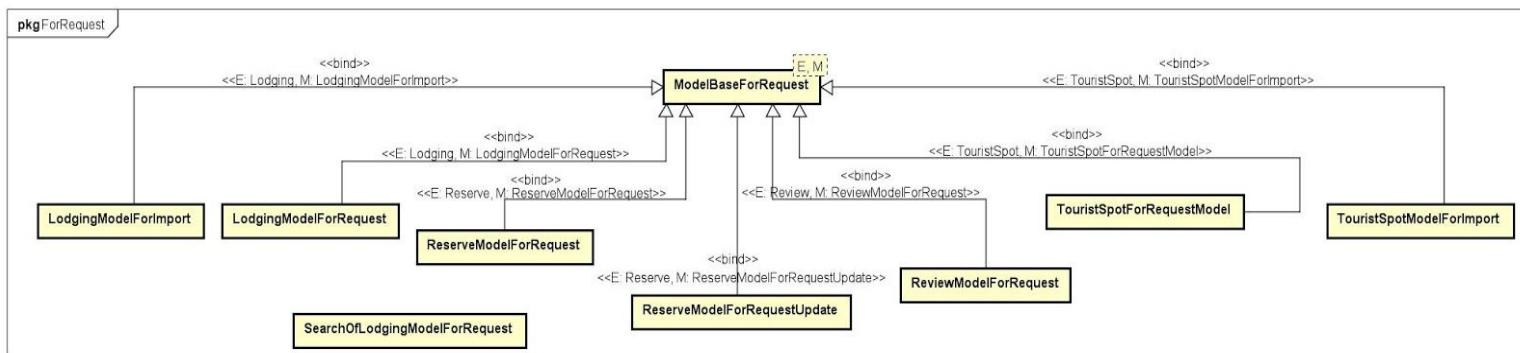
El mismo, está compuesto por DTOs (Data Transfer Object), los cuales utilizamos para diseñar la forma en que la información debe viajar desde la capa de servicios a la aplicación o capa de presentación, ya que si usáramos directamente las entidades del dominio para retornar los datos en las *requests* lo que ocurriría es que terminaríamos retornando más datos de los necesarios, y exponiendo información que realmente no deberíamos exponer. Por esta razón, decidimos utilizar modelos en lugar de retornar directamente las clases del dominio.

Otra decisión de diseño que decidimos tomar en cuanto a los *DTO*, es tener modelos para las *requests*, llamémosle *ModelsForRequest* y otros modelos para las *responses* llamados *ModelsForResponse*.

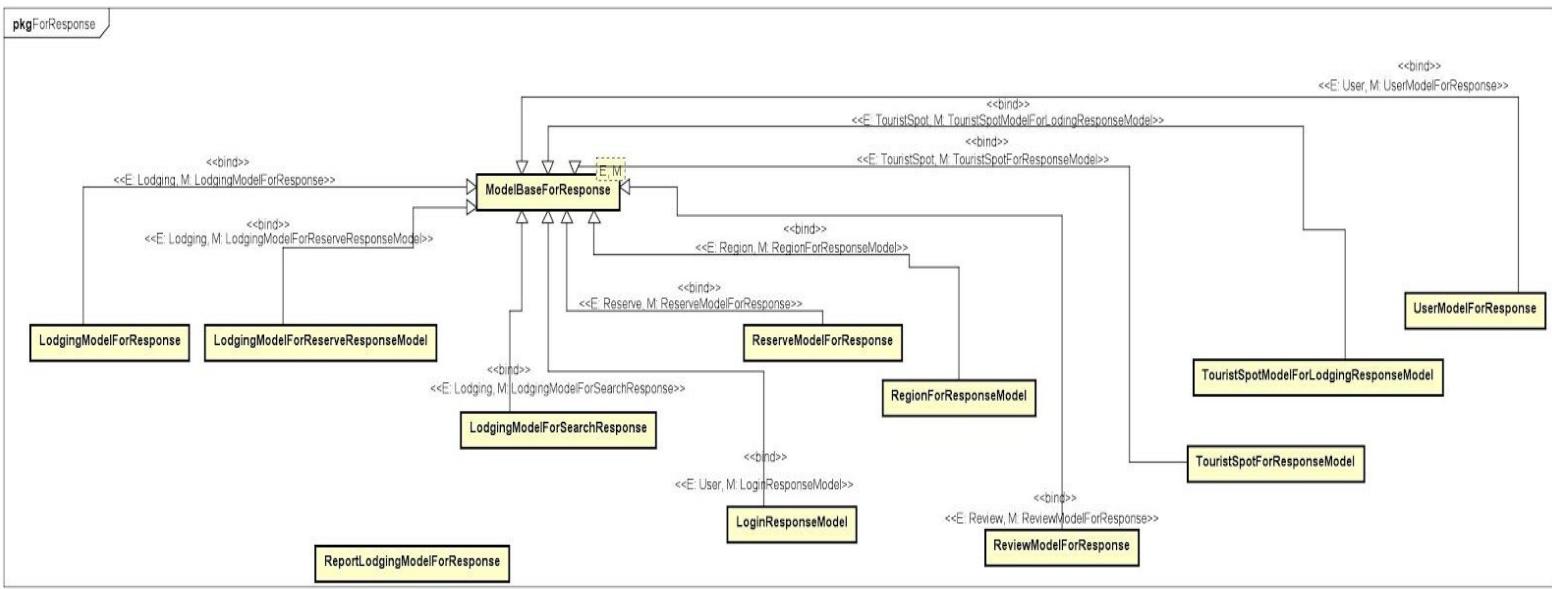
Esta decisión se basa principalmente en que en la mayoría de los casos, los datos que recibimos en una *request* para realizar una determinada operación, no son los mismos que queremos mostrar cuando nuestra *API* da respuesta.

Veamos un diagrama de clases que modela estos dos tipos de DTOs mencionados previamente:

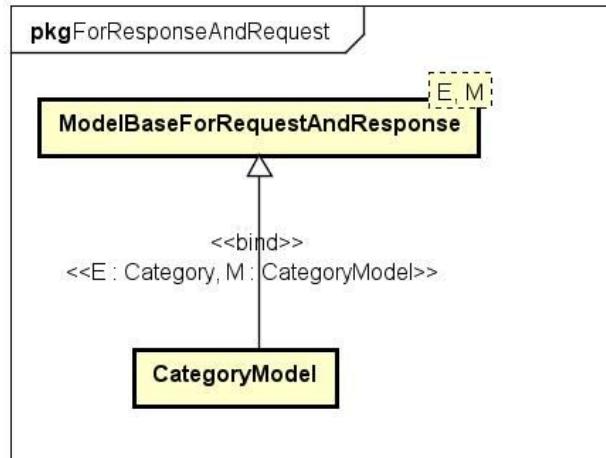
En este primer diagrama (a alto nivel que no contiene operaciones), podemos notar como todos los modelos (excepto el *SearchOfLodgingModelForRequest*, que como la búsqueda no representa una entidad del dominio, este clase modelo no va a utilizar el método que permite pasar de modelo a entidad) utilizados para *requests* heredan de una clase base llamada *ModelBaseForRequest*, la cual es genérica para una entidad (E) y un modelo (M) entre los cuales se quiere hacer el pasaje. Esta clase genérica, contiene un método abstracto que es sobreescrito por las clases que heredan y que permiten convertir un modelo a una entidad del dominio.



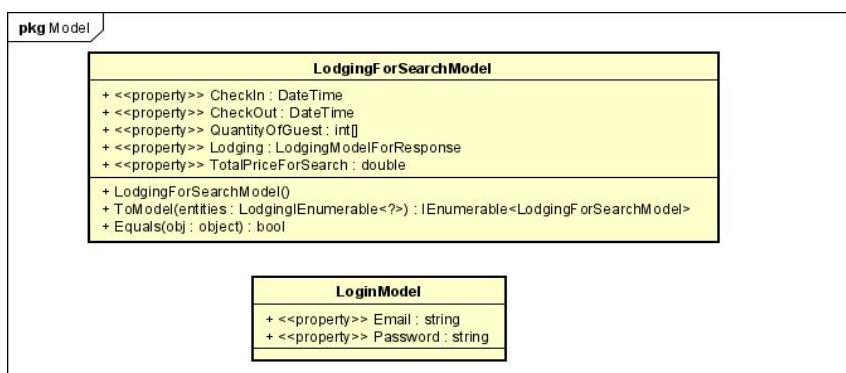
De forma análoga, tendremos lo mismo pero para las *responses*, en el siguiente diagrama podemos observar que todas las clases presentes heredan de una clase base genérica llamada *ModelBaseForResponse*, que tiene un método que toma una lista de elementos de una cierta entidad del dominio (E) y la convierte a una lista de elementos del modelo (M). Esta función utiliza otra función que convierte una entidad (E) en un modelo (M), la cual es implementada por las clases que heredan de la base.



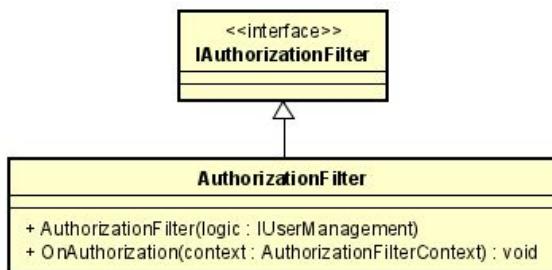
También, es posible que algunas entidades requieran del mismo modelo para las *Requests* y las *Responses*, y en ese sentido, contamos con el paquete *ForResponseAndRequest*, el cual cuenta con una clase base genérica llamada *ModelBaseForRequestAndResponse*.



También encontramos que hay dos entidades dentro del paquete `Model`, que no se encuentran diagramadas dentro de ninguno de los paquetes anteriormente mencionados. Esto debido a que son modelos, los cuales no necesitan heredar de ninguna de las clases base, ya que no hay una necesidad previa de transformar una entidad existente en la base de datos directamente a algunos de estos modelos, o viceversa, es decir un modelo transformarlo en una entidad existente en el sistema.



Dentro del paquete *WebApi*, tenemos al paquete *Filters*. Estos, nos permiten ejecutar código antes o después de determinadas fases en el procesamiento de una solicitud HTTP. Nuestra aplicación requiere que para utilizar determinadas funcionalidades se deba estar logueado en el sistema como administrador, y para mayor seguridad este control se realiza en diferentes capas: desde la interfaz por medio de una guarda que chequea si el usuario está logueado (verificando si quedo el token de sesión almacenado en el data storage), hasta en el backend por medio de un filtro de autorización (*AuthorizationFilter*), que lo utilizamos para aplicar la política de autorización y seguridad. Entonces, aquellas operaciones que requieran autenticación, deberán recibir por header el token de la sesión (llamado token), y antes de llegar a la operación que se desea realizar se ejecuta el filtro de autorización, el cual verifica que se haya recibido un token y que el mismo sea válido (es decir, que este presenta en la tabla de sesiones de usuario). En caso de que no sea válido se retorna un error, impidiendo llegar a la operación del controller que se desea utilizar.



Para que estos controladores puedan resolver las solicitudes HTTP y puedan brindar una respuesta, necesitan conocer los servicios ofrecidos por las reglas de negocio (*BusinessLogicInterface*).

Es por esto que cada uno de los *controllers*, poseen un objeto de la *BusinessLogicInterface* relativo a la entidad a la que se hace mención en su nombre (excluyendo de esto al controlador *SearchOfLodgingController*, debido a que no pertenece a una entidad pertinente del dominio del problema). Este tipo de objeto, lo que nos permite es conectar a la *WebApi* con los servicios brindados por la lógica de negocios, y hacer uso de las operaciones que ella presenta para poder resolver las solicitudes.

El hecho de que la *WebApi* dependa de una abstracción de las reglas de negocio, y no de una implementación concreta, favorece a la mantenibilidad de la aplicación, ya que estamos cumpliendo con el *DIP (Dependency Inversion Principle)*, evitando que un módulo de alto nivel (*WebApi*) dependa de uno de bajo nivel (*BusinessLogic*), y haciendo que ambos dependan de abstracciones bien definidas. Entonces, si en el futuro cambia la forma en que están implementadas las reglas de negocio, los controladores de la *WebApi* no se verán afectados en lo absoluto porque están dependiendo de una abstracción y no de las implementaciones particulares.

Además, decidimos que para cada una de las entidades que se encuentran en el Dominio, se le realice una interfaz del manejador que le permite gestionar los eventos pertinentes a esta entidad, con la correspondiente implementación.

Esta decisión fue tomada, en lo que expone el autor Robert C. Martín en su libro *Clean Code*, en la sección de Responsabilidad Única del capítulo 10.¹

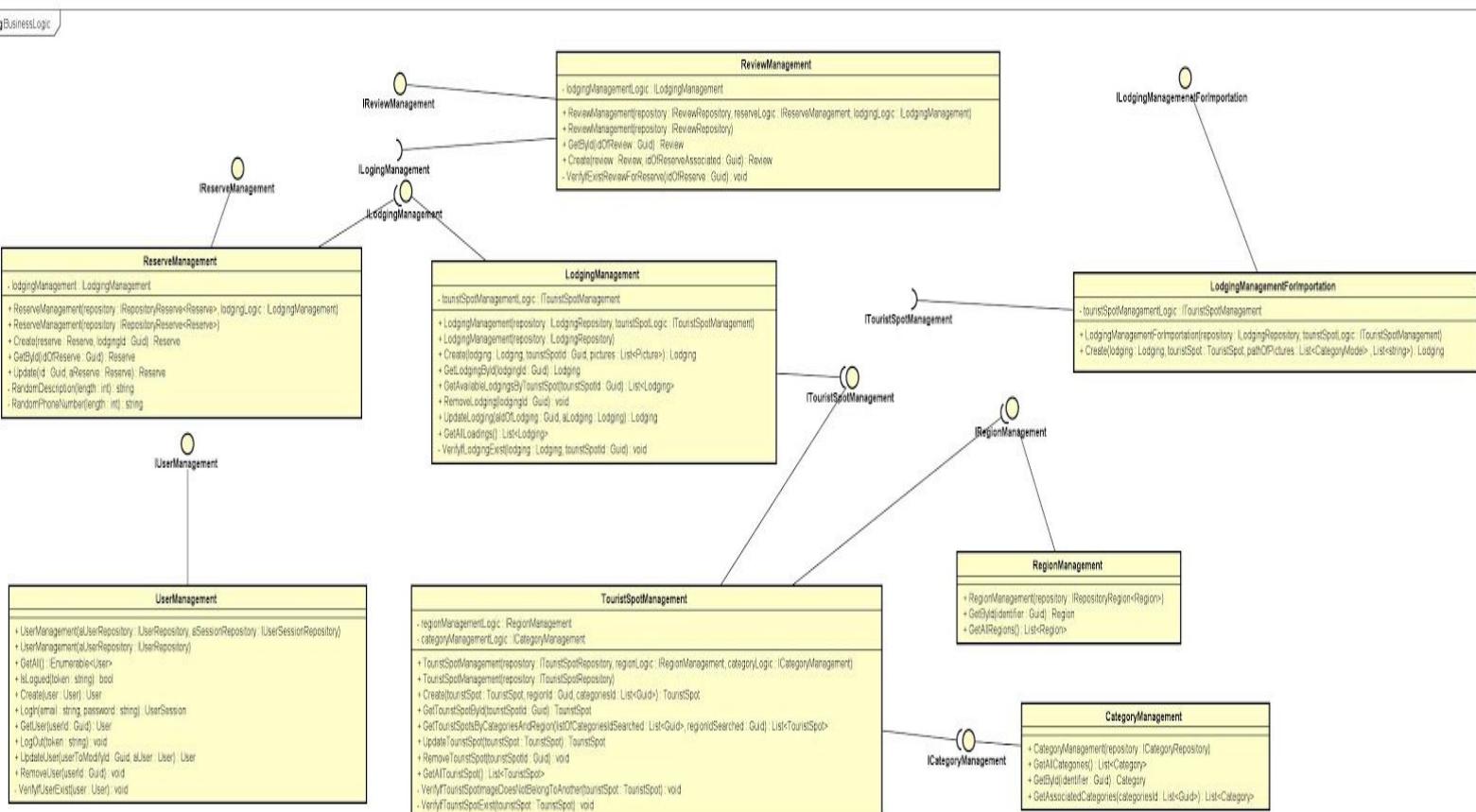
Martín nos dice que una clase debe tener uno y solo un motivo para cambiar. ¿A qué nos referimos con esto?

En particular sí se desarrollara una única clase Sistema, la cual tuviera múltiples persistencias y múltiples métodos públicos, según el principio que describió Martín, lo que sucedería es que dicha clase tendrá múltiples responsabilidades, describiendo dichas responsabilidades serían:

1. Se encargaría de verificar el formato de cada uno de los objetos que se crean.
2. Se encargaría de agregar a las diferentes tablas los diferentes tipos de objetos creados.
3. Se encargaría de eliminar a los objetos de las diferentes tablas.
4. Se encargaría de “tirar” excepciones una vez verificado el formato.

Entonces la justificación en la que nos basamos para no utilizar una única clase la cual es responsable de manejar todo y también una única interfaz la cual es responsable de exponer los servicios brindados por las reglas de negocio, es que la implementación de dicha interfaz tendría múltiples responsabilidades ya que tiene diferentes tipos de objetos asociados.

Además el logro que conlleva dividir una clase sistema en diferentes subsistemas, lleva al cumplimiento del **GRASP Controller**. Estos pequeños subsistemas, constituyen diferentes controladores, y cada uno de estos, cumplen con que la única responsabilidad que tienen es el manejo de los objetos del tipo del objeto del Dominio, los cuales están claramente identificados dentro del nombre del controlador (siendo así que en un comienzo le llamamos manejadores, pero el término correcto son controladores).



¹ Interpretado de: Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education. Página: 132-133. Página: 138.

Otro punto importante a destacar del patrón, es que nos dice que la lógica de negocios debe estar separada de la capa de servicios, esto para aumentar la reutilización de código y a la vez tener un mayor control, lo cual se cumple en su totalidad dentro del proyecto, en el cual el dominio está dividido en su totalidad de las reglas de negocio, donde se encuentra una dependencia en que las reglas de negocio dependen del dominio.

Un patrón el cual no se logra implementar dentro de la solución existente por falta de tiempo, fue el patrón de diseño Builder, el mismo es usado para permitir la creación de una variedad de objetos complejos desde un objeto fuente (Producto), el objeto fuente se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo a través de un conjunto de llamadas secuenciales a una implementación específica que extienda la clase Abstract Builder. Así, cada implementación existente de Abstract Builder construirá un objeto complejo Producto de una forma diferente deseada.

En este caso, la construcción de un hospedaje proveniente de la importación de un archivo no se realiza de la misma forma que si la construcción se realiza por medio de una llamada de la WebApi, ambos requieren diferentes validaciones y tomar diferentes decisiones (en el caso del importador el hospedaje puede venir únicamente con el id del punto turístico porque ya existe en la base de datos, o en otro caso, si no existe el punto turístico, se pasan los datos completos del mismo para que pueda ser agregado a la base de datos y poder asociarlo con el hospedaje en cuestión). Esto último no sucede con una creación de hospedaje desde una llamada de la WebApi, ya que a la hora de la creación únicamente se admite además de los datos del hospedaje, el id del punto turístico en donde se encuentra el hospedaje (no un punto turístico que el sistema no conoce), y en caso de que ese id de punto turístico asociado no exista, simplemente el hospedaje no se crea.

También en consecuencia de la subdivisión de diferentes controladores (no en sentido de los existentes en la Web Api), nos lleva al cumplimiento de dos patrones diferentes. Siendo estos, el patrón **GRASP de Alta cohesión** y también el patrón **GRASP de Bajo acoplamiento**.

- Alta cohesión: Es la medida en la que un módulo de un sistema tiene una sola responsabilidad. Por ende, un módulo con alta cohesión será aquel que guarde una alta relación entre sus funcionalidades, manteniendo el enfoque a su único propósito. Nótese cómo este principio se relaciona de manera casi directa con el principio de responsabilidad única desarrollado anteriormente.

Lo cual nos lleva a decir, que sí hubiera una única clase la cual se conociera como “System”, lo que sucedería es que dicha clase tendría múltiples responsabilidades como mencionamos anteriormente, pero además tendría una baja cohesión ya que no tendría una única responsabilidad, debido a que no tendría una alta relación entre sus funcionalidades.

Lo que nos lleva a decir que cada uno de los “manejadores” tiene una alta cohesión, ya que la responsabilidad que tienen siempre está orientada al mismo tipo de objeto del dominio, es decir no tiene múltiples responsabilidades con objetos del dominio. Cumpliendo así el GRASP de alta cohesión, debido a la subdivisión en pequeños “System’s”.

- Bajo acoplamiento: Siendo el acoplamiento la relación que se guardan entre los módulos de un sistema y la dependencia entre ellos. El bajo acoplamiento dentro de un sistema indica que los módulos no conocen o conocen muy poco del funcionamiento interno de otros módulos, evitando la fuerte dependencia entre ellos. Lo que también nos lleva a

relacionarlo con el principio **SOLID abierto/cerrado** debido a que si no tenemos mucha dependencia entre los módulos, podemos extender el comportamiento de un módulo sin afectar a otro.

Este bajo acoplamiento está dado por el hecho de que los módulos de alto nivel como es el caso de la *WebApi* no dependen de los de bajo nivel como las implementaciones de las reglas de negocio (*BusinessLogic*), sino que ambos dependen de interfaces bien definidas que exponen los servicios y reducen el acoplamiento entre los módulos. De forma análoga sucede con los módulos de *BusinessLogic*, ya que los mismos no dependen de las implementaciones específicas de los módulos de *DataAccess*, sino qué sucede que dependen de las abstracciones de los mismos, sin importar cómo estos están implementados. Lo cual conduce a poder decir que cada módulo de *BusinessLogic* tiene asociado una o varias abstracciones de la persistencia.

Una de las decisiones de diseño que tuvimos que considerar en las dos versiones del sistema fue cuando empezamos a diseñar la solución, nos cuestionamos acerca de qué entidad teníamos que asignarle la responsabilidad de verificar el formato, siendo las posibles cuestiones los diferentes manejadores o la entidad del dominio. Entonces es clave notar, que contamos con la presencia del **GRASP Experto en Información, el cual es el principio básico de asignación de responsabilidades**.

Entonces, aplicando el **GRASP Experto**, consideramos realizar las validaciones del formato en cada una de las entidades correspondientes del dominio cuando se procede a la creación de las mismas, ya que son entidades que cuentan con la información necesaria para cumplir la responsabilidad. Realizando la implementación de esta forma, se mantiene el encapsulamiento, ya que los objetos utilizan su propia información para resolver tareas, además de que se logra distribuir el comportamiento entre las clases que tienen la información requerida.

En este caso, la información necesaria para cada entidad, son las propiedades que las mismas tienen, de tal forma que se pueden utilizar los datos que se poseen luego de proceder a la creación y verificar si se trata de una entidad válida.

Mediante la aplicación de este patrón en todas las entidades del dominio encontramos que se logra un diseño con mayor cohesión y así la información se mantiene encapsulada (disminución del acoplamiento). Tal como mencionamos anteriormente.

Importación de hospedajes

Para satisfacer el requerimiento de que sea posible importar hospedajes se crearon los paquetes: Importation, ImporterJson e ImporterXml. A continuación se presenta un diagrama en conjunto de las clases de estos paquetes para mostrar las relaciones existentes entre ellas.

El paquete Importation, posee una interfaz llamada IImport, en donde se define el método que obtiene el nombre del importador (GetName), otro que permite obtener los parámetros necesarios para que el importador funcione (GetParameters), y por ultimo un método llamado Import que contiene la lógica necesaria para realizar la importación del archivo desde la fuente que se desee.

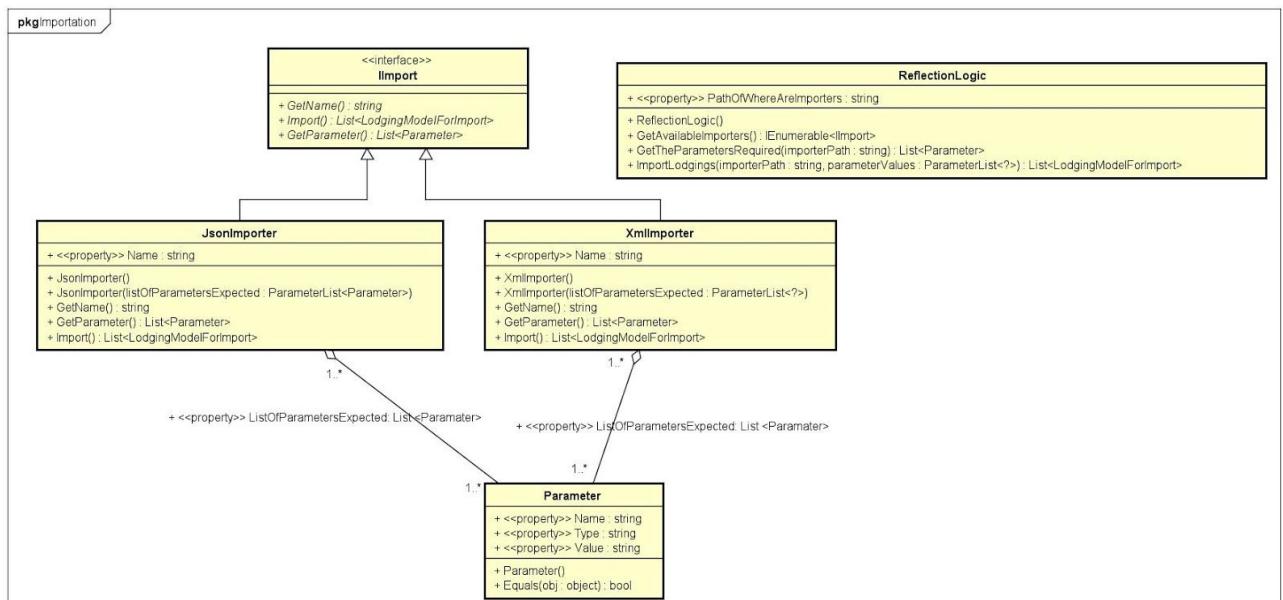
Como para esta versión se solicitó que se puedan realizar importaciones desde archivos JSON y XML se realizaron las implementaciones particulares de los importadores para posteriormente generar el dll solicitado.

Entonces si un tercero desea agregar una nueva forma de importación (ya sea csv, base de datos, entre otros), lo que debe hacer es crear una nueva clase que implemente la interfaz `IImport` (y por lo tanto tener los métodos que en ella se definen), es decir, debe indicar el nombre que posee el importador, luego debe definir los parámetros que el importador requiere para funcionar, en donde cada parámetro posee: un nombre, un tipo y el valor. Por ej. para un importador JSON se requiere un único parámetro, que puede ser definido de la siguiente manera: nombre: 'ruta', type: 'file', value: 'C/MiDirectorio/archivo.json'.

Luego de esto, el importador que el tercero desarrolle debe implementar el método `Import` de la interfaz el cual contiene la lógica necesaria para realizar la importación necesaria desde la fuente que se quiera.

Cabe mencionar algunas consideraciones sobre los importadores ya realizados, tales como:

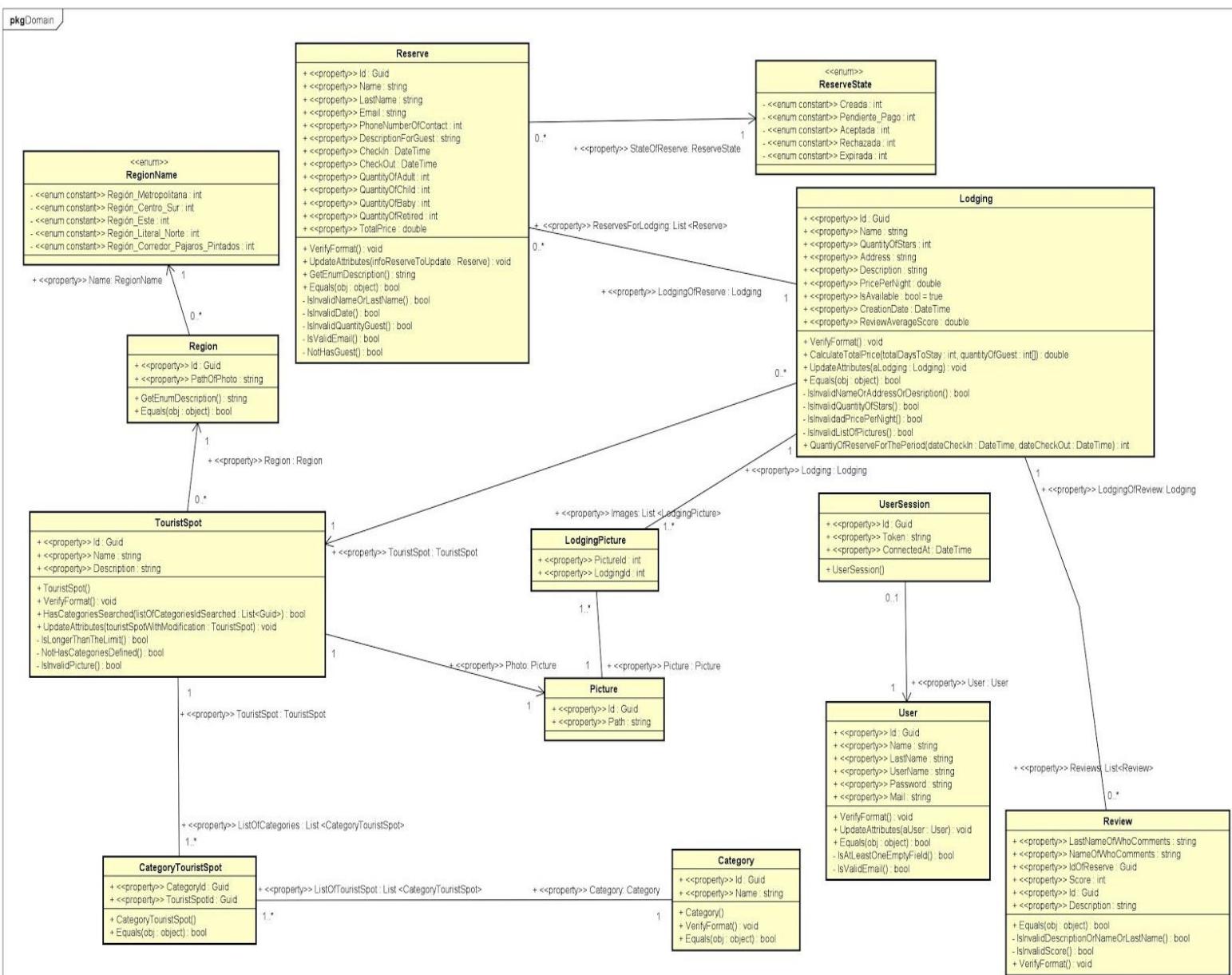
- Si se importa un archivo que contiene algunos hospedajes válidos y otros inválidos (con algún campo incorrecto), se agregan únicamente los hospedajes correctos, notificando al usuario de que algunos hospedajes no se pudieron importar correctamente.
- Si alguno de los hospedajes está formado incorrectamente, es decir no es un archivo JSON o XML (dependiendo del importador que se esté utilizando), no se agrega ninguno de los hospedajes, y se notifica el error al usuario.
- La importación de archivos XML fue implementada de tal forma que la raíz del archivo debe llamarse "Lodgings".



Dominio de la solución del problema

Dónde a las clases existentes: Reserve, Lodging, Region, TouristSpot, Category, User, UserSession, Picture, CategoryTouristSpot y PictureLodging, se agrega: Review que permite modelar las reseñas que hacen los turistas sobre el hospedaje en que se alojaron.

Cabe destacar que en cuanto al modelado de las clases que componen a este paquete, consideramos apropiado seguir un modelo de dominio no anémico, porque creemos necesario que los objetos del dominio deben tener cierta lógica, es decir, un objeto debería saber si su formato es el correcto, mostrar su estado, en el caso de un punto turístico saber si tiene determinadas categorías. Utilizando un modelo de dominio anémico, va en contra un poco de la idea básica del diseño orientado a objetos, el cual es combinar los datos y comportamientos en una única unidad, ya que en este modelo se tratan de representar los objetos del dominio pero sin comportamiento².



² Fowler, M. (2003, noviembre 25). AnemicDomainModel. Recuperado 25 de septiembre de 2020, de <https://martinfowler.com/bliki/AnemicDomainModel.html>

Mecanismo de acceso de datos

Para la implementación del acceso de datos se utilizó el *Repository Pattern*. El repositorio es una Fachada (Facade) que abstrae el dominio (o capa de lógica de negocio) de la persistencia. Se comporta como una colección (como un IList o un ICollection) escondiendo los detalles técnicos de la implementación.

Este patrón nos da la ventaja que nos permite utilizar las operaciones relacionadas a los datos, sin saber detalle alguno de la implementación. Esto conduce a que se esté dependiendo de abstracciones que cómo consecuencia produce una mayor extensibilidad, ya que en nuestra aplicación los datos se persisten en una base de datos relacional, pero bien podrían ser almacenados en: un servicio externo, en archivos XML o simplemente en memoria, y para otros módulos de más alto nivel como la BusinessLogic sería totalmente indiferente, porque el mismo solo depende la abstracción y no de ninguna implementación concreta.

La forma en que decidimos implementar este patrón, es mediante un repositorio genérico (interfaz IRepository), el cual provee los servicios básicos de lectura y acceso a datos, conocidos como operaciones CRUD. Siendo estas Create, Read, Update y Delete. Lo que conlleva el uso de estas operaciones es resumir las funciones requeridas por un usuario para crear y gestionar datos. Además de esto las ventajas que introduce la utilización de las operaciones CRUD son la reunión en un solo elemento de configuración del software todas las acciones básicas que se realizan sobre una entidad de dominio. De igual manera mejora la reusabilidad del código, evitando la duplicación del mismo evitando generar diferentes implementaciones particulares de los repositorios por separado, lo cual conllevaría la repetición de código en abundancia, en las operaciones en común entre los repositorios específicos.

Luego, nuestro sistema cuenta con una clase genérica llamada *BaseRepository*, la cual se encarga de implementar las operaciones mencionadas previamente que se exponen en la interfaz IRepository, es decir, esta clase implementa de forma genérica la interfaz IRepository.

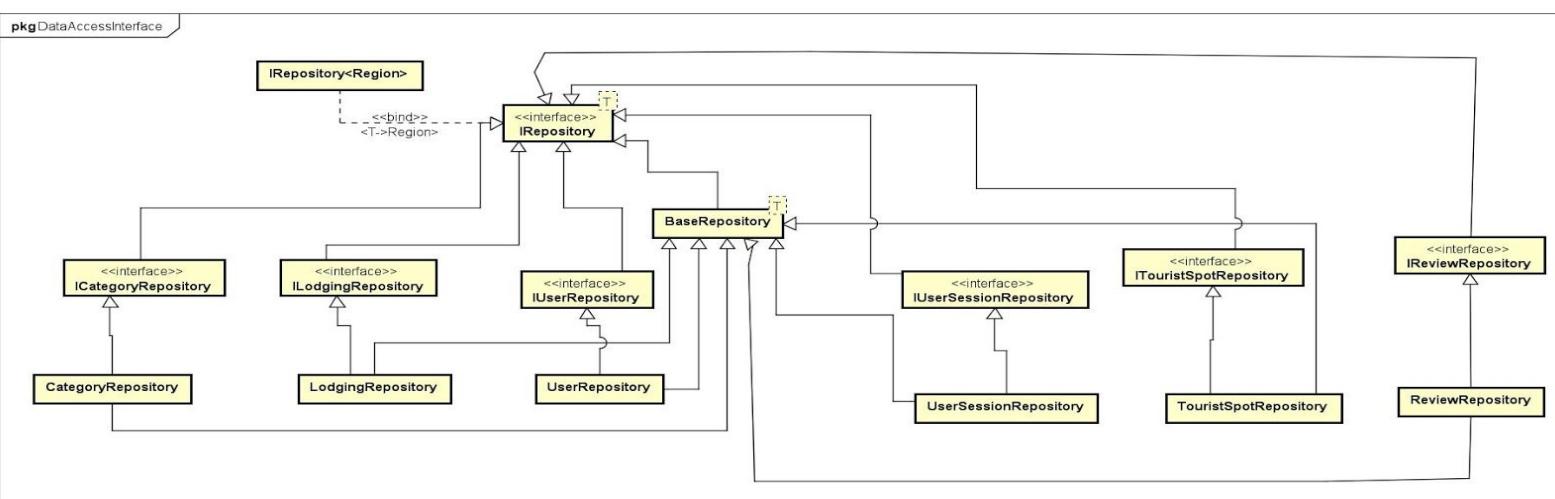
Pero con solo tener las operaciones CRUD no nos alcanza para desarrollar nuestra solución, en muchas ocasiones queremos obtener un determinado registro dada una condición, obtener registros por un determinado atributo que no sea el identificador, entre otros.

A raíz de esto, tenemos disponibles dos opciones para resolver ese tipo de solicitudes: la primera opción es la de solo manejar un repositorio genérico para cada entidad que se quiera persistir, y utilizar únicamente las operaciones CRUD. Esto significa que si queremos traer determinados registros por un determinado filtro, en primer lugar debemos hacer uso de la operación *getAll()* para obtener todos los registros de la base de datos, y luego en memoria filtrar sobre ese listado utilizando una determinada condición, lo cual no es muy bueno en términos de performance, ya que estamos trayendo todos los registros a memoria para luego recién aplicar el filtro correspondiente.

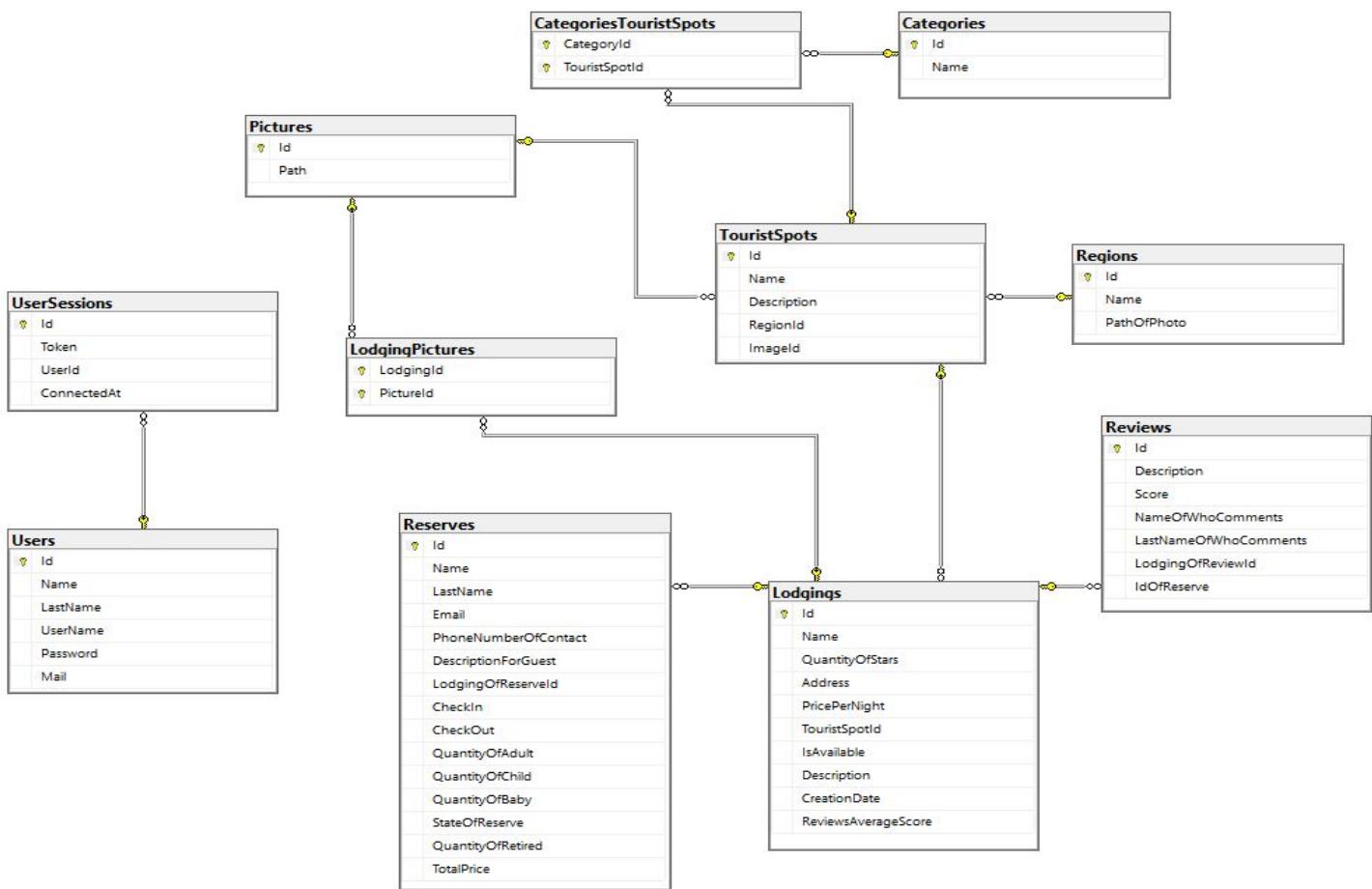
La otra opción disponible que fue por la que nos inclinamos es la de implementar para cada entidad que requiera de operaciones particulares, un repositorio particular con su correspondiente interfaz que exponga las operaciones del repositorio. Para cada entidad, lo diseñamos mediante una interfaz en donde se exponen las operaciones particulares que queremos que tenga el repositorio, esta interfaz hereda de la interfaz base genérica IRepository (donde se exponen las operaciones CRUD), y es implementada por una clase concreta que hereda de la clase llamada *BaseRepository* (clase en donde se implementan las operaciones

CRUD). Como resultado de esto, tenemos para cada entidad que requiera operaciones particulares, una interfaz que expone los servicios de cada repositorio en concreto, además de poseer las operaciones CRUD, y por medio de esta, podemos desde las clases de las reglas de negocio hacer uso de las operaciones y evitar el filtrado en memoria tal como se mencionó previamente.

Para observar las relaciones mencionadas previamente, se actualizó el siguiente diagrama de clases (a alto nivel), agregando el repositorio para almacenar reviews. No se incluyen los métodos contenidos dentro de cada paquete, ya que lo que se desea mostrar es la relaciones de jerarquía de herencia e interfaces.



Estructura de las tablas en la base de datos



Actualizando la solución, la misma pasa a contar con 11 tablas, agregando la tabla *Review*. Donde cada una de ellas representa a una entidad del dominio, excepto las llamadas: *CategoriesTouristSpots* y *LodgingPictures*, que representan las relaciones entre: Categorías y Puntos Turísticos, y entre Hospedajes e Imágenes. Como la relación entre estas entidades es N a N, se crea la tabla antes mencionada, que nos permite obtener dada una categoría los puntos turísticos con los que se relaciona (y viceversa), así como también, dado un hospedaje poder obtener sus fotos (y viceversa).

La tabla *CategoriesTouristSpots* al relacionar categorías y puntos turísticos posee dos claves foráneas: una hacia *Categories* y la otra hacia *TouristSpots*.

Mientras que la tabla *LodgingPictures*, al relacionar hospedajes con sus imágenes, posee dos claves foráneas: una hacia *Pictures* y la otra hacia *Lodgings*.

Las restantes tablas representan a las entidades del dominio, es por esto que tenemos una tabla *Users*, que contiene todos los datos de los usuarios administradores, y como clave primaria un ID.

Luego, la tabla *UserSessions*, mantiene almacenada la sesión de los usuarios mientras la misma esté activa, es decir, una vez que el usuario hace *logout*, se elimina su sesión de esta tabla. Tiene una clave foránea hacia el usuario administrador que se encuentra logueado en el sistema.

La tabla *Reserves*, permite alojar toda la información relativa a una reserva de un hospedaje. Al tener que guardar el hospedaje que el cliente ha reservado, posee una clave foránea hacia la tabla *Lodgings*.

La tabla *Lodgings*, alberga toda la información relacionada con los hospedajes de nuestro sistema. Posee una clave foránea hacia la tabla *TouristSpots*, debido a que cada hospedaje se caracteriza por estar ubicado dentro de un punto turístico

Luego, la tabla *TouristSpots*, posee toda la información relativa a los puntos turísticos de nuestro sistema. Posee una clave foránea hacia la tabla *Regions*, debido a que cada hospedaje se caracteriza por estar ubicado dentro de una región. También posee una clave foránea hacia la tabla *Pictures*, esto debido a que cada punto turístico debe contar con una imagen.

La tabla *Regions*, guarda la información a las regiones presentes en nuestro sistema. Cabe aclarar que en esta tabla no se permite la inserción de elementos, debido a que unos de los requerimientos que se indicaron fue que las regiones son fijas y no variarán con el tiempo.

La tabla *Pictures*, la utilizamos simplemente para guardar las imágenes, cada imagen tiene un id que la identifica y un *path* que representa la ruta de acceso a dicha imagen.

La tabla *Categories*, posee la información relativa a las categorías de los puntos turísticos de nuestro sistema.

Por último la tabla *Reviews*, posee la información relacionada a las reviews que realizan los turistas sobre los hospedajes en donde estuvieron, esta entidad se agregó para la nueva funcionalidad lanzada en esta versión.

Manejo de excepciones

En cuanto al manejo de errores, se prefirió utilizar excepciones frente a devolver códigos de error. Básicamente ya que si se utiliza códigos de error, el invocador debe procesar el error de forma inmediata, además de generarse una gran estructura de condicionales anidados, que hacen al código más complejo y difícil de entender.

En cambio, si utilizamos excepciones, el código de procesamiento del error se puede separar del código de ruta (de la función en sí), haciendo que el código sea más sencillo de comprender.

Para obtener una solución más clara, tomamos la decisión de dividir las excepciones según las capas con las que cuenta la aplicación.

De esta forma, tenemos el paquete `BusinessLogicException`, que se encarga de manejar las excepciones relacionadas a las reglas de negocio. En el diagrama, podemos observar que dentro de este paquete tenemos tres clases:

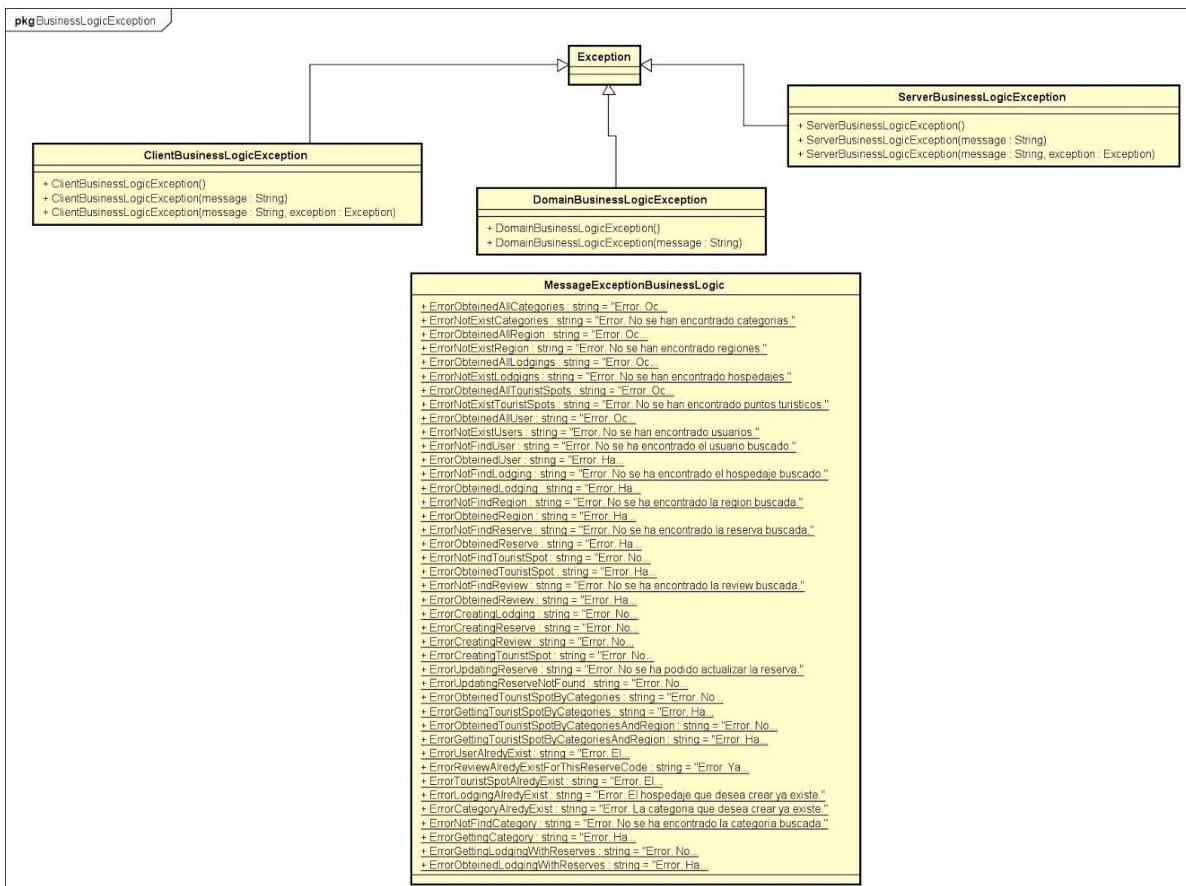
`ClientBusinessLogicException`, `DomainBusinessLogicException`, `ServerBusinessLogicException` y `MessageExceptionBusinessLogic`.

La `ClientBusinessLogicException` maneja las excepciones que de alguna manera fueron causadas por el cliente. Estas excepciones son capturadas desde los `controllers` de la `WebApi` lanzando códigos de error 404 (*Not Found*).

La `DomainBusinessLogicException` maneja las excepciones relativas a cuando un usuario desea crear algún objeto del dominio de forma inválida. Estas excepciones son capturadas desde los `controllers` de la `WebApi` lanzando códigos de error 400 (*Bad Request*).

La `ServerBusinessLogicException` maneja las excepciones relacionadas a errores internos del servidor. Estas, son capturadas desde los `controllers` de la `WebApi`, lanzando un código de error 500 (*Internal Server Error*).

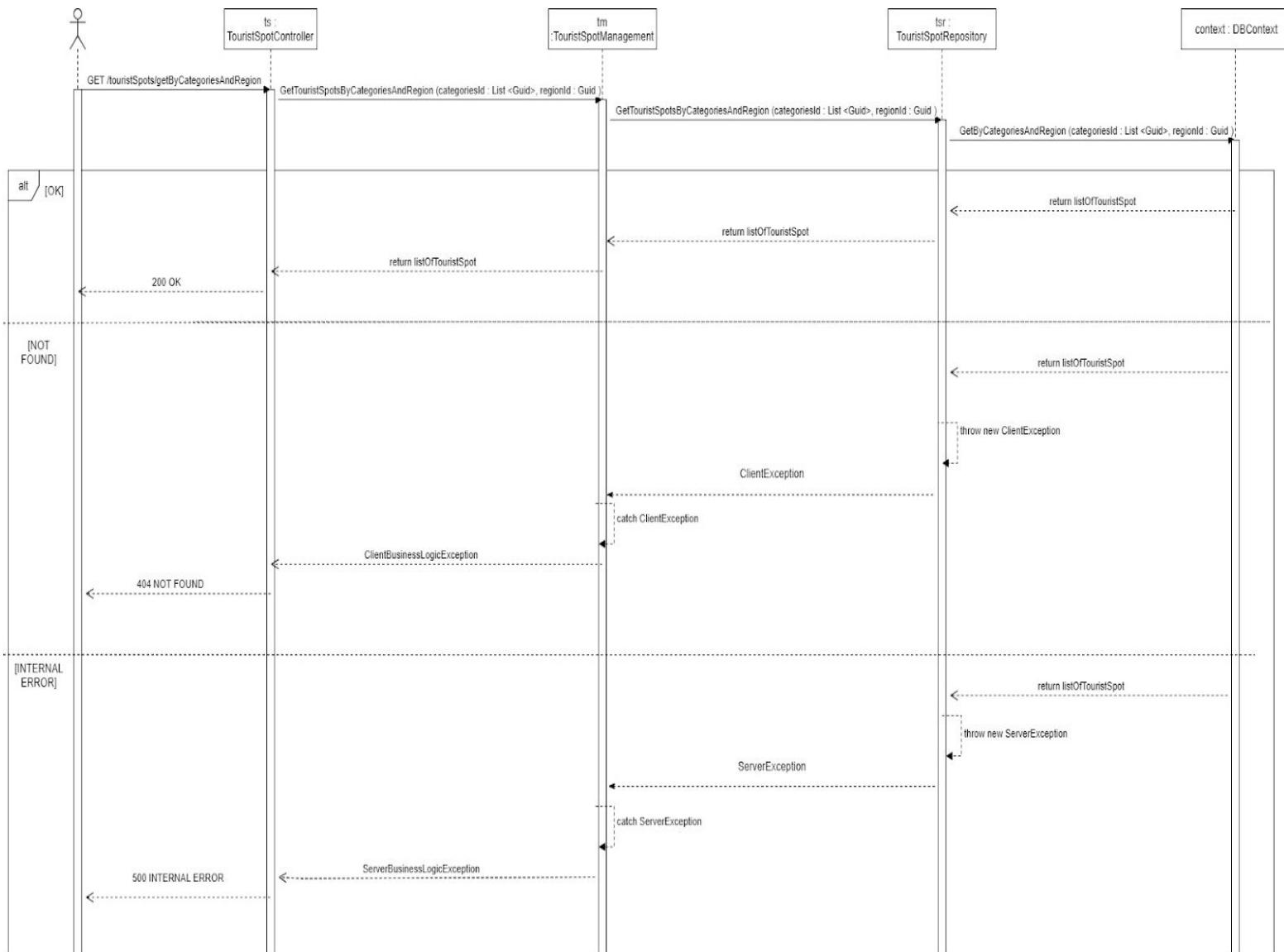
Por último, la clase `MessageExceptionBusinessLogic`, contiene únicamente métodos estáticos que contienen los mensajes que son mostrados en el momento en que se lanzan las excepciones, se realizó de esta forma debido a que de este modo es más mantenable y extensible en el tiempo, ya que si el día de mañana cambian estos mensajes de error, solo habría que realizar cambios en un lugar (en este clase), y no cambiar todas las ocurrencias de ese mensaje de error (que es lo que ocurriría si no contáramos con una clase como la `MessageExceptionBusinessLogic`).



De la misma forma se manejan excepciones para el dominio, acceso a datos y para las importaciones cuyos diagramas y explicación se pueden observar en el anexo.

Diagramas de secuencia

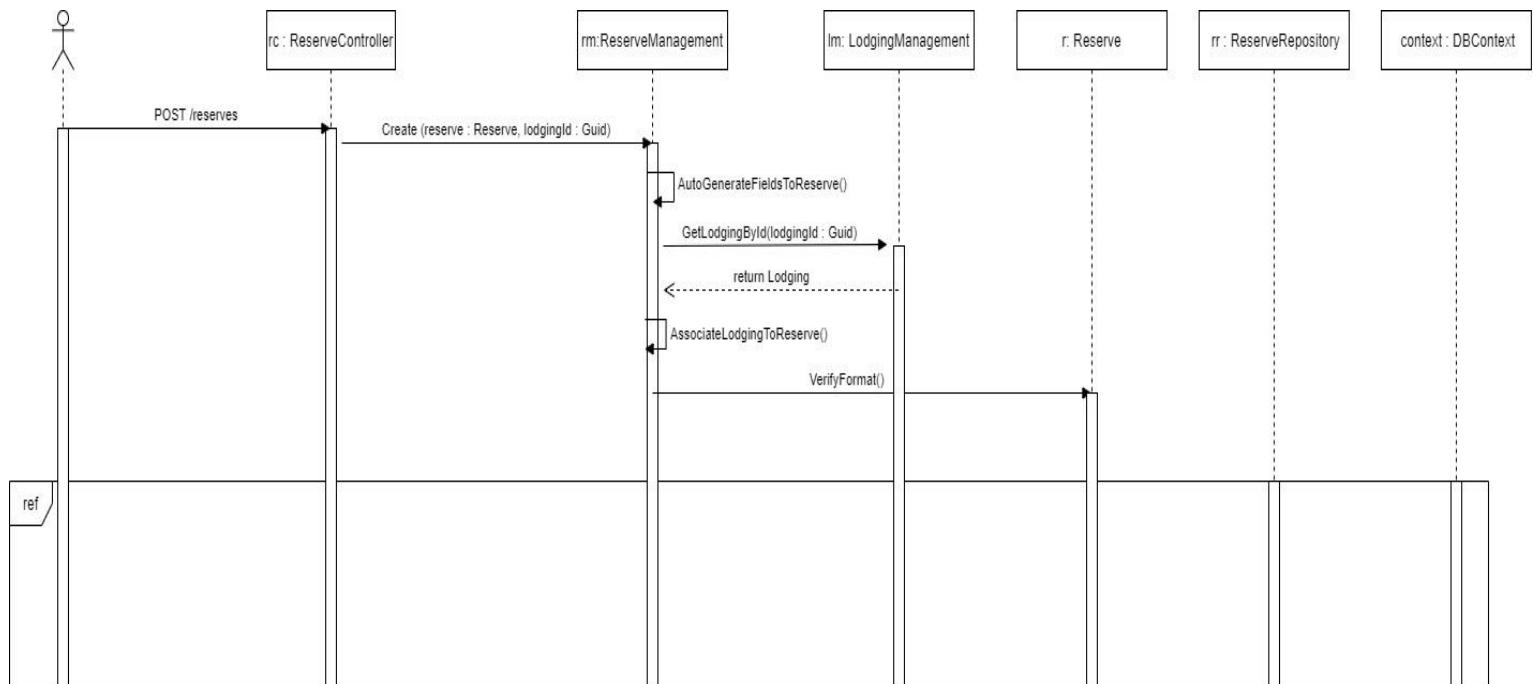
Obtener puntos turísticos por región y categorías



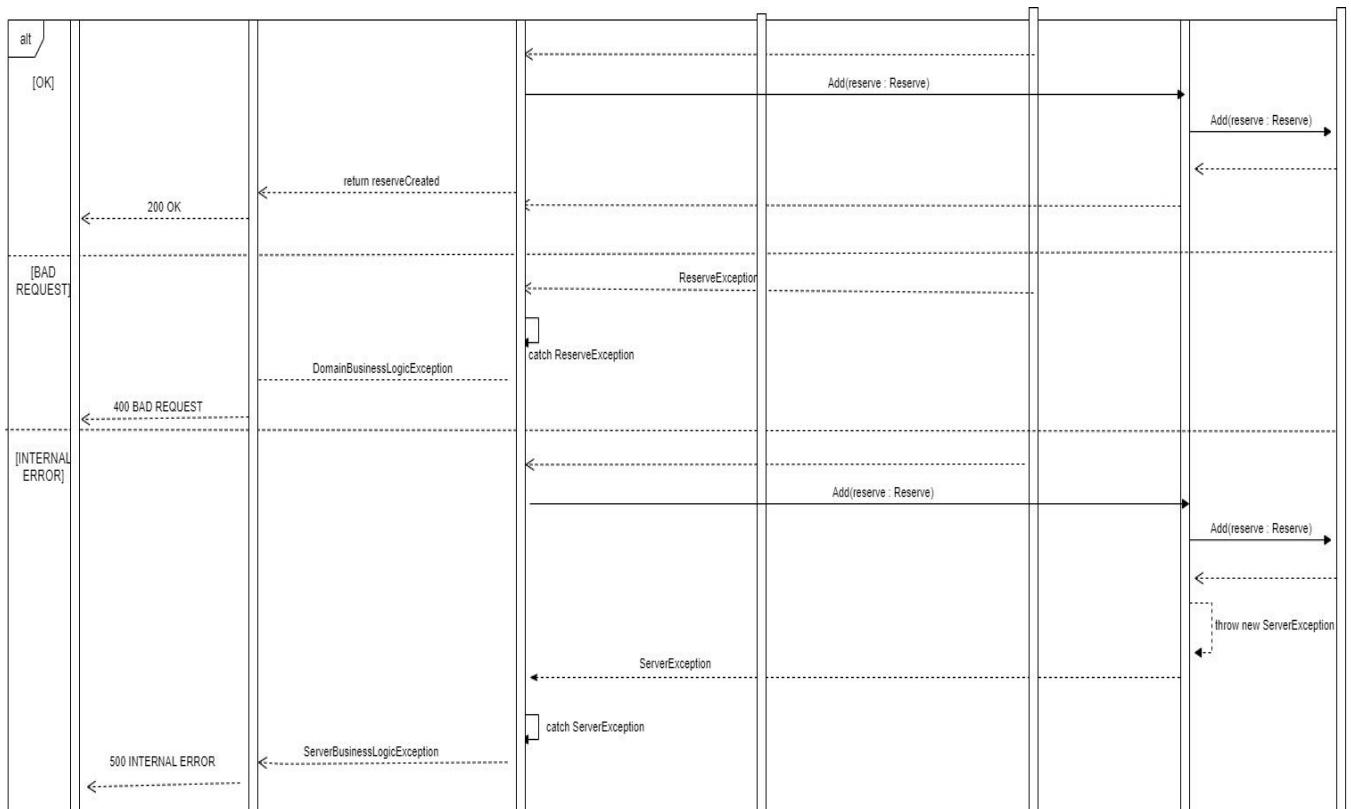
En el siguiente diagrama, se pueden visualizar las diferentes interacciones entre los objetos a la hora de obtener puntos turísticos por una región y por una lista de categorías. Se pueden observar tres casos: cuando la petición es correcta y se retorna un código 200 (OK), cuando hay un error del cliente y el recurso no es encontrado, retornando un código 404 (NOT FOUND), y por último, cuando ocurre un error interno en el servidor, devolviendo un código 500 (INTERNAL ERROR SERVER).

Ingreso de una reserva

Se puede observar las interacciones de los objetos al ingresar una reserva al sistema. Se pueden observar tres casos: cuando la petición es correcta y se retorna un código 200 (OK), cuando hay un error del cliente debido a que se colocaron algunos de los campos de la reserva de forma inválida, retornando un código 400 (BAD REQUEST), y por último, cuando ocurre un error interno en el servidor, devolviendo un código 500 (INTERNAL ERROR SERVER).



Este fragmento alt hace referencia al fragmento ref del diagrama mostrado arriba.



Generación de reporte de reservas en hospedajes

Se puede observar las interacciones de los objetos al solicitar la creación de un reporte, para cierto rango de fecha y un cierto punto turístico. Se pueden observar tres casos: cuando la petición es correcta y se retorna un código 200 (OK), cuando no se encuentran hospedajes con reservas para dicho rango de fechas y punto turístico, se retorna un código de estado 404 (NOT FOUND), y por último, cuando ocurre un error interno en el servidor, devolviendo un código 500 (INTERNAL ERROR SERVER).

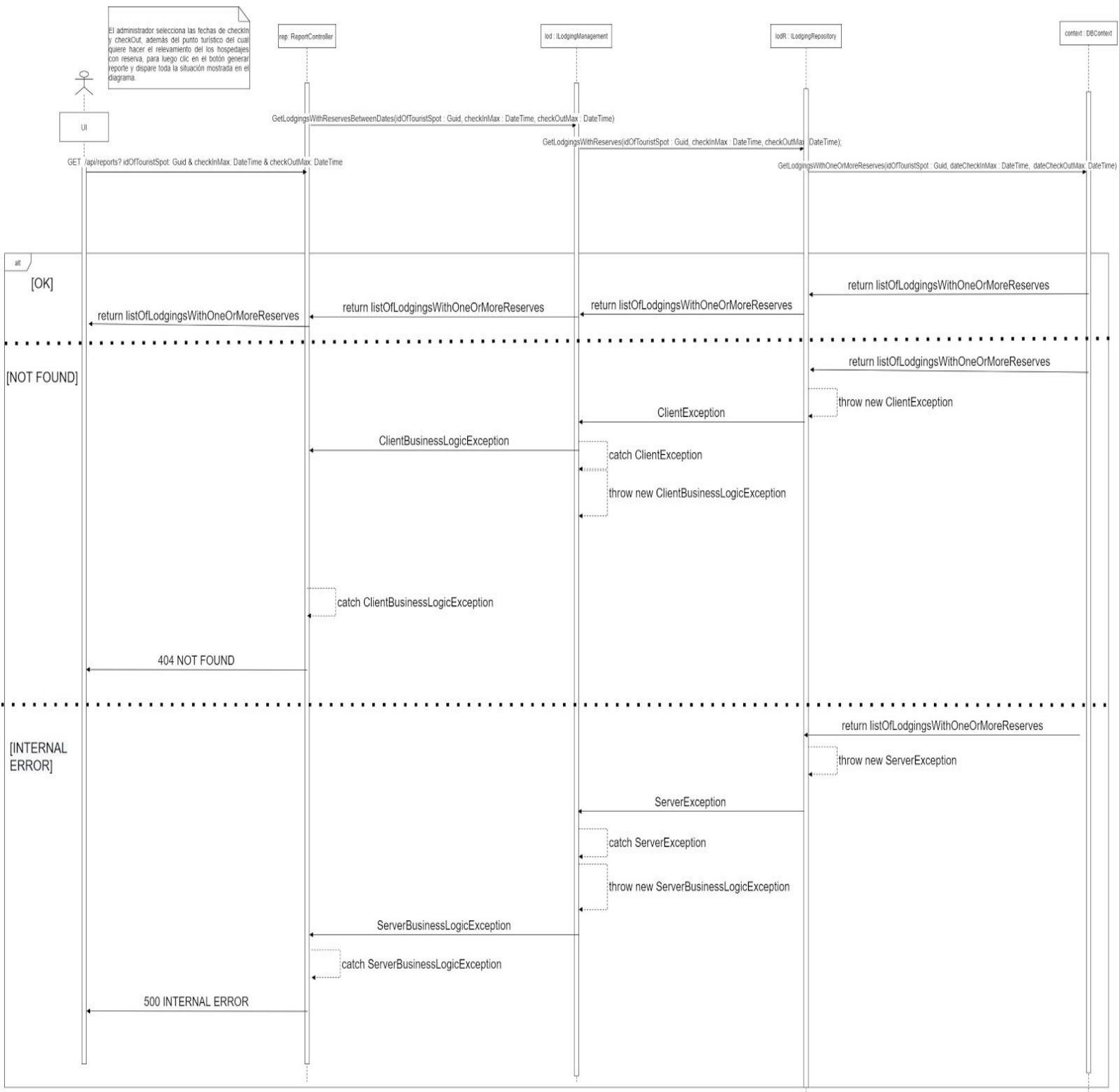
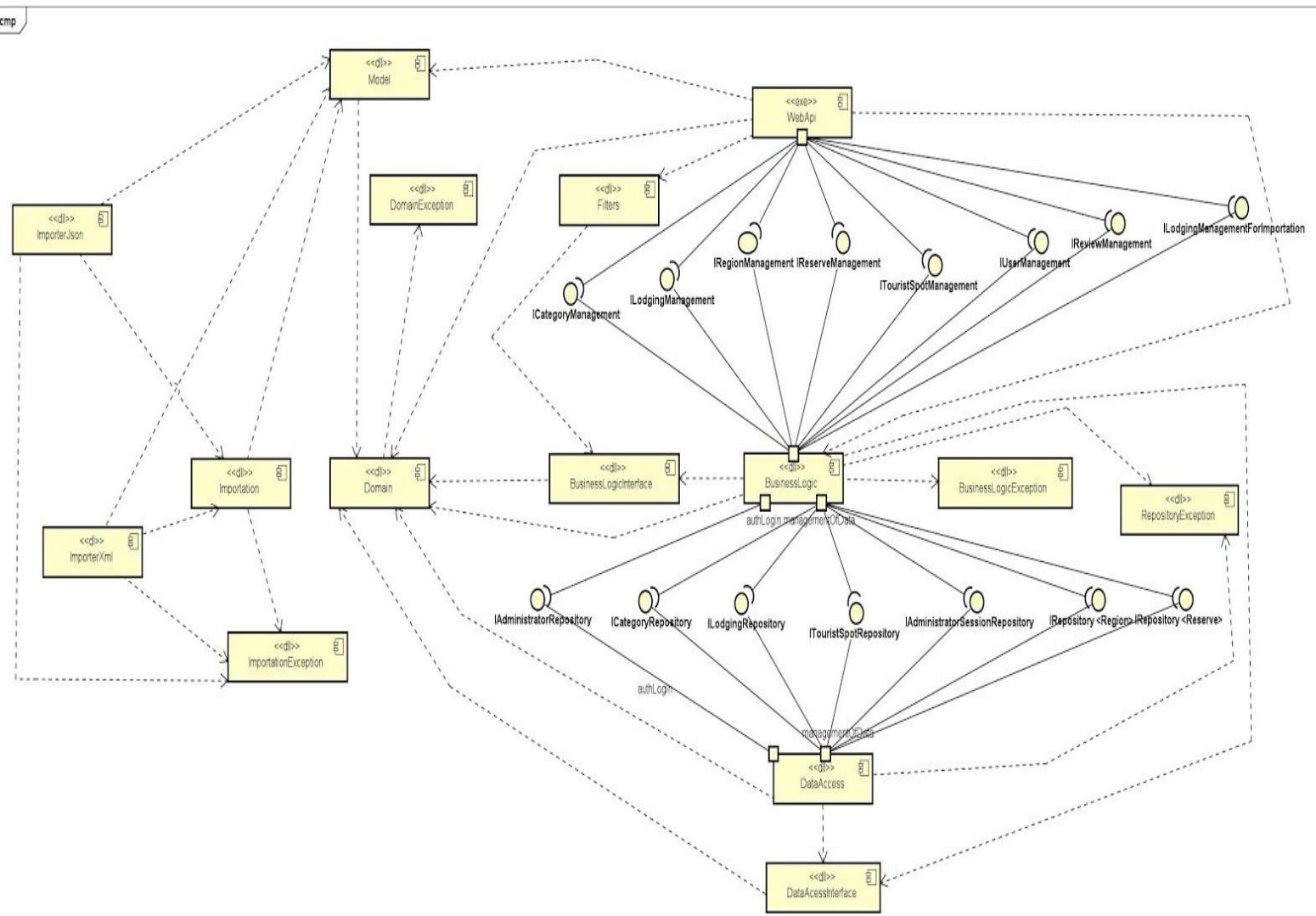


Diagrama de componentes



Comenzando desde el nivel más alto, tenemos el componente llamado *WebApi*, este existe por la necesidad de tener que procesar las diferentes solicitudes que se reciben. El mismo, para dar respuesta a estas solicitudes, necesita hacer uso de la lógica de negocios, pero no de una forma directa, sino que por medio de una abstracción de las reglas de negocio, que evite la dependencia directa con las mismas. Una vez la solicitud se encuentra procesada en la lógica de negocios, esta operación puede conllevar a determinadas consultas a un determinado espacio de almacenamiento, lo que da lugar al componente llamado *DataAccess*. Por el mismo motivo que se explicó lo que sucede entre la *WebApi* y la *BusinessLogic*, surge la necesidad de tener el componente *DataAccessInterface*, funcionando como una indirección entre *BusinessLogic* y *DataAccess*, de manera que no se utilicen implementaciones específicas, y sean abstracciones las utilizadas.

Finalmente ante la ocurrencia de un error en nuestro sistema, el mismo debe cortar la ejecución, y mostrar un mensaje de error al usuario, que sea descriptivo y específico al error acontecido, es por esto que se crea un componente de excepciones para cada una de las diferentes capas.

En esta nueva versión, se agrega el componente Importation que es donde se encapsula la lógica necesaria para resolver el requerimiento de importación de hospedajes desde diversas fuentes. A su vez como era necesario entregar un dll que permita importar hospedajes desde

archivo JSON y otro que permita hacerlo desde un archivo XML, decidimos crear dos componentes por separado para poder generar dos dlls independiente: ImporterXml e ImporterJson que poseen la lógica para importar hospedajes en esos formatos.

A su vez, en el proceso de importación de hospedajes pueden ocurrir diversos errores, y en ese sentido creamos el componente ImporterException que se encarga de manejar los errores que se pueden originar en el proceso de importación.

A su vez, se agregan las interfaces: IReviewManagement e ILodgingManagementForImportation que permiten manejar respectivamente la lógica relacionada a las reviews que los turistas ingresan acerca de un hospedaje y la creación de los hospedajes a partir del archivo importado.

Justificación y explicación del diseño en base al uso de principios de diseño, patrones de diseño en paquetes y métricas.

Principio de clausura común (CCP)

Este principio nos introduce al concepto de que las clases que cambian juntas, pertenecen al mismo grupo. Dentro de nuestro proyecto podemos observar paquetes tales como Dominio, el cual tiene las diferentes clases del negocio del dominio. Todas estas implementaciones están agrupadas en el mismo paquete debido a que, si ocurre un cambio en algunas de ellas, otras podrían verse afectadas. Este resultado de agrupar las clases en el mismo grupo, permite llevar a cabo el deseo de minimizar la cantidad de paquetes que van cambiar en cualquier ciclo de lanzamiento del producto.

Siendo un ejemplo de esto, es que un hospedaje tiene un punto turístico, por lo cual hay una asociación directa, de tal forma que si el punto turístico cambia, también el hospedaje cambiará, es por esto que se agrupan en el paquete de dominio, para que se cumpla con el Principio de clausura común.

Un dilema que hay acerca de este principio, es la premonición qué hay que tener para poder llevar a cabo el cumplimiento de este principio. Esto debido a que para poder llevárselo a cabo, nos tuvimos que anticipar y determinar cuales son los posibles tipos de cambio que son probables, de tal forma que se agruparon las clases que cambian juntas en el mismo paquete, siendo que el impacto de cambio del paquete de una versión a otra de la aplicación se minimiza. Donde el cambio producido afecta únicamente a ese paquete qué ha cambiado, afectando a todas las clases que se encuentran dentro del mismo, y no afectando a ningún otro paquete.

CRP - Common Reuse Principle

El principio mencionado anteriormente no puede satisfacerse de forma simultánea con CCP. Esto se debe a que cada principio beneficia a diferentes necesidades dentro del proyecto.

El CRP apunta a la reutilización, mientras que el CCP apunta a la mantención. El CCP se esfuerza por hacer que los paquetes sean lo más grandes posible (después de todo, si todas las clases viven en un solo paquete, solo un paquete cambiará). El CRP, sin embargo, intenta hacer que los paquetes sean muy pequeños.

Como apuntamos a un proyecto mantenible, optamos seguir el CCP, dejando un tanto de lado el Common Reuse Principle.

En particular, en la solución presentada podemos notar que no se cumple con el *Common Reuse Principle*. Dicho principio indica que las clases que no son rehusadas conjuntamente, no deben ser agrupadas.

Si bien la solución no cumple de forma estricta con lo indicado en el *Common Reuse Principle*, la implementación se realizó de forma tal que exista un reuso mínimo por la separación de capas del proyecto.

Para nombrar algún ejemplo, no se agrupan en un mismo paquete las clases que se encargan del almacenamiento de datos y las que implementan las reglas de negocio, porque claramente no son rehusadas conjuntamente, es decir si un componente utiliza las reglas de negocio, este componente no debería verse afectado ante un cambio en el componente de acceso a datos por ejemplo, y por esta razón, se debe utilizar este principio y separar estas clases que no son rehusadas.

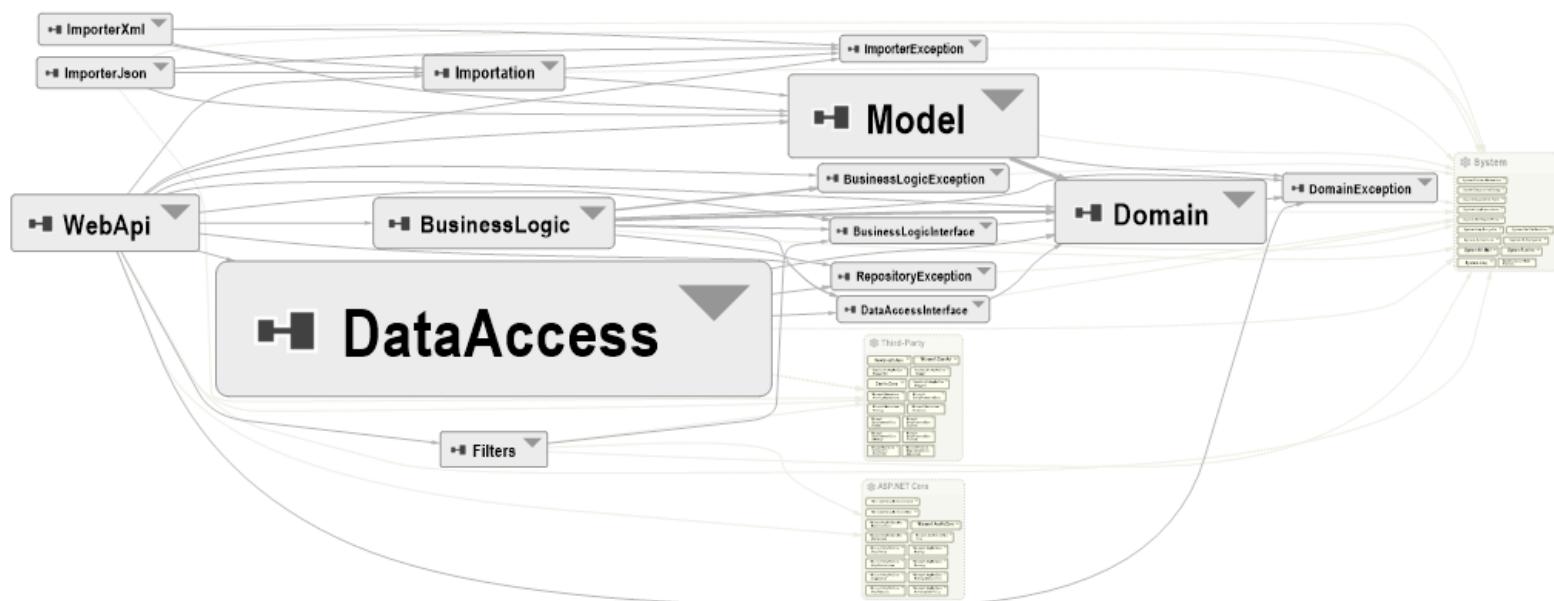
Sin embargo hay clases que se encuentran agrupadas con otras, como puede ser las clases del paquete Model, las mismas no son rehusadas conjuntamente, y ante un cambio en cualquiera de las clases del paquete Models, va a tener cómo consecuencia que las dependencias (WebAPI por ejemplo) deban pasar por el esfuerzo de actualizar y revalidar aunque no tengan ninguna relación con la clase que cambió.

Principio de dependencias acíclicas (ADP)

Este principio fue definido por el autor ya reconocido, Robert Martín el que establece que el gráfico de dependencias de paquetes o componentes no debe tener ciclos. Esto lo que implica es que las dependencias forman un grafo acíclico dirigido.

Este principio es muy importante, porque en caso de no cumplirlo se genera un caos, ya que las dependencias transitivas entre módulos harán que cada módulo dependa de todos los demás módulos, y un cambio en uno de ellos, termina impactando en toda la solución.

Se puede observar que el siguiente grafo es acíclico dirigido, implicando que las dependencias de la solución no forman un ciclo.



Principio de dependencias estables (SDP)

Este principio nos indica que se debe depender en la dirección de la estabilidad. Siendo que la estabilidad está relacionada con la cantidad de trabajo necesario para realizar un cambio. Cuando se requiere mucho trabajo para cambiar algo, quiere indicar qué se trata de algo estable.

Analizaremos si nuestra solución cumple con el principio de dependencias estables, tomando en cuenta el grafo de dependencias y la métrica de inestabilidad de cada uno de los *namespaces* de la solución.

¿Cuales son las dependencias existentes? ¿Cumplen con el principio SDP?

Matriz de dependencia

Paquetes	BusinessLogicInterface	DataAccessInterface	Importation	Models	Filters	BusinessLogic	DataAccess	ImporterXml	ImporterJson	ImporterException	Domain	RepositoryException	DataManagerException	BusinessLogicException	WebApi
BusinessLogicInterface											X				
DataAccessInterface											X				
Importation				X						X					
Models											X		X		
Filters	X														
BusinessLogic	X	X									X	X			X
DataAccess		X									X	X			
ImporterXml			X	X						X					
ImporterJson			X	X						X					
ImporterException															
Domain													X		

RepositoryException															
DomainException															
BusinessLogicException															
WebApi	X		X	X	X						X			X	

Estas dependencias se disponen en el sentido: fila depende de columna.

De tal manera que encontramos las métricas de estabilidad que nos brinda la herramienta **NDepend**. Dónde encontramos que las métricas que utilizaremos para esta sección son el cálculo de la métrica de inestabilidad, ya que dicha herramienta nos muestra únicamente el cálculo de esta métrica. De tal forma que las otras métricas existentes como **Ca** (acoplamiento aferente) y **Ce** (acoplamiento eferente), se pueden deducir de la matriz presentada anteriormente. Dónde:

- **Acoplamiento aferente**: indica el número de clases fuera del paquete que dependen de clases del paquete analizado (dependencias entrantes).
- **Acoplamiento eferente (dependencias salientes)**: indica el número de clases de dentro del paquete que dependen de clases de fuera del paquete.
- **Inestabilidad**: Indica cuán inestable es un paquete, refiriéndose a cuán fácil es cambiarlo dependiendo de cuántos paquetes lo utilizan.

$$= \begin{matrix} - \\ + \end{matrix}$$

Métricas de inestabilidad

- BusinessLogicInterface → $I = 0.44$
- DataAccessInterface → $I = 0.45$
- Importation → $I = 0.88$
- Models → $I = 0.73$
- Filters → $I = 0.94$
- BusinessLogic → $I = 0.98$
- DataAccess → $I = 0.99$
- ImporterXml → $I = 1.00$
- ImporterJson → $I = 1.00$
- ImporterException → $I = 0.5$
- Domain → $I = 0.35$
- RepositoryException → $I = 0.22$

- DomainException → $I = 0.21$
- BusinessLogicException → $I = 0.19$
- WebApi: → $I = 1.00$

Con la matriz de dependencias antes mostrada, y con los cálculos de las métricas de inestabilidad, se puede concluir que la solución **cumple con el principio de dependencias estables**, es decir, la dependencia se da en el sentido de la estabilidad. O dicho de otra forma se está dependiendo de los paquetes cuya métrica de inestabilidad es menor que la del paquete dónde estamos posicionadas. Es decir que si un paquete tiene menor métrica de inestabilidad, quiere decir que es más estable, y por lo tanto debo depender de él.

Principio de abstracciones estables (SAP)

Este principio nos introduce al concepto de que *los paquetes estables son aquellos que deben ser paquetes abstractos*. Dónde los paquetes más estables son aquellos que otros paquetes dependen de él y él no depende de otros. Entonces un paquete con muchas dependencias entrantes es muy estable porque requiere una gran cantidad de trabajo para reconciliar cualquier cambio con todos los paquetes dependientes, por lo cual deben tender a ser abstractos (mediante interfaces o clases abstractas). Mientras que los paquetes inestables deben ser concretos.

Observando la métrica de inestabilidad en cada uno de los paquetes de la solución, podemos notar que paquetes estables como *BusinessLogicInterface* ($I = 0.44$) y *DataAccessInterface* ($I = 0.45$), son abstractos (están compuestos por interfaces), mientras que paquetes inestables como *WebApi*, *BusinessLogic*, *DataAccess* son concretos, cumpliendo con el principio que se describe.

En conclusión, ¿en qué nos favorece la aplicación de este principio? Tal como R.Martín dice en su libro *Agile Principles, Patterns, and Practices in C#*, los beneficios que encontramos, es que mediante la aplicación de que los paquetes estables sean abstractos, ayuda a que su estabilidad no impida que estos paquetes se extiendan, mientras que por el lado de que un paquete inestable debe ser concreto permite que fácilmente cambie, ya que su inestabilidad permite cambiar fácilmente.

Encontramos que cuanto más paquetes sean difíciles de cambiar, menos flexibles será el diseño. Entonces los paquetes altamente estables en la parte inferior de la red de dependencias pueden ser muy difíciles de cambiar, pero según OCP no tienen porque ser difíciles de extender.

Si sucediera que los paquetes estables que se encuentran en la parte inferior son abstractos, serían fácilmente extensibles, de forma que podemos tener una aplicación compuesta por paquetes inestables (fáciles de cambiar) y paquetes estables (fáciles de extender).

Abstracción vs Inestabilidad

El gráfico de abstracción vs inestabilidad nos ayuda a detectar qué paquetes son potencialmente dolorosos de mantener (es decir, concretos y estables) y cuáles son potencialmente inútiles (es decir, abstractos e inestables).

- **Abstracción:** la abstracción es una medida de la rigidez de un sistema de software. A mayor abstracción, menor rigidez, y viceversa. Si un paquete contiene muchos tipos abstractos (es decir, interfaces y clases abstractas), y poco tipos concretos se lo considera abstracto, siendo así que dicho sistema es más fácil de extender y cambiar, que si dependiera de clases concretas. Dónde el rango para la métrica de abstracción es de 0 a 1, donde $A = 0$ indica que el paquete es completamente concreto y $A = 1$, indica que un paquete es completamente abstracto.
- **Inestabilidad :** Es la relación entre el acoplamiento eferente (C_e) y el acoplamiento total. $I = C_e / (C_e + C_a)$. Esta métrica es un indicador de la resistencia del paquete al cambio. El rango para esta métrica es de 0 a 1, donde $I = 0$ indica un paquete completamente estable e $I = 1$ indica un paquete completamente inestable.

Tal cómo mencionamos anteriormente, mediante la utilización de estas dos métricas podremos construir un gráfico en el cual podremos colocar cada paquete de nuestro proyecto en un diagrama que muestra el cuadrado unitario con la inestabilidad en el eje horizontal y la abstracción en el eje vertical.

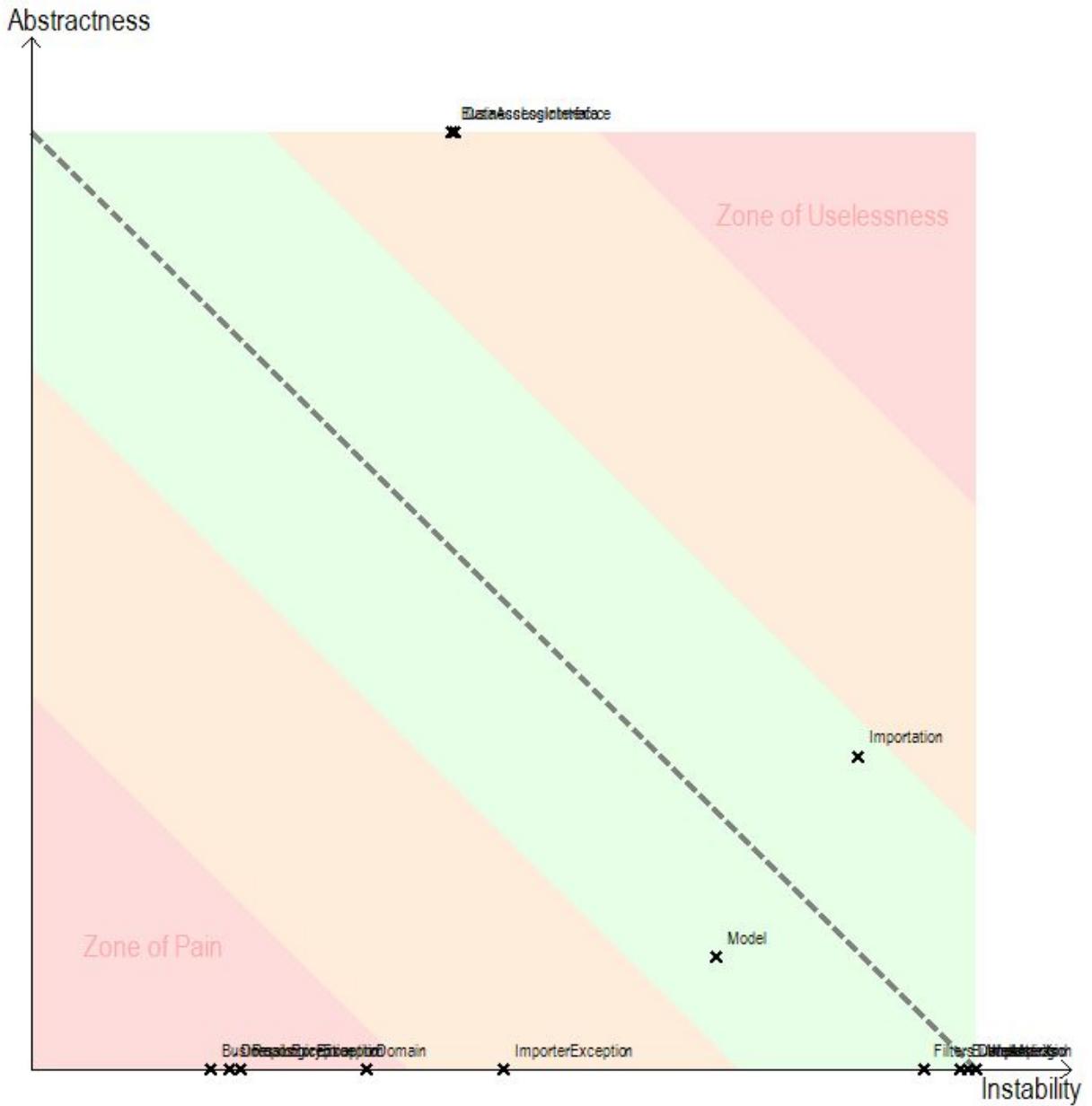
De tal forma que a partir de la gráfica podremos reformular el principio de abstracciones estables en términos gráficos, siendo así que los paquetes no deben estar demasiado lejos de la diagonal descendente del diagrama, que se llama **secuencia principal**. Esto significa que los paquetes con un bajo grado de inestabilidad deben tener un alto grado de abstracción y viceversa.

Los paquetes que residen en las esquinas aparte de la Secuencia Principal presentan problemas específicos: la esquina inferior izquierda se llama Zona de Dolor, ya que sus habitantes son estables y concretos, contraviniendo así el Principio de Abstracciones Estables, mientras que la esquina superior derecha se llama el Zona de Inutilidad, la cual contiene paquetes que son muy abstractos y de los que nadie depende. Estas son las dos áreas del diagrama que deben evitarse si es posible.

Es por esto que encontramos una métrica que nos permite medir la distancia normalizada desde la secuencia principal (D). Siendo que su cálculo es el siguiente:

$$|A + I - 1| \text{ dónde el rango de valores está entre } [0,1]$$

Dónde el resultado que obtenemos al utilizar estas métricas, es un indicador del equilibrio del paquete entre abstracción y estabilidad. Un paquete directamente en la secuencia principal está óptimamente equilibrado con respecto a su abstracción y estabilidad. Los paquetes ideales son completamente abstractos y estables ($I = 0, A = 1$) o completamente concretos e inestables ($I = 1, A = 0$). El rango de esta métrica es de 0 a 1, donde $D = 0$ indica un paquete que coincide con la secuencia principal y $D = 1$ indica un paquete que está lo más lejos posible de la secuencia principal.



Analizando el gráfico de *abstracción vs inestabilidad*, podemos observar que los paquetes: *BusinessLogicException*, *DomainException*, *RepositoryException* y *Domain*, se encuentran en la denominada *Zone of Pain*. Los paquetes que se encuentran en esta zona son concretos y estables (responsable). Estos paquetes no son buenos porque no son extensibles y si cambian impactan en otros. En particular en nuestra solución, tres de los cuatro paquetes que se encuentran en esta zona, corresponden a paquetes que se encargan de manejar las excepciones de cada una de las capas del sistema. Por esta razón, estos paquetes son estables (hay varios paquetes que dependen de ellos), y a la vez son concretos (no poseen ninguna abstracción), lo que implica la violación del principio de abstracciones estables que indica que los paquetes estables son aquellos que deben ser paquetes abstractos. En el caso de las excepciones decidimos no incorporar abstracciones

porque consideramos que son paquetes que no tienen una lógica muy compleja y que no tenderían a sufrir modificaciones en el futuro.

Lo mismo sucede con el paquete *Domain* que se encuentra en esta zona, el mismo es un paquete muy concreto, y al tener la responsabilidad de modelar el dominio del problema sucede que hay muchas clases fuera de este paquete que dependen de él, lo que implica que un cambio en este paquete repercute en gran parte del sistema, por eso es que son muy estables pero tiene un costo muy alto su modificación.

Es deseable que nuestros paquetes se sentaran en la línea de secuencia. Una posición en esta línea significa que el paquete es abstracto en proporción a sus dependencias entrantes y concreto en proporción a sus dependencias salientes. En otras palabras, las clases de dicho paquete se ajustan al *D/I/P*. En nuestra solución, los paquetes *ImporterXml* e *ImporterJson* se encuentran sobre la línea de secuencia, lo que implica que no son ni muy abstractos, ni muy inestables, teniendo una relación abstracción-inestabilidad óptima.

Muy cerca de la secuencia principal se encuentran los paquetes *DataAccess* ($d = 0.01$), *BusinessLogic* ($d = 0.01$), *Filters* ($d = 0.04$), *Model* ($d = 0.11$), significando que tienen una muy buena relación abstracción-inestabilidad (no son ni muy abstractos ni muy inestables).

Luego, en una franja intermedia entre la óptima (verde) y la zona de inutilidad (rojo), en la zona de color anaranjado se encuentran los paquetes: *BusinessLogicInterface* ($d = 0.31$) y *DataAccessInterface* ($d = 0.32$), estos paquetes son totalmente abstractos (debido a que contienen las interfaces de las reglas de negocio y del acceso a datos respectivamente) y tienen un valor medio de estabilidad (0.44 y 0.45). Para estar en la zona óptima (cercana a la secuencia principal), estos paquetes deberían ser más estables (muchos dependan de ellos y ellos no dependan de muchos), esta relación de dependencias entrantes y salientes para estos paquetes no pudo ser optimizada en función de la realidad presentada.

En cuanto a la zona de poca utilidad, no tenemos ningún paquete allí, lo cual es un aspecto positivo debido a que en ella se encuentran paquetes abstractos de los cuales nadie depende.

Mejoras al diseño

Precio de la reserva

Durante la primer entrega el equipo no se había percatado de que no se estaba almacenando en la tabla de reservas en la base de datos el precio de la misma, lo que generaba que cada vez que deseaba traer una reserva de la base de datos nuevamente se calculaba el precio de la misma haciendo el flujo mas ineficiente.

Para mejorar esto, se agregó en la entidad reserva un atributo que posee el precio de la misma, y es almacenado en la base de datos, por lo que si desea traer una reserva de la base de datos, el precio ya es conocido y por lo tanto, no es necesario volver a recalcular como sucedía en la versión anterior del sistema, haciendo el flujo más eficiente.

Verbos en URI

En la primera versión del sistema, se dejaron algunos verbos en las URIs, y el estándar RESTful exige que sean utilizados sustantivos y no verbos o nombres de métodos.

Un ejemplo que se tenía era:

getTouristSpotsByRegionAndCategories para traer los puntos turísticos dada una región y una lista de categorías.

Realmente esta forma de nomenclatura es incorrecta y se asemeja a funciones de lenguaje de programación orientado a objetos. En lugar de esto, para el diseño más adecuado, utilizamos sustantivos en lugar de verbos de la siguiente manera: *ByCategoriesAndRegion*.

Endpoints, controller y tests innecesarios

En la primera versión del sistema, se realizó la construcción de la WebApi sin la interfaz gráfica, de tal forma que hubo varios EndPoints los cuales fueron construidos “de más”. Esto debido a cuando se implementa la interfaz gráfica actual (segunda versión de la aplicación), se logró observar cómo había EndPoints qué eran totalmente innecesarios, además de un controlador que venía por defecto con la creación del proyecto WebApi, que había sido eliminado, pero no se había eliminado del proyecto. Ese controlador que venía por defecto, fue eliminado correctamente del proyecto para asegurar la calidad del mismo y la extensibilidad, debido a que se hacía creer que era un controlador del sistema, pero realmente no formaba parte del mismo.

Por último se eliminaron EndPoints innecesarios, en particular se eliminó el Post de categoría debido a que la letra de la primera versión indicaba que tanto las categorías como las regiones no necesitaban un alta, sino que esto se hace directo desde la base de datos, de este endpoint también se eliminaron los test asociados a toda la creación de categorías. Además de eliminar los EndPoints de obtención de puntos turísticos por región y por categoría, ya que para la necesidad de la aplicación, se hace conjuntamente en unEndPoint ya existente conocido como *byCategoriesAndRegionId*, dónde se filtra particularmente por categorías y regiones, correspondiente con losEndPoint eliminados, se eliminaron todos los test relacionados con la obtención de puntos turísticos por él identificador de región, y todo los test de obtención de puntos turísticos por las categorías.

Extensibilidad ante nuevos tipos de huéspedes

En la primera versión del sistema, solo se tienen 3 tipos de huéspedes: adultos, bebés y niños. Para esta nueva versión se agregó la categoría jubilado, y si bien no es una mejora de esta versión del sistema, cabe remarcar de que por la forma en que estaba implementado fue muy sencillo incorporar el nuevo tipo de huésped y realizar los cálculos del precio de la reserva en base a los beneficios que el mismo posee, y que si en un futuro se agregaran nuevos tipos de huésped el sistema sería fácil de adaptar para satisfacer dicho requerimiento.

Manual de instalación

Requisitos previos:

- Si no cuenta con la característica de Windows “Internet Information Services (IIS)” activada, debe activarla.
- Luego de esto debe activar dentro de ella, los paquetes “Service World Wide Web ” y “Tools for administration Web”.
- Dentro de la WWW (World Wide Web), debe asegurarse de que se encuentren activados los paquetes de ASP.NET (esto dentro del paquete “features of application development”).
- Tener instalado ASP.NET Core 5.0 Runtime (v5.0.0) - Windows Hosting Bundle Installer. Verificar que el mismo aparece en los módulos de IIS AspNetCoreModuleV2 como módulo nativo.

Deployment BackEnd

1. Compilar la aplicación en Release.
2. Crear una publicación de nuestro sistema. Nos posicionamos sobre el paquete de WebApi y damos en publicar, dejamos la ruta por defecto.
3. Si vamos a la ruta “[WebApiProject]\bin\Release\netcoreapp3.1\” encontraremos una carpeta llamada “publish” y dentro estarán los archivos de nuestra aplicación así como también las bibliotecas de clases de otras dependencias como EFCore.
4. Copiamos todo estos archivos, y luego vamos al directorio “C:\inetpub\wwwroot” creamos una carpeta en dónde vamos a poner el FrontEnd y BackEnd, en particular dentro de esta carpeta creamos dos nuevas carpetas una para el FrontEnd, y otra para el BackEnd. Dentro de la Back, pegamos todos los archivos que teníamos copiados.
5. Por defecto no podremos ejecutar peticiones del tipo PUT ni DELETE. Para ello debemos quitar el módulo de WebDAV de nuestra aplicación. Dentro de la carpeta “publish” generada, se encuentra un archivo llamado “web.config”, el contenido es similar a este:

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <configuration>
3    <system.webServer>
4      <handlers>
5        | <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />
6      </handlers>
7      <aspNetCore processPath="dotnet" arguments=".\\AGT.WebApi.dll" stdoutLogEnabled="false" stdoutLogFile=".\\logs\\stdout" />
8    </system.webServer>
9  </configuration>
10 <!--ProjectGuid: E3772596-65C4-4544-8C10-66158F624620-->

```

Dónde en este código debemos agregar la siguiente configuración:

```

<modules runAllManagedModulesForAllRequests="false">
<remove name="WebDAVModule" />
</modules>

```

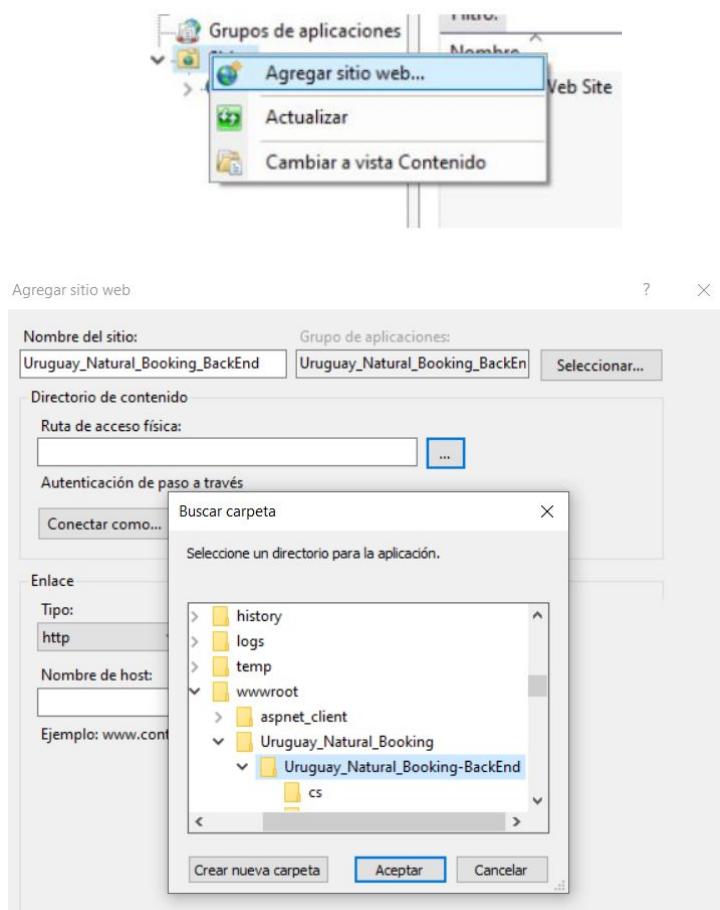
Quedando un archivo como el siguiente:

```

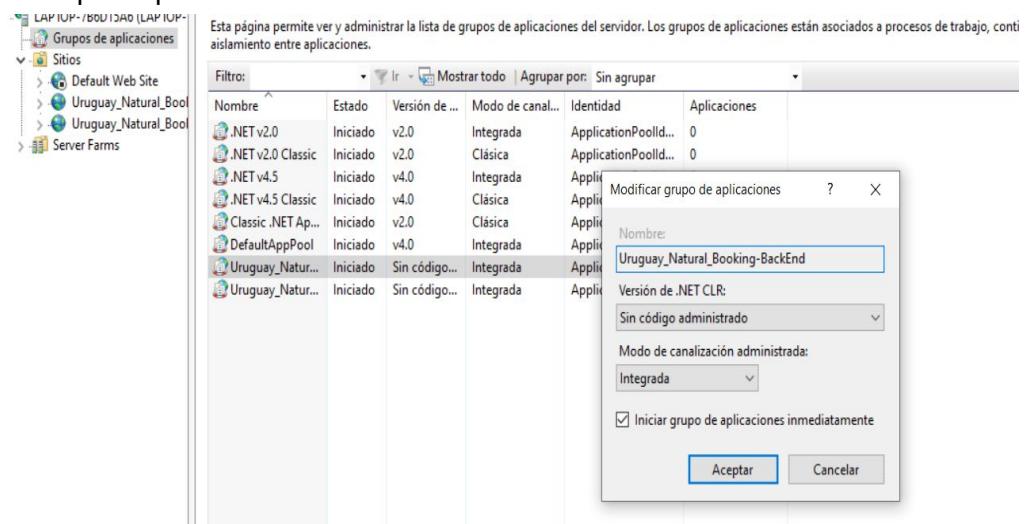
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <modules runAllManagedModulesForAllRequests="false"> ←
      | <remove name="WebDAVModule" />
    </modules>
    <handlers>
      | <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="dotnet" arguments=".\\AGT.WebApi.dll" stdoutLogEnabled="false" stdoutLogFile=".\\logs\\stdout" />
  </system.webServer>
</configuration>
<!--ProjectGuid: E3772596-65C4-4544-8C10-66158F624620-->

```

6. Ir a IIS y crear un nuevo sitio:



- El nombre del sitio es a elección, en nuestro caso lo llamamos igual que la carpeta donde creamos el BackEnd, pero esto no es un requisito necesario.
- La ruta de acceso física es la ruta a nuestra carpeta donde copiamos los archivos anteriores que contiene la aplicación, esto en el directorio "C:\inetpub\wwwroot\Uruguay_Natural_Booking\Uruguay_Natural_Booking-BackEnd"
- El puerto es también a elección. Por defecto IIS cuenta con un sitio por defecto "Default Website" que ya corre en el puerto 80. Sólo podemos tener corriendo un único sitio por puerto. Pueden poner otro puerto o dejar el 80 y luego detener el sitio por Default. En nuestro caso elegimos el puerto 6969.
- Cuando creamos un nuevo sitio, IIS nos creará un nuevo POOL o grupo de aplicaciones para ese nuevo sitio, luego de esto nos dirigiremos a grupo de aplicaciones, y haremos doble click sobre el pool recién creado qué tendrá el mismo nombre que le pusimos al sitio.



Aquí lo que haremos serán los siguientes pasos:

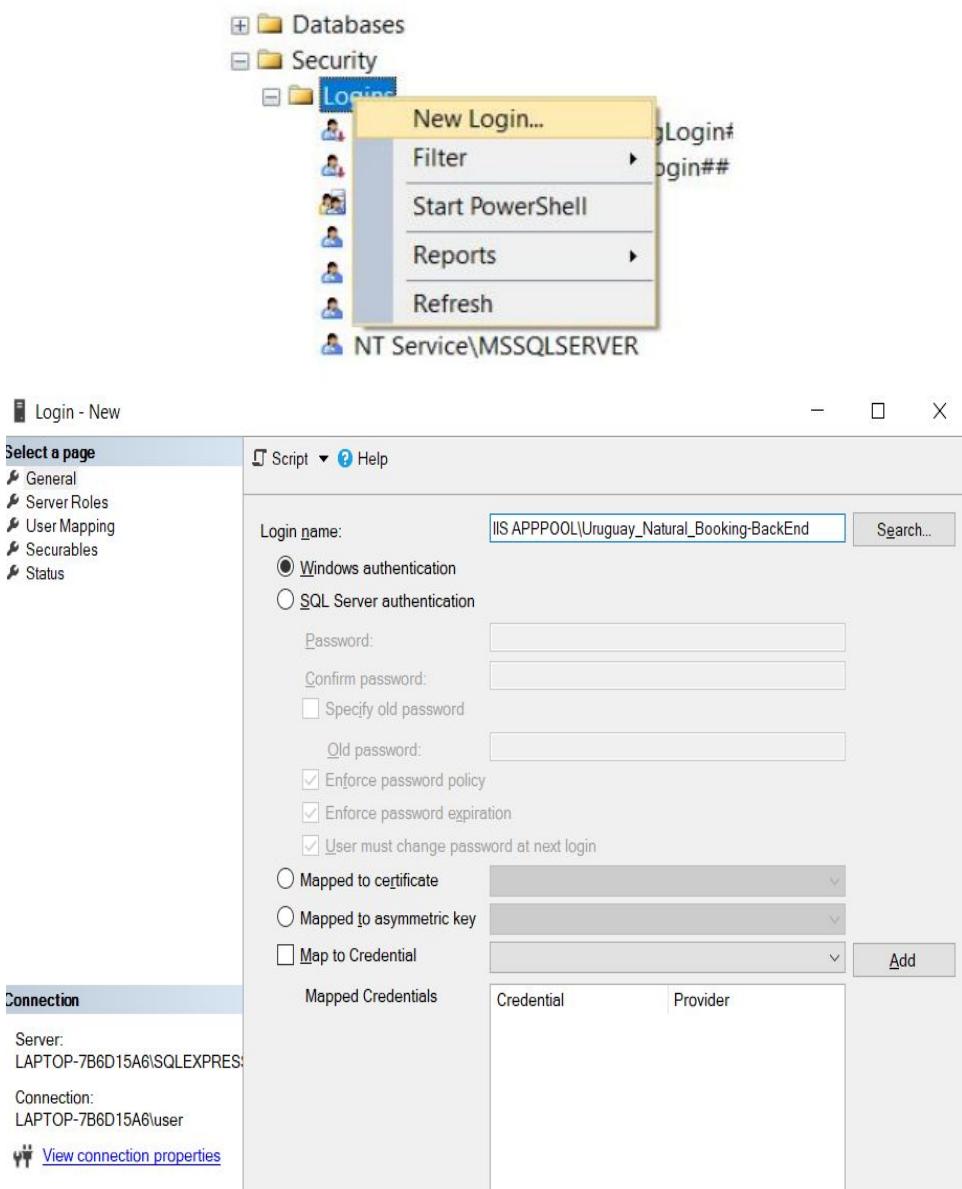
1. Quitar el CLR por defecto, y poner sin código administrado. Esto ya que .NET Core no usa el CLR por defecto configurado en IIS.

Una vez hecho los pasos anteriores, proseguimos a continuar:

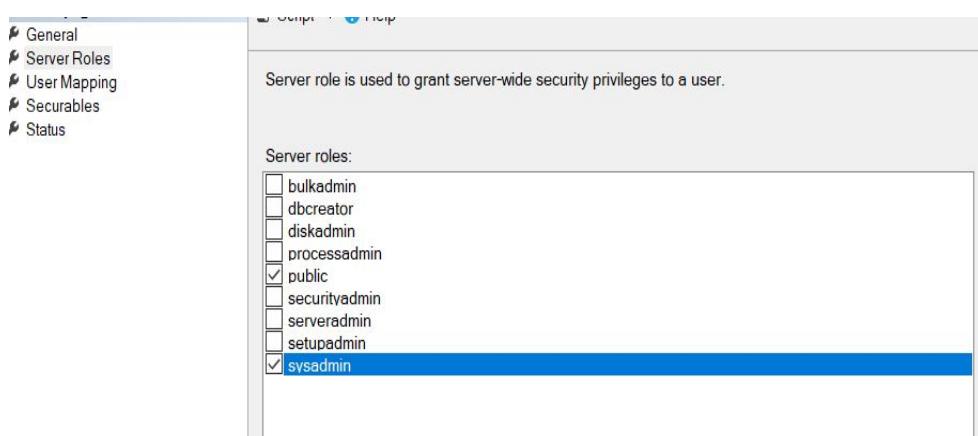
4 - Otorgar permisos al usuario del sitio de IIS sobre la base

El usuario que interactúa contra el motor de SQL Server es el usuario creado para el sitio de IIS que creamos anteriormente (IIS APPPOOL\XXXX). Debemos entonces configurar un nuevo inicio de sesión en el motor de SQL Server y otorgarle permisos para que pueda leer y escribir de a la base. Simplemente crearemos un usuario sysadmin.

A - Ir a la carpeta de Security y crear un nuevo inicio de sesión



A este nuevo usuario debemos asignarle el ROL deseado, en nuestro caso sysadmin. Una vez hecho esto, darle OK.



Luego de esto visualizamos si carga correctamente el BackEnd, introduciendo en el navegador la siguiente URL “<http://localhost:6969/swagger/index.html>”, y sí carga correctamente significa que ha quedado funcionado de manera correcta.

Deployment FrontEnd

1. Configuración URL del backend Se recomienda tener la sección de url que indica la dirección de la webAPI en un único lugar, por ejemplo en el environment-prod.ts.

```
src > environments > TS environment.prod.ts > ...
1  export const environment = {
2    production: true,
3    baseUrl: 'http://localhost:6969/',
4  };
5
```

2. Build Angular App

Ejecutamos ng build --prod

```
C:\Users\user\Desktop\DA2\233375-2333610b1\Uru_NaturalBookingFrontend\Uru_NaturalBookingFrontend>ng build --prod

chunk {} runtime.acf0dec4155e77772545.js (runtime) 1.45 kB [entry] [rendered]
chunk {1} main.e3bf0338f8a2debd92.js (main) 857 kB [initial] [rendered]
chunk {2} polyfills.00e02af16340fdc82052.js (polyfills) 36.7 kB [initial] [rendered]
chunk {3} styles.2ccad2732fb1642b6060.css (styles) 208 kB [initial] [rendered]
Date: 2020-11-26T15:56:10.859Z - Hash: 01c77b3905f110ca44d4 - Time: 76396ms
```

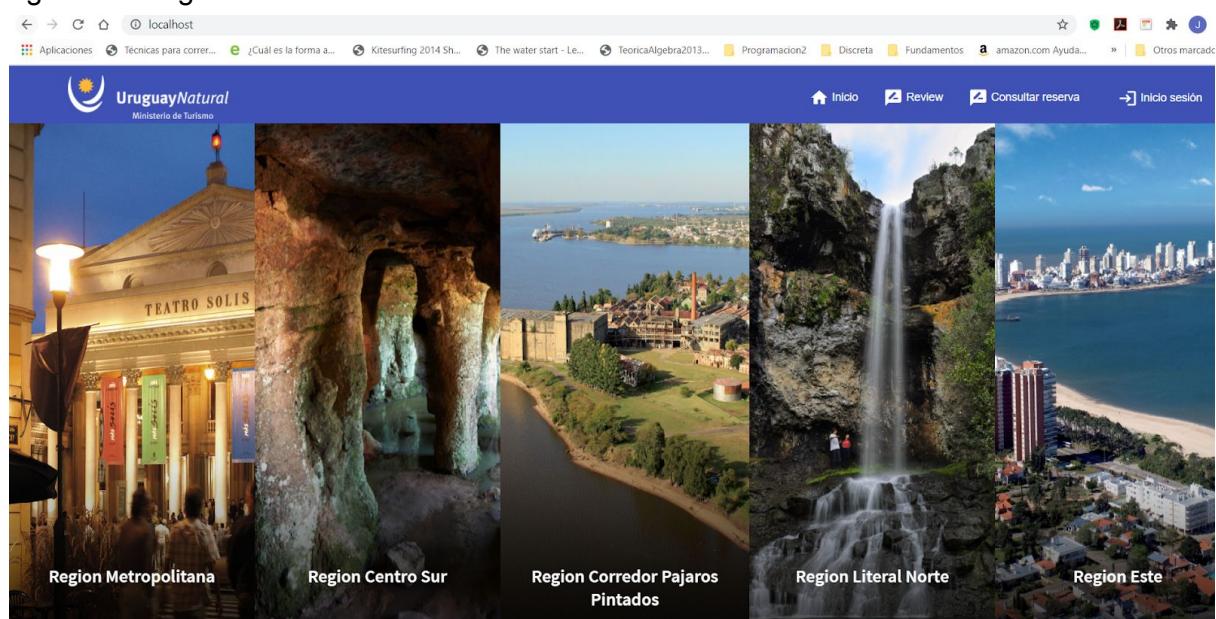
Este comando nos genera dentro de la carpeta dist (está en la raíz del sitio) una carpeta con el nombre de nuestra Angular App. Copiamos todos los archivos de esta carpeta y los pegamos en la carpeta del directorio donde anteriormente habíamos creado dos carpetas una para FrontEnd y otra para BackEnd, en este caso pegamos los archivos en la carpeta de FrontEnd.

3. Luego de esto realizamos el mismo procedimiento de ir al IIS, crear un nuevo sitio web, repetir el procedimiento de poner el nombre (en nuestro caso él mismo nombre que le pusimos a la carpeta “Uruguay_Natural_Booking-FrontEnd”), dónde se encuentran los archivos y dejar el puerto por defecto 80. Para esto debemos detener el sitio web por defecto que se crea, ya que está utilizando el mismo puerto y esto no puede ocurrir, ya que dos sitios web con el mismo puerto no pueden correr.
4. De manera opcional para qué funcione correctamente él ruteo también por medio de la URL, lo que debemos hacer es agregar en el directorio “C:\inetpub\wwwroot\Uruguay_Natural_Booking\Uruguay_Natural_Booking-FrontEnd”, un archivo web.config con el siguiente código:

```
C:\inetpub>wwwroot>Uruguay_Natural_Booking>Uruguay_Natural_Booking-FrontEnd>web.config
1  <?xml version="1.0" encoding="utf-8"?>
2  <configuration>
3  <system.webServer>
4  <rewrite>
5  <rules>
6  <rule name="Angular Routes" stopProcessing="true">
7  <match url=".*" />
8  <conditions logicalGrouping="MatchAll">
9  <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
10 <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
11 </conditions>
12 <action type="Rewrite" url="./index.html" />
13 </rule>
14 </rules>
15 </rewrite>
16 </system.webServer>
17 </configuration>
18
```

- Una vez finalizado esto, lo que debemos hacer para que funcione correctamente la funcionalidad de importar hospedajes utilizando este mecanismo es pegar los archivos de importer que se encuentran en una carpeta llamada “Importers” dentro del código fuente, en el proyecto WebAPI, copiarla y pegarla en el directorio “C:\inetpub\wwwroot\Uruguay_Natural_Booking\Uruguay_Natural_Booking-BackEnd” De igual manera se debe hacer con los archivos de importación que contienen los hospedajes.

Ya habiendo finalizado todo estos pasos, para verificar qué ha funcionado correctamente, lo que debemos hacer es acceder al navegador con la siguiente URL “<http://localhost/>”. Si carga correctamente, podrá observar y manejar su página web, tal como se muestra en la siguiente imagen.



Anexo

Mecanismo de borrado de elementos

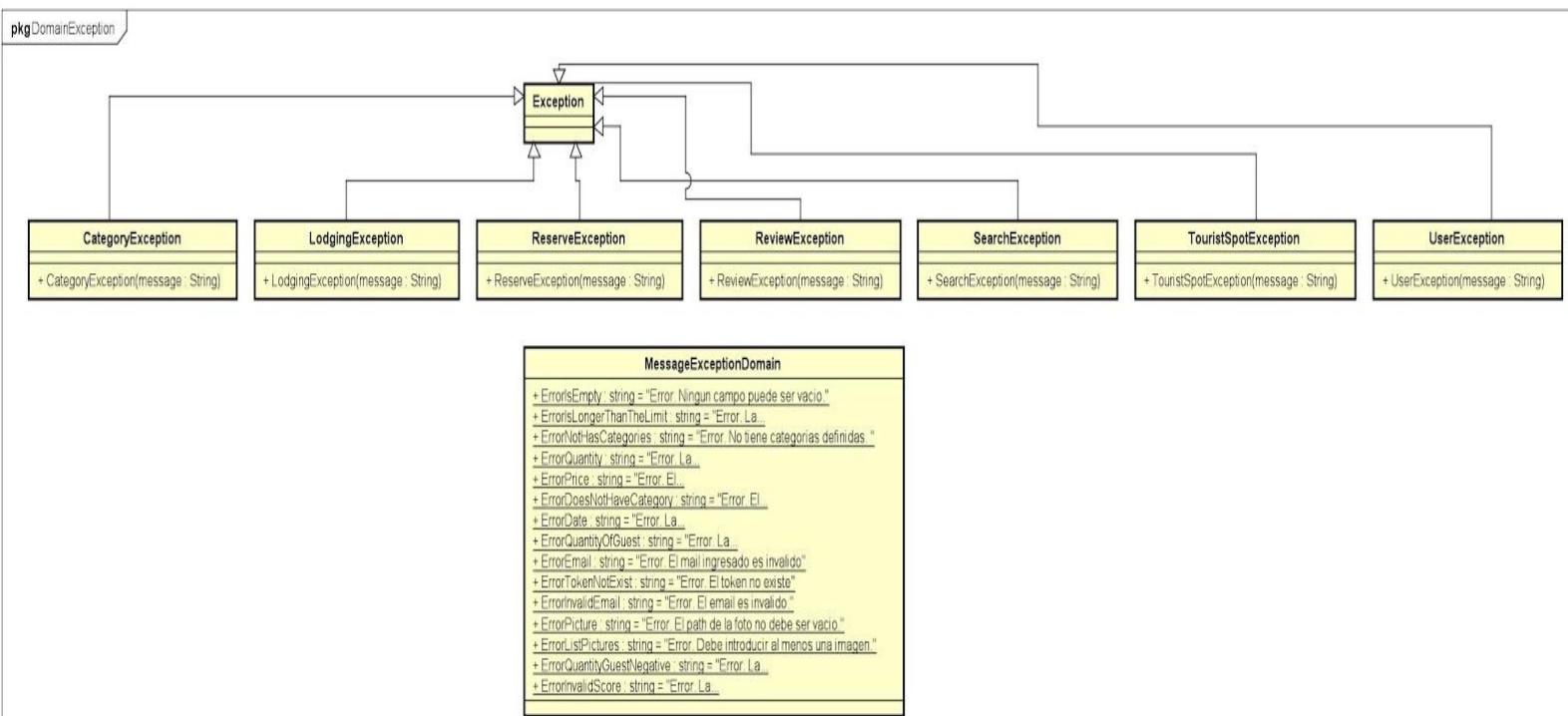
Otro aspecto a destacar, es que decimos realizar borrado físico en lugar de borrado lógico. Tomamos esta decisión, debido a que consideramos que es muy costoso mantener todos esos datos eliminados en la base de datos, que ya no tiene ningún sentido que permanezcan almacenados (no nos interesa mantener un historial de los registros de nuestra base de datos).

Como utilizamos este tipo de borrado, decidimos utilizar la opción de borrado en cascada, para aquellas entidades que solo “viven” mientras no se haya eliminado el registro de la tabla primaria que lo referencia. A partir del momento en que se borra la fila de la tabla primaria, la fila referenciada de la tabla secundaria deja de tener sentido, y por tanto, la eliminamos utilizando un borrado en cascada.

Excepciones en el dominio

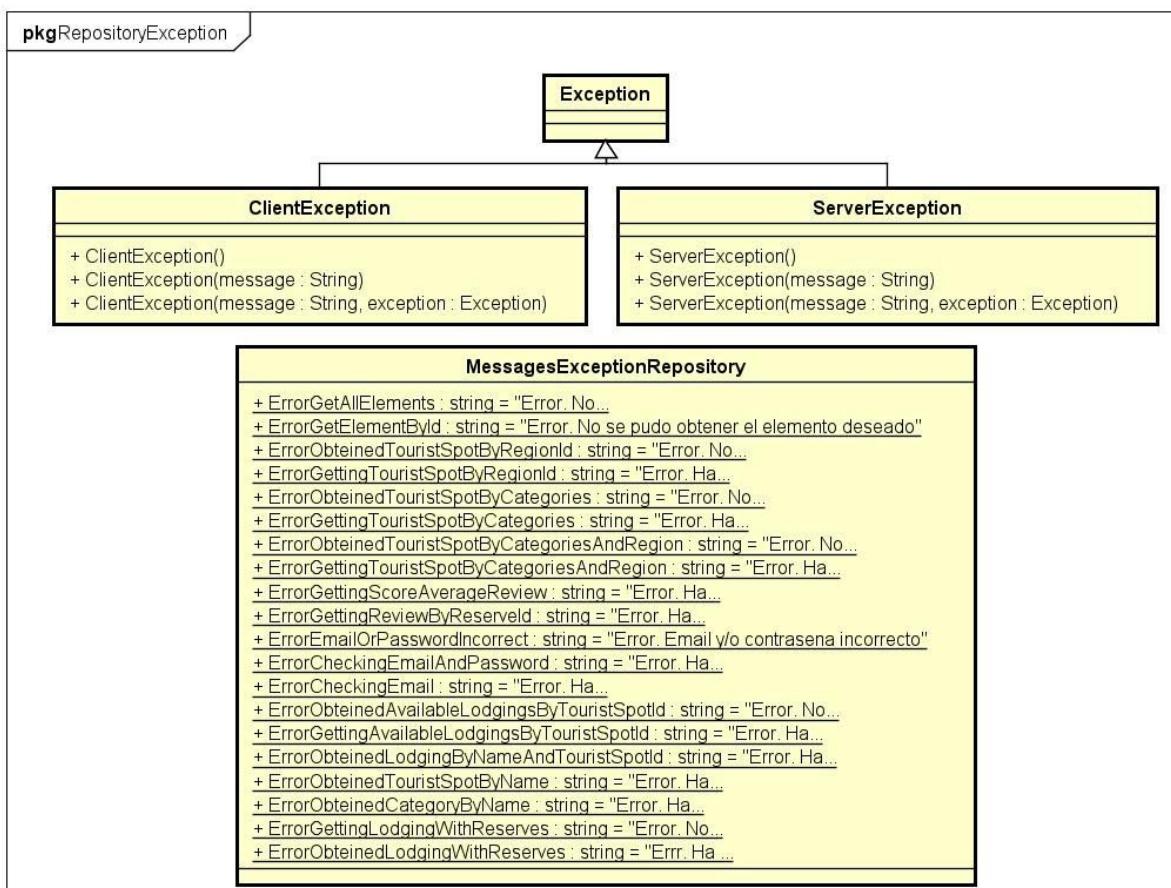
En referencia a las excepciones, también contamos con un paquete llamado *DomainException*, que se encarga de manejar todas las excepciones relacionadas al dominio del problema. En particular a las excepciones existentes en la versión anterior del sistema, en las cuales se contaba con una clase por cada una de las entidades del dominio, las cuales eran: *CategoryException*, *LodgingException*, *ReserveException*, *TouristSpotException*, *UserException*, se agregaron las clases de excepciones *ReviewException* y *SearchException*, donde cada una de ellas se encarga de manejar las excepciones relacionadas a una entidad del dominio como su nombre lo indica.

Por último, tenemos la clase *MessageExceptionDomain*, cuya finalidad es la de contener los mensajes de error que se muestran al lanzar una excepción (de forma similar a como se describió en la sección en la que se hace referencia a esta parte del anexo).



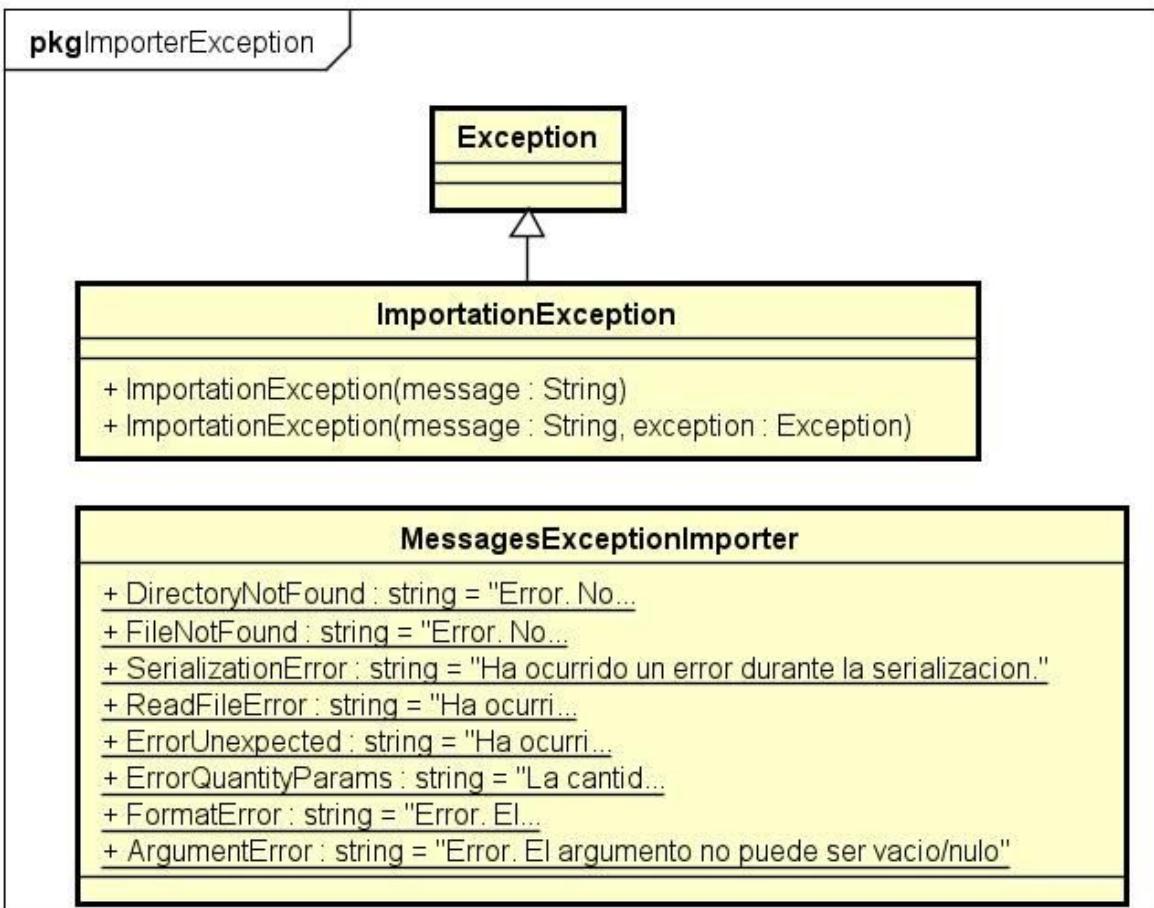
Excepciones en el acceso a datos

De manera similar a las excepciones en el dominio, sucede de igual forma en el acceso a datos, dónde tenemos el paquete *RepositoryException* que se encarga de manejar las excepciones relacionadas a la base de datos, tanto aquellas causadas por errores del cliente (*ClientException*) como las causadas por errores internos del servidor (*ServerException*). Luego tenemos una clase que contiene los mensajes de error que se mostrarán al lanzar las excepciones (*MessagesExceptionRepository*). Esto es de esta forma, debido a lo descrito anteriormente en la sección Manejo de excepciones donde se introducen las excepciones en las reglas de negocio.



Excepciones en la importación

Finalmente, tenemos el paquete *ImportationException* que se encarga de manejar las excepciones relacionadas a la importación de hospedajes. Luego tenemos una clase que contiene los mensajes de error que se mostrarán al lanzar las excepciones (*MessagesExceptionImporter*). Esto es de esta forma, debido a lo descrito anteriormente en la sección Manejo de excepciones donde se introducen las excepciones en las reglas de negocio.



Cobertura de las pruebas

Utilizando el explorador de pruebas, podemos apreciar que se realizaron un total de 369 tests, obteniendo un porcentaje de pruebas exitosas igual al 100 %.

Esos 369 *tests* se dividen de la siguiente forma:

- 158 pertenecen al paquete *BusinessLogicTest*.
- 81 pertenecen al paquete *DataAccessTest*.
- 20 pertenecen al paquete *ImportationTest*.
- 110 pertenecen al paquete *WebApiTest*.

Prueba	Duración	Rasgos	Mensaje de error
▷ ✓ BusinessLogicTest (158)	1,1 s		
▷ ✓ DataAccessTest (81)	2,4 s		
▷ ✓ ImportationTest (20)	443 ms		
▷ ✓ WebApiTest (110)	664 ms		

La aplicación fue desarrollada siguiendo *TDD* (*Test Driven Development*). Esta práctica consiste en escribir antes la prueba que la funcionalidad. Para esto, en primer lugar, se escribe una prueba y se verifica que la nueva prueba falle. Luego, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito.

Utilizando esta práctica, se alcanza un porcentaje de cobertura de pruebas unitarias muy bueno. Si pasamos a ejecutar el analizador de cobertura de código, obtenemos finalmente que el porcentaje total de cobertura para toda la solución es igual al **97,94 %**.

Jerarquía	No cubiertos (bloques)	No cubiertos (% de bloques)	Cubiertos (bloques)	Cubiertos (% de bloques)
user_LAPTOP-7B6D15A6 202...	307	2,06 %	14595	97,94 %
▷ businesslogic.dll	0	0,00 %	573	100,00 %
▷ businesslogicexception.dll	0	0,00 %	17	100,00 %
▷ businesslogictest.dll	122	1,81 %	6600	98,19 %
▷ dataaccess.dll	37	6,60 %	524	93,40 %
▷ dataacesstest.dll	43	3,60 %	1150	96,40 %
▷ domain.dll	18	2,95 %	592	97,05 %
▷ domainexception.dll	0	0,00 %	13	100,00 %
▷ filters.dll	0	0,00 %	23	100,00 %
▷ importation.dll	1	0,88 %	113	99,12 %
▷ importationtest.dll	16	6,69 %	223	93,31 %
▷ importerexception.dll	0	0,00 %	5	100,00 %
▷ importerjson.dll	2	4,55 %	42	95,45 %
▷ importerxml.dll	2	4,35 %	44	95,65 %
▷ model.dll	64	6,29 %	953	93,71 %
▷ repositoryexception.dll	0	0,00 %	13	100,00 %
▷ webapi.dll	0	0,00 %	472	100,00 %
▷ webapitest.dll	2	0,06 %	3238	99,94 %

Ahora realizaremos un análisis de los resultados de cobertura obtenidos para cada paquete.

BusinessLogic

Pasando a analizar el paquete *BusinessLogic*, podemos notar que el porcentaje de cobertura de líneas de código del mismo es igual al **100 %**, esto quiere decir, que cada línea de las clases pertenecientes a *BusinessLogic* fueron testeadas.

user_LAPTOP-7B6D15A6 202...	307	2,06 %	14595	97,94 %
▷ businesslogic.dll	0	0,00 %	573	100,00 %
▷ BusinessLogic	0	0,00 %	573	100,00 %
▷ CategoryManage...	0	0,00 %	33	100,00 %
▷ LodgingManagem...	0	0,00 %	122	100,00 %
▷ LodgingManagem...	0	0,00 %	61	100,00 %
▷ LodgingManagem...	0	0,00 %	2	100,00 %
▷ RegionManagement	0	0,00 %	19	100,00 %
▷ ReserveManageme...	0	0,00 %	74	100,00 %
▷ ReserveManageme...	0	0,00 %	4	100,00 %
▷ ReserveManageme...	0	0,00 %	4	100,00 %
▷ ReviewManagement	0	0,00 %	48	100,00 %
▷ TouristSpotManag...	0	0,00 %	95	100,00 %
▷ UserManagement	0	0,00 %	111	100,00 %

BusinessLogicException

En este paquete, podemos notar que el porcentaje de cobertura es igual al **100 %**, es decir, todas las excepciones de las reglas de negocio fueron testeadas por completo y de forma exitosa.

▲	businesslogicexception.dll	0	0,00 %	17	100,00 %
◀	BusinessLogicException	0	0,00 %	17	100,00 %
▷	ClientBusinessLogi...	0	0,00 %	6	100,00 %
▷	DomainBusinessLo...	0	0,00 %	4	100,00 %
▷	MessageException...	0	0,00 %	1	100,00 %
▷	ServerBusinessLogi...	0	0,00 %	6	100,00 %

BusinessLogicTest

Podemos apreciar que el porcentaje de cobertura de *BusinessLogicTest* es igual a **98,19 %**. Este análisis lo hacemos básicamente para saber si tenemos ciertas líneas en los tests que no son necesarias dejarlas porque no se están ejecutando, en este caso, al ingresar a revisar qué es lo que está sucediendo, notamos que no es que existan líneas que no se están ejecutando, sino que considera a las llaves “{” y “}” como parte del código y no lo está contabilizando cuando analiza la cobertura.

▲	businesslogictest.dll	122	1,81 %	6600	98,19 %
◀	BusinessLogicTest	122	1,81 %	6600	98,19 %
▷	CategoryTest	5	2,02 %	243	97,98 %
▷	LodgingTest	27	2,23 %	1182	97,77 %
▷	LodgingTestForIm...	13	1,45 %	885	98,55 %
▷	RegionTest	4	2,80 %	139	97,20 %
▷	ReserveTest	15	1,65 %	895	98,35 %
▷	ReviewTest	11	1,53 %	709	98,47 %
▷	TouristSpotTest	23	1,65 %	1368	98,35 %
▷	UserTest	24	2,00 %	1179	98,00 %

DataAccess

Analizando el porcentaje de cobertura del paquete *DataAccess*, se obtiene que el mismo es igual a **93,40 %**. Analizando los métodos particulares en que la cobertura no llega al 100 %, notamos que esto se debe a que dentro de estos métodos se está capturando una *Exception* con el fin de que si ocurre un error interno en la base de datos, como puede ser una desconexión, el sistema pueda capturar esa excepción e informar al usuario que es lo que ha sucedido. Lo que sucede es que este tipo de excepciones, al surgir por un error inesperado de la base de datos, no es posible probarlas desde los métodos de *tests* (a no ser que se pruebe con mocking, pero no es adecuado para las pruebas de acceso a datos), y eso implica que el porcentaje de cobertura descienda.

Cabe destacar que aquí, se utilizó el tag [ExcludeFromCodeCoverage] con el fin de que no se tomen en cuenta las migraciones a la hora de realizar el análisis de cobertura de código.

▲	dataaccess.dll	37	6,60 %	524	93,40 %
◀	DataAccess	37	6,60 %	524	93,40 %
▷	BaseRepository<T>	6	8,96 %	61	91,04 %
▷	CategoryRepository	2	9,52 %	19	90,48 %
▷	ContextFactory	0	0,00 %	9	100,00 %
▷	ContextObl	11	6,59 %	156	93,41 %
▷	LodgingRepository	6	7,14 %	78	92,86 %
▷	LodgingRepository...	0	0,00 %	2	100,00 %
▷	LodgingRepository...	0	0,00 %	10	100,00 %
▷	ReviewRepository	4	8,51 %	43	91,49 %
▷	TouristSpotReposit...	4	7,14 %	52	92,86 %
▷	TouristSpotReposit...	0	0,00 %	8	100,00 %
▷	UserRepository	4	7,41 %	50	92,59 %
▷	UserSessionReposi...	0	0,00 %	36	100,00 %

DataAccessTest

Analizando la cobertura del paquete *DataAccessTest* notamos que la misma es igual al **96,40 %**. La razón por la cual no se cubre un 100 % se debe a la misma razón que la del paquete *BusinessLogicTest* (está contabilizando como que las “{” y “}” no se están ejecutando).

▶  dataaccesstest.dll	43	3,60 %	1150	96,40 %
▶  DataAccessTest	43	3,60 %	1150	96,40 %
▶  CategoryDATest	7	5,07 %	131	94,93 %
▶  LodgingDATest	9	5,88 %	144	94,12 %
▶  RegionDATest	4	3,70 %	104	96,30 %
▶  ReportDATest	3	1,91 %	154	98,09 %
▶  ReserveDATest	4	3,01 %	129	96,99 %
▶  ReviewDATest	0	0,00 %	117	100,00 %
▶  TouristSpotDATest	9	4,74 %	181	95,26 %
▶  UserDATest	4	4,30 %	89	95,70 %
▶  UserSessionDATest	3	2,88 %	101	97,12 %

Domain

Analizando la cobertura del paquete *Domain* podemos observar que la misma es igual al **97,05 %**. La razón por la cual no se llega al 100 % de cobertura, se basa básicamente es que no fueron probados, algunas secciones de métodos equals, específicamente en aquellos en que se recibe un objeto *null* por parámetro, luego restan probar algunos métodos get y set principalmente referidos a las clases *CategoryTouristSpot* y *LodgingPicture*, que son clases para representar las relaciones entre *CategoryTouristSpot*, y *Lodging* y *Picture* respectivamente, entonces los atributos de id, no son muy utilizados en las pruebas a nivel de lógica de negocios o dominio, debido a que el propósito que tienen dichos atributos es referido al nivel de base de datos.

▶  domain.dll	18	2,95 %	592	97,05 %
▶  Domain	18	2,95 %	592	97,05 %
▶  Category	1	5,56 %	17	94,44 %
▶  CategoryTouristSpot	2	20,00 %	8	80,00 %
▶  Lodging	1	0,74 %	134	99,26 %
▶  Lodging.<>c_Dis...	1	6,25 %	15	93,75 %
▶  LodgingPicture	3	37,50 %	5	62,50 %
▶  Picture	2	33,33 %	4	66,67 %
▶  Region	2	6,90 %	27	93,10 %
▶  Reserve	2	1,24 %	159	98,76 %
▶  Review	2	2,90 %	67	97,10 %
▶  TouristSpot	0	0,00 %	77	100,00 %
▶  TouristSpot.<>c	0	0,00 %	2	100,00 %
▶  User	1	1,49 %	66	98,51 %
▶  UserSession	1	8,33 %	11	91,67 %

DomainException

Analizando la cobertura de *DomainException*, se observa que la misma es igual a **100 %**, es decir, que todas las excepciones relativas al dominio fueron testeadas.

domainexception.dll	0	0,00 %	13	100,00 %
{ } DomainException	0	0,00 %	13	100,00 %
▷ 🏰 LodgingException	0	0,00 %	2	100,00 %
▷ 🏰 MessageException...	0	0,00 %	1	100,00 %
▷ 🏰 ReserveException	0	0,00 %	2	100,00 %
▷ 🏰 ReviewException	0	0,00 %	2	100,00 %
▷ 🏰 SearchException	0	0,00 %	2	100,00 %
▷ 🏰 TouristSpotExcepti...	0	0,00 %	2	100,00 %
▷ 🏰 UserException	0	0,00 %	2	100,00 %

Filters

Se puede apreciar que la cobertura del paquete *Filters* es igual al **100 %**, concluyendo entonces, que el *AuthorizationFilter* (único filtro presente), ha sido testeado por completo.

filters.dll	0	0,00 %	23	100,00 %
{ } Filters	0	0,00 %	23	100,00 %
▷ 🏰 AuthorizationFilter	0	0,00 %	23	100,00 %

Models

Analizando la cobertura del paquete *Models*, se puede observar que la misma es igual a **93.71 %**. Básicamente este porcentaje se debe a que no todos los métodos *equals* se encuentran probados por completo, faltan testear algunos *gets* y *sets* de algunas *properties* en particular.

model.dll	64	6,29 %	953	93,71 %
{ } Model	2	3,23 %	60	96,77 %
{ } Model.ForRequest	21	6,54 %	300	93,46 %
{ } Model.ForResponse	34	5,83 %	549	94,17 %
{ } Model.ForResponseAn...	7	13,73 %	44	86,27 %

Repository Exception

Se puede apreciar que el porcentaje de cobertura del paquete *RepositoryException* es igual a **100 %**. A partir de esto podemos indicar que todas las excepciones que fueron creadas para ser lanzadas cuando ocurre un error en la base de datos, están probadas por completo.

repositoryexception.dll	0	0,00 %	13	100,00 %
{ } RepositoryException	0	0,00 %	13	100,00 %
▷ 🏰 ClientException	0	0,00 %	6	100,00 %
▷ 🏰 MessagesExceptionRe...	0	0,00 %	1	100,00 %
▷ 🏰 ServerException	0	0,00 %	6	100,00 %

WebApi

Analizando la cobertura de *WebApi* podemos observar que la misma asciende al **100 %**, es decir, que todos los controllers de la *API* fueron probados por completo. Cabe aclarar que en este paquete se utilizó el tag `[ExcludeFromCodeCoverage]` con el fin de que no se tomen en cuenta las clases *Program* y *Startup* a la hora de realizar el análisis de cobertura de código.

webapi.dll	0	0,00 %	472	100,00 %
WebApiController	0	0,00 %	472	100,00 %
CategoryController	0	0,00 %	26	100,00 %
ImporterController	0	0,00 %	63	100,00 %
ImporterController...	0	0,00 %	2	100,00 %
LodgingController	0	0,00 %	75	100,00 %
RegionController	0	0,00 %	26	100,00 %
ReportController	0	0,00 %	15	100,00 %
ReserveController	0	0,00 %	51	100,00 %
ReviewController	0	0,00 %	33	100,00 %
SearchOfLodgingC...	0	0,00 %	29	100,00 %
TouristSpotControl...	0	0,00 %	59	100,00 %
UserController	0	0,00 %	93	100,00 %

WebApiTest

Analizando la cobertura del paquete *WebApiTest* podemos ver que la misma es igual al **99.94 %**, esto nos indica que prácticamente todas las líneas de los *tests* de este paquete están siendo ejecutadas, y que ninguna es innecesaria.

webapitest.dll	2	0,06 %	3238	99,94 %
WebApiTest	2	0,06 %	3238	99,94 %
AuthorizationFilter...	0	0,00 %	80	100,00 %
CategoryController...	0	0,00 %	152	100,00 %
ImporterController...	2	1,10 %	179	98,90 %
LodgingController...	0	0,00 %	580	100,00 %
RegionControllerT...	0	0,00 %	126	100,00 %
ReportControllerTe...	0	0,00 %	139	100,00 %
ReserveControllerT...	0	0,00 %	423	100,00 %
ReviewControllerT...	0	0,00 %	270	100,00 %
SearchOfLodgingC...	0	0,00 %	220	100,00 %
TouristSpotControl...	0	0,00 %	482	100,00 %
UserControllerTest	0	0,00 %	587	100,00 %

Importation

Analizando la cobertura de *Importation* podemos observar que la misma asciende al **99.12 %**, es decir, que prácticamente todas las líneas de código de este paquete fueron probadas, excepto algunos casos del método equals como se muestra a continuación.

importation.dll	1	0,88 %	113	99,12 %
Importation	1	0,88 %	113	99,12 %
Parameter	1	5,00 %	19	95,00 %
Equals(object)	1	8,33 %	11	91,67 %
Parameter()	0	0,00 %	2	100,00 %
get_Name()	0	0,00 %	1	100,00 %
get_Type()	0	0,00 %	1	100,00 %
get_Value()	0	0,00 %	1	100,00 %
set_Name(string)	0	0,00 %	1	100,00 %
set_Type(string)	0	0,00 %	1	100,00 %
set_Value(string)	0	0,00 %	1	100,00 %

ImportationTest

Analizando la cobertura del paquete *ImportationTest* notamos que la misma es igual al **93.31 %**. La razón por la cual no se cubre un 100 % se debe a la misma razón que la del paquete *BusinessLogicTest* (está contabilizando como que las “{“ y “}” no se están ejecutando).

◀	importationtest.dll	16	6,69 %	223	93,31 %
◀	ImportationTest	16	6,69 %	223	93,31 %
▷	ReflectionLogicTest	16	6,69 %	223	93,31 %

ImporterException

Se puede apreciar que el porcentaje de cobertura del paquete *ImporterException* es igual a **100 %**. A partir de esto podemos indicar que todas las excepciones que fueron creadas para ser lanzadas cuando ocurre un error en la base de datos, están probadas por completo.

◀	importerexception.dll	0	0,00 %	5	100,00 %
◀	ImporterException	0	0,00 %	5	100,00 %
▷	ImportationExcept...	0	0,00 %	4	100,00 %
▷	MessagesExceptio...	0	0,00 %	1	100,00 %

ImporterJson

Se aprecia que el porcentaje de cobertura del paquete *ImporterJson* asciende a **95.45 %**, la razón por la cual no se llega al 100 % se debe a que el método de importación posee una *Exception* genérica que fue colocada luego de capturar excepciones particulares por si ocurre algún error inesperado. Esta excepción no pudo ser probada por medio de los tests, pero decidimos dejarla por si llegase a ocurrir alguna excepción que no estamos contemplando con el resto de las excepciones particulares.

◀	importerjson.dll	2	4,55 %	42	95,45 %
◀	ImporterJson	2	4,55 %	42	95,45 %
▷	JsonImporter	2	4,55 %	42	95,45 %

ImporterXml

Se aprecia que el porcentaje de cobertura del paquete *ImporterXml* asciende a **95.65 %**, la razón por la cual no se llega al 100 % es idéntica al caso del paquete ImporterJson.

◀	importerxml.dll	2	4,35 %	44	95,65 %
◀	ImporterXml	2	4,35 %	44	95,65 %
▷	XmllImporter	2	4,35 %	44	95,65 %

API Rest

En particular primero debemos decir que es una API. Siendo así qué es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones. Siendo así que una API permite que sus servicios se comuniquen con otros sin necesidad de saber cómo están implementados, otorgando flexibilidad, simplificación en el diseño, entre otras características.

Pero por otro lado, encontramos a REST. Siendo que es un estilo de arquitectura para la creación de sistemas distribuidos basados en hipermédia. REST es independiente de cualquier protocolo subyacente y no está necesariamente unido a HTTP. Sin embargo, en las implementaciones más comunes de REST se usa HTTP como protocolo de aplicación, y esta guía se centra en el diseño de API de REST para HTTP, tal como es el caso de nuestro sistema. REST se compone de una lista de principios que se deben cumplir en el diseño de la arquitectura de una API.

Principios REST

Pasaremos a analizar y verificar que nuestra API cumple con cada uno de los principios REST.

Interfaz uniforme

- Nuestra API está diseñada en torno a recursos, que son cualquier tipo de objeto, dato o servicio al que puede acceder el cliente. Por ejemplo, tenemos el recurso User (Id, Nombre, Apellido, Mail, Nombre de usuario, contraseña).
- En nuestra API, un recurso tiene un identificador, que es un URI que identifica de forma única ese recurso.
- Los clientes interactúan con un servicio mediante el intercambio de representaciones de recursos. El servidor envía los datos vía JSON pero el mecanismo de almacenamiento interior (una base de datos en nuestro caso) para el cliente es transparente.
- La representación del recurso que le llega al cliente, es la suficiente para que el mismo pueda crearlo, modificarlo y borrarlo, esto suponiendo que para estas acciones el mismo tenga permisos. Esto debido a que incluye el uso de verbos HTTP, como GET, POST, PUT y DELETE.
- Se utilizan mensajes descriptivos por medio de las características del protocolo HTTP, las cuales son verbos y códigos de estado.
- Para obtener una API jerárquica y con ciertas reglas, procuramos el uso de nombres en plural.

Peticiones sin estado

Las API REST usan un modelo de solicitud sin estado. Las solicitudes HTTP deben ser independientes y pueden producirse en cualquier orden, por lo que no es factible conservar la información de estado transitoria entre solicitudes, es por esto que decimos que todas las consultas tienen la información necesaria para entenderla y no se pueden utilizar datos almacenados. El estado no guardará información de las últimas consultas y tratará a cada una como una nueva.

Por ejemplo tenemos que:

```
GET api/users/12345  
DELETE api/users/12345
```

En la segunda petición hemos tenido que indicar el identificador del recurso que queremos borrar.

El servidor no guarda los datos de la consulta previa que tenía el cliente en particular. Una petición del tipo DELETE api/users debe dar error, ya que falta el identificador y el servidor almacena los datos.

Cacheable

Cuando se manda una request se puede establecer que se cachee o no, en caso de que se cachee, se le da la posibilidad al cliente de usar la respuesta después.

En nuestra API decidimos no cachear las respuestas del servidor, sabiendo de que no será muy alto el número de solicitudes hacia un determinado recurso, y por lo tanto, no tendremos problemas de rendimiento en ese sentido.

Separación de cliente y servidor

Esto se refiere a que para que REST exista deben haber dos actores, un cliente que consuma y un servidor que almacene o genere información.

En ese sentido, nuestra API sigue los siguientes criterios:

- El cliente y servidor están separados, su unión es mediante la interfaz uniforme.
- Mientras la interfaz no cambie, podremos cambiar el cliente o el servidor sin problemas.

Sistema dividido en capas

Capas distribuidas jerárquicamente en la que se restringe el comportamiento de forma que disminuya la visibilidad directa entre componentes. Además, permite que se dividan las capas en distintos servidores.

En ese sentido, nuestra API sigue los siguientes criterios:

- Un cliente no sabe inicialmente (ni debería interesarle) si está conectada directamente con el servidor o si hay componentes intermedios que incrementan la seguridad o distribuyen la carga entre los servidores de la aplicación.
- El uso de capas intermedias, en un futuro servirá para aumentar la escalabilidad o para implementar nuevas políticas de seguridad.

Mecanismos de autenticación de request

Lo que sucede es que dentro de nuestro sistema encontramos, hay diferentes acciones las cuales únicamente pueden ser realizadas por los administradores. Estas son por ejemplo: creación de una categoría, creación de un punto turístico, creación de un hospedaje, modificación de un hospedaje, eliminación de un hospedaje, actualización de una reserva, entre otras. Pero para poder realizar estas consultas debe haber un mecanismo de autenticación para que no cualquier usuario de la API pueda hacer request a los diferentes endpoints, pudiendo así hacer lo que él deseé.

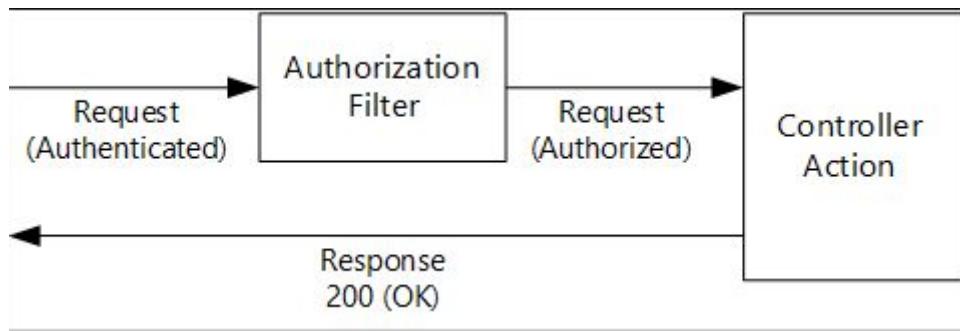
Es por eso que encontramos que uno de los mecanismos de autenticación de los request los cuales son requeridos y realizados únicamente por administradores del sistema, utilizan la implementación de un filtro de autorización. Siendo así que la autenticación y autorización son las bases de la seguridad en aplicaciones. La autenticación establece la identidad de un usuario validando las credenciales, mientras que la autorización determina si un usuario está autorizado para realizar una acción solicitada. Siendo que sí nuestra API cumple con dichas características, se trata de una API protegida, es decir que autentica solicitudes y autoriza el acceso al recurso solicitada en función de la identidad establecida.

Siendo así que los filtros permiten que se ejecute el código antes o después de determinadas fases de la canalización del procesamiento de la solicitud. De tal forma que para el desarrollo de nuestra API se utilizó un filtro de autorización qué tal lo dice la palabra es de autorización. Se ejecutan en primer lugar y sirven para averiguar si el usuario está autorizado para realizar la solicitud. Los filtros de autorización pueden cortocircuitar la canalización si una solicitud no está autorizada, es decir, permiten evitar llegar al controller en caso de no cumplir con las políticas de seguridad, que es lo que se busca cuando un usuario no está autorizado a realizar alguna de las acciones mencionadas anteriormente.

Las operaciones que nos permiten autorizar o denegar una solicitud están comprendidas en la interfaz `IAuthorizationFilter`. Para implementar el filtro, entonces, creamos una clase que implemente dicha interfaz y el método `OnAuthorization` (`AuthorizationFilterContext context`) en donde se implementa la lógica de autorización. Dentro de esta clase hacemos uso de una clase llamada `UserManagement` que se encarga de manejar la lógica relativa a las sesiones de los usuarios.

Dentro del método mencionado, lo que hacemos es verificar que en el header de la solicitud esté contenido un identificador (token, el cual también es llamado token en los headers), y que el mismo se encuentre en la tabla donde se encuentran los usuarios que se encuentran logueados actualmente.

Lo que sucede es que en los casos que no se cumple con la política de seguridad se le asigna un `ContentResult` al resultado de la solicitud, y automáticamente se responde la solicitud no llegando de esta manera al método del Controller.



Descripción general de códigos de error

De tal manera que primero debemos introducirnos acerca de que es un código de estado HTTP. De forma tal que un código de estado HTTP es un mensaje que devuelve el servidor cada vez que el navegador realiza una petición al servidor. Si el servidor es capaz de devolver el contenido que solicita el navegador y no existe ningún error, estos códigos HTTP no son visibles para el usuario. En cambio, si algo va mal, el servidor devuelve un código de estado HTTP que indica que algo no salió como esperaba con un mensaje de error incluido en la implementación de nuestra Web Api, de forma tal que es entendible para él usuario de aplicación entender qué fue lo que ha salido mal.

De tal manera que en función del código de estado que devuelve el servidor, el usuario de la Web Api, además del mensaje de error, observando el código de estado HTTP se podrá hacer una idea del tipo de error que se ha producido. De forma que es un buen principio utilizarlos cuando se tiene una API, ya que es lo que da noción de qué situación el usuario se encuentra cuando realiza peticiones a la misma.

Es por eso que encontramos que los códigos de estado los podemos dividir en cinco categorías diferentes, de manera que visualizando únicamente el permitir dígito del código de estado le permite identificar al usuario de la API, al tipo de respuesta que la misma le está dando. Siendo así que las diferentes categorías son:

- Código de estado 1xx: Se trata de respuestas de carácter informativo dónde el servidor le notifica al cliente que la petición actual continua.
- Código de estado 2xx: Se trata de respuestas que son satisfactorias. Indica que la acción solicitada por el cliente ha sido recibida, entendida, aceptada y procesada correctamente.
- Código de estado 3xx: Se trata de códigos los cuales hacen referencia a que la solicitud ha sido recibida. Sin embargo, para asegurar un procesamiento exitoso es necesario que el cliente tome una acción adicional como, por ejemplo, una redirección.
- Código de estado 4xx: Estos códigos de estado hacen referencia a errores a que se ha presentado un error de cliente. Esto quiere decir que se ha recibido la solicitud, pero esta no se puede llevar a cabo.
- Código de estado 5xx: Estos códigos HTTP también muestran errores, pero por el lado del servidor.

De tal forma que existen más de 70 códigos de estado HTTP, esto da la idea de que todos los códigos existentes se deberían utilizar lo cual no es cierto, ya que la mayoría de los desarrolladores no conocen a todos, de tal manera que si se utilizan códigos de estado poco conocidos, se está guiando a qué se consuma nuestra API, pero luego se redirigen a otra pagina a buscar cual es el significado del código de estado qué les está llegando.

De tal manera que siguiendo con las buenas prácticas en el desarrollo de la API, decidimos utilizar no mucha cantidad de códigos de estado, y tampoco códigos pocos conocidos.

Entonces, en resumen, todo lo anterior nos lleva a preguntar, ¿cuántos códigos de estado debe usar en la API?

Y la respuesta que encontramos a esto fue lo que se implementó de manera satisfactoria en nuestra API. Se podrían implementar con los códigos más utilizados (200, 400 y 500), y a partir de allí extenderlos dependiendo de las necesidades de la aplicación. Pero en particular la API no debería tener más de ocho códigos de error. Esto debido a que una buena práctica, es utilizar pocos y conocidos códigos de estado HTTP.

Siendo así que los códigos de estados HTTP qué se utilizan y devuelven en nuestra api son:

Código de estado	Mensaje de estado	Descripción
200	Ok	La solicitud ha tenido éxito.
201	Creado	La solicitud ha tenido éxito y se ha creado un nuevo recurso como resultado de ello. Ésta es típicamente la respuesta enviada después de una petición PUT
204	Sin contenido	La petición se ha completado con éxito pero su respuesta no tiene ningún contenido, aunque los encabezados pueden ser útiles
400	solicitud incorrecta	No se puede procesar la solicitud porque tiene un formato incorrecto o es incorrecta.
401	Sin autorizar	La información de autenticación necesaria falta o no es válida para el recurso.

403	prohibido	Se denegó el acceso al recurso solicitado. Es posible que el usuario no tenga permisos suficientes
404	No encontrado	El recurso solicitado no existe.
500	error interno del servidor	Se produjo un error interno del servidor al procesar la solicitud.

Ahora, pasaremos a explicar en qué casos usamos cada código de error y con qué finalidad.

200 OK: Este código generalmente lo usamos en los casos en que damos respuesta a solicitudes HTTP GET exitosas, para indicarle al cliente que la solicitud que realizó ha tenido éxito.

201 CREATED: Este código lo usamos cuando se realiza una petición HTTP POST o HTTP PUT, y se crea o modifica un nuevo recurso de forma exitosa, en ese sentido, consideramos apropiado devolver este código que indica que un nuevo recurso ha sido creado.

204 NO CONTENT: Cuando se realiza una petición HTTP DELETE, y la petición se ha completado con éxito pero su respuesta no tiene ningún contenido (debido a que el objeto se ha eliminado).

400 BAD REQUEST: Generalmente, este código de error lo utilizamos cuando se realiza una solicitud HTTP POST o HTTP PUT con alguno de los campos requeridos de forma inválida, en este sentido, nos parece apropiado indicar que el servidor no pudo interpretar la solicitud dada una sintaxis inválida.

401 UNAUTHORIZED: Este código de error lo usamos para impedir que un usuario que no se encuentre logueado pueda hacer uso de las funcionalidades que requieren autenticación.

403 FORBIDDEN: Este código de error lo utilizamos cuando verificamos que un usuario esté logueado para que pueda realizar una determinada funcionalidad que requiere autenticación, y se tiene que el token recibido por header no es válido.

404 NOT FOUND: Este código de error lo utilizamos cuando se realiza una petición HTTP GET y no se encuentra el recurso solicitado, así como también en los casos en que se realiza un HTTP POST o HTTP PUT para crear o modificar un recurso que tiene asociado otro recurso, y no es posible encontrar ese recurso.

500 INTERNAL SERVER ERROR: Este código de error es utilizado cuando ocurre un problema interno, como es el caso de un problema en la base de datos.

Descripción de los resources de la API

Para observar de forma resumida las operaciones disponibles sobre cada uno de los recursos, y el resultado que se espera obtener, ir a sección 1 de Anexo.

Región controller

GET	/api/regions
Resource	Regions
Description	Devuelve todas las regiones existentes en el sistema.
Parameters	No recibe parametros.
Responses	A) 200: Ocurre cuando la petición es exitosa y se devuelven todas las regiones existentes. B) 404: Ocurre cuando no existe el recurso solicitado, es decir, no hay regiones existentes. C) 500: Ocurre cuando sucede un problema interno en el servidor.
Headers	No recibe ningun header adicional a los por defecto.

GET	/api/regions/{id}
Resource	Regions
Description	Realiza la búsqueda de una region en el sistema a partir del id ingresado, y en caso de que exista la retorna.
Parameters	Recibe el id de la region a buscar.
Responses	A) 200: Ocurre cuando la petición es exitosa y se devuelve la region que posee el id parametro. B) 404: Ocurre cuando no existe el recurso solicitado, es decir, no existe una region con el id buscado. C) 500: Ocurre cuando sucede un problema interno en el servidor.
Headers	No recibe ningun header adicional a los por defecto.

Category controller

GET	/api/categories/
Resource	Categories
Description	Devuelve todas las categorias existentes en el sistema.
Parameters	No recibe parametros.
Responses	A) 200: Ocurre cuando la petición es exitosa y se devuelven todas las categorias existentes. B) 404: Ocurre cuando no existe el recurso solicitado, es decir, no hay categorias existentes. C) 500: Ocurre cuando sucede un problema interno en el servidor.
Headers	No recibe ningun header adicional a los por defecto.

GET	/api/categories/{id}
Resource	Categories
Description	Realiza la busqueda de una categoria en el sistema a partir del id ingresado, y en caso de que exista la retorna.
Parameters	Recibe el id de la categoria a buscar.
Responses	A) 200: Ocurre cuando la peticion es exitosa y se devuelve la categoria que posee el id parametro. B) 404: Ocurre cuando no existe el recurso solicitado, es decir, no existe una categoria con el id buscado. C) 500: Ocurre cuando sucede un problema interno en el servidor.
Headers	No recibe ningun header adicional a los por defecto.

Tourist Spot controller

GET	/api/touristSpots
Resource	Tourist Spots
Description	Devuelve todos los puntos turisticos existentes en el sistema.
Parameters	No recibe parametros.
Responses	A) 200: Ocurre cuando la peticion es exitosa y se devuelven todos los puntos turisticos B) 404: Ocurre cuando no existe el recurso solicitado, es decir, no hay puntos turisticos existentes. C) 500: Ocurre cuando sucede un problema interno en el servidor.
Headers	No recibe ningun header adicional a los por defecto.

GET	/api/touristSpots/{id}
Resource	Tourist Spots
Description	Realiza la busqueda de un punto turistico en el sistema a partir del id ingresado, y en caso de que exista la retorna.
Parameters	Recibe el id del punto turistico a buscar.
Responses	A) 200: Ocurre cuando la peticion es exitosa y se devuelve el punto turistico que posee el id parametro. B) 404: Ocurre cuando no existe el recurso solicitado, es decir, no existe un punto turistico con el id buscado. C) 500: Ocurre cuando sucede un problema interno en el servidor.
Headers	No recibe ningun header adicional a los por defecto.

POST	/api/touristSpots
Resource	Tourist Spots
Description	Crea y retorna un nuevo punto turistico con la informacion contenida en el body (el cual es obligatorio). Esto se realiza siempre y cuando haya un administrador logueado en el sistema y los campos sean correctos.
Parameters	No recibe parametros.
Body	{ "name": "string", "description": "string", "regionId": "3fa85f64-5717-4562-b3fc-2c963f66afa6", "imagePath": "string", "listOfCategoriesId": ["3fa85f64-5717-4562-b3fc-2c963f66afa6"] }
Responses	A) 201: Ocurre cuando la peticion es exitosa y se crea el punto turistico con la informacion contenida en el body. Una vez creado, el mismo se retorna. B) 400: Ocurre cuando el servidor no puede procesar la peticion debido a algo que es percibido como un error del cliente, y muestra un mensaje indicando el error ocurrido. C) 401: Ocurre cuando se intenta realizar la operacion sin estar logueado. D) 404: Ocurre cuando no existe el recurso solicitado, en este caso, cuando no existe la region o las categorias asociadas. e) 500: Ocurre cuando sucede un problema interno en el servidor.
Headers	Recibe un header adicional a los por defecto con el token de la sesion del administrador que se encuentra logueado y desea realizar la operacion.

GET	/api/touristSpots/byCategoriesAndRegion/
Resource	TouristSpots
Description	Realiza la busqueda de los puntos turisticos en el sistema a partir de los ids de las categorias y region ingresadas y en caso de que existan los retorna.
Parameters	Recibe una lista de ids de las categorias y un id de la region por las cuales se quieren obtener los puntos turisticos que los tienen
Responses	A) 200: Ocurre cuando la peticion es exitosa y se devuelven los puntos turisticos que tienen las categorias y region buscadas. B) 404: Ocurre cuando no existe el recurso solicitado, es decir, no existen puntos turisticos para las categorias o la region pasadas por parametro. C) 500: Ocurre cuando sucede un problema interno en el servidor.
Headers	No recibe ningun header adicional a los por defecto.

Lodging controller

GET	/api/lodgings
Resource	Lodgings
Description	Devuelve todos los hospedajes existentes en el sistema.
Parameters	No recibe parametros.
Responses	A) 200 : Ocurre cuando la petición es exitosa y se devuelven todos los hospedajes. B) 404 : Ocurre cuando no existe el recurso solicitado, es decir, no hay hospedajes existentes. C) 500 : Ocurre cuando sucede un problema interno en el servidor.
Headers	No recibe ningun header adicional a los por defecto.

GET	/api/lodgings/{id}
Resource	Lodgings
Description	Realiza la búsqueda de un hospedaje en el sistema a partir del id ingresado, y en caso de que exista lo retorna.
Parameters	Recibe el id del hospedaje a buscar.
Responses	A) 200 : Ocurre cuando la petición es exitosa y se devuelve el hospedaje que posee el id parametro. B) 404 : Ocurre cuando no existe el recurso solicitado, es decir, no existe un hospedaje con el id buscado. C) 500 : Ocurre cuando sucede un problema interno en el servidor.
Headers	No recibe ningun header adicional a los por defecto.

POST	/api/lodgings
Resource	Lodgings
Description	Crea y retorna un nuevo hospedaje con la informacion contenida en el body (el cual es obligatorio). Esto se realiza siempre y cuando haya un administrador logueado en el sistema y los campos sean correctos.
Parameters	No recibe parametros.
Body	{ "name": "string", "description": "string", "quantityOfStars": 0, "address": "string", "images": ["string"], "pricePerNight": 0, "isAvailable": true, "touristSpotId": "3fa85f64-5717-4562-b3fc-2c963f66afa6" }
Responses	A) 201 : Ocurre cuando la peticion es exitosa y se crea el hospedaje con la informacion contenida en el body. Una vez creado, el mismo se retorna. B) 400 : Ocurre cuando el servidor no puede procesar la peticion debido a algo que es percibido como un error del cliente, y muestra un mensaje indicando el error ocurrido. C) 401 : Ocurre cuando se intenta realizar la operacion sin estar logueado. D) 404 : Ocurre cuando no existe el recurso solicitado, en este caso, cuando no existe el punto turistico asociado E) 500 : Ocurre cuando sucede un problema interno en el servidor.
Headers	Recibe un header adicional a los por defecto con el token de la sesion del administrador que se encuentra logueado y desea realizar la operacion.

Nota: El nombre del header que contiene el token de autenticación se llama: **token**.

DELETE	/api/lodgings/{id}
Resource	Lodgings
Description	Elimina un hospedaje cuyo id coincide con el pasado por parametro. Esto lo realiza siempre y cuando haya un administrador logueado en el sistema.
Parameters	Recibe el id del hospedaje a eliminar.
Responses	A) 204 : Ocurre cuando la peticion es exitosa y se ha eliminado correctamente el hospedaje. No muestra contenido. B) 401 : Ocurre cuando se intenta realizar la operacion sin estar logueado. C) 404 : Ocurre cuando no existe el recurso solicitado, en este caso, cuando no existe el hospedaje a eliminar D) 500 : Ocurre cuando sucede un problema interno en el servidor.
Headers	Recibe un header adicional a los por defecto con el token de la sesion del administrador que se encuentra logueado y desea realizar la operacion.

Nota: El nombre del header que contiene el token de autenticación se llama: **token**.

PUT	/api/lodgings/{id}
Resource	Lodgings
Description	Modifica y retorna el hospedaje cuyo id coincide con el parametro, actualizando sus datos con la informacion contenida en el body (el cual es obligatorio). Esto se realiza siempre y cuando haya un administrador logueado y los campos sean correctos.
Parameters	Recibe el id del hospedaje a actualizar.
Body	{ "name": "string", "description": "string", "quantityOfStars": 0, "address": "string", "images": ["string"], "pricePerNight": 0, "isAvailable": true, "touristSpotId": "3fa85f64-5717-4562-b3fc-2c963f66afa6" }
Responses	<p>A) 201: Ocurre cuando la peticion es exitosa y se actualiza el hospedaje con la informacion contenida en el body. Una vez actualizado, el mismo se retorna.</p> <p>B) 400: Ocurre cuando el servidor no puede procesar la peticion debido a algo que es percibido como un error del cliente, y muestra un mensaje indicando el error ocurrido.</p> <p>C) 401: Ocurre cuando se intenta realizar la operacion sin estar logueado.</p> <p>D) 404: Ocurre cuando no existe el recurso solicitado, en este caso, cuando no existe el hospedaje que se quiere actualizar.</p> <p>E) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	Recibe un header adicional a los por defecto con el token de la sesion del administrador que se encuentra logueado y desea realizar la operacion.

Nota: El nombre del header que contiene el token de autenticación se llama: **token**.

Search of lodgings controller

POST	/api/searchOfLodgings
Resource	Search of lodgings
Description	Crea y retorna el resultado de realizar una nueva busqueda de hoespedajes con la informacion contenida en el body (el cual es obligatorio). Esto se realiza siempre y cuando los campos sean correctos.
Parameters	No recibe parametros.
Body	{ "checkIn": "2020-10-15T02:50:33.028Z", "checkOut": "2020-10-15T02:50:33.028Z", "quantityOfAdult": 0, "quantityOfChilds": 0, "quantityOfBabies": 0, "touristSpotIdSearch": "3fa85f64-5717-4562-b3fc-2c963f66afa6" }
Responses	<p>A) 201: Ocurre cuando la peticion es exitosa y se retornan los hospedajes disponibles de acuerdo a la informacion de la busqueda.</p> <p>B) 400: Ocurre cuando el servidor no puede procesar la peticion debido a algo que es percibido como un error del cliente, y muestra un mensaje indicando el error ocurrido.</p> <p>C) 404: Ocurre cuando no existe el recurso solicitado, en este caso, cuando no existen hospedajes para la busqueda o no existe el punto turistico asociado a la busqueda.</p> <p>E) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	No recibe ningun header adicional a los por defecto.

Reserve controller

POST	/api/reserves
Resource	Reserves
Description	Crea y retorna una nueva reserva con la informacion contenida en el body (el cual es obligatorio). Esto se realiza siempre y cuando los campos sean correctos.
Parameters	No recibe parametros.
Body	{ "name": "string", "lastName": "string", "email": "string", "checkIn": "2020-10-15T02:57:18.772Z", "checkOut": "2020-10-15T02:57:18.772Z", "quantityOfAdult": 0, "quantityOfChild": 0, "quantityOfBaby": 0, "idOfLodgingToReserve": "3fa85f64-5717-4562-b3fc-2c963f66afa6" }
Responses	<p>A) 201: Ocurre cuando la peticion es exitosa y se crea la reserva con la informacion contenida en el body. Una vez creada, la misma se retorna.</p> <p>B) 400: Ocurre cuando el servidor no puede procesar la peticion debido a algo que es percibido como un error del cliente, y muestra un mensaje indicando el error ocurrido.</p> <p>C) 404: Ocurre cuando no existe el recurso solicitado, en este caso, cuando no existe el hospedaje a reservar.</p> <p>D) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	No recibe ningun header adicional a los por defecto.

GET	/api/reserves/{id}
Resource	Reserves
Description	Realiza la busqueda de una reserva en el sistema a partir del id ingresado, y en caso de que exista la retorna.
Parameters	Recibe el id de la reserva a buscar.
Responses	<p>A) 200: Ocurre cuando la peticion es exitosa y se devuelve la reserva que posee el id parametro.</p> <p>B) 404: Ocurre cuando no existe el recurso solicitado, es decir, no existe una reserva con el id buscado.</p> <p>C) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	No recibe ningun header adicional a los por defecto.

PUT	/api/reserves/{idForUpdateReserve}
Resource	Reserves
Description	Modifica y retorna la reserva cuyo id coincide con el parametro, actualizando sus datos con la informacion contenida en el body (el cual es obligatorio). Esto se realiza siempre y cuando haya un administrador logueado y los campos sean correctos.
Parameters	Recibe el id de la reserva a actualizar.
Body	{ "description": "string", "stateOfReserve": 0 }
Responses	<p>A) 201: Ocurre cuando la peticion es exitosa y se actualiza la reserva con la informacion contenida en el body. Una vez actualizada, la misma se retorna.</p> <p>B) 400: Ocurre cuando el servidor no puede procesar la peticion debido a algo que es percibido como un error del cliente, y muestra un mensaje indicando el error ocurrido.</p> <p>C) 401: Ocurre cuando se intenta realizar la operacion sin estar logueado.</p> <p>D) 404: Ocurre cuando no existe el recurso solicitado, en este caso, cuando no existe la reserva que se quiere actualizar.</p> <p>E) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	Recibe un header adicional a los por defecto con el token de la sesion del administrador que se encuentra logueado y desea realizar la operacion.

Nota: El nombre del header que contiene el token de autenticación se llama: **token**.

User controller

GET	/api/users
Resource	Users
Description	Devuelve todos los usuarios existentes en el sistema, siempre y cuando se este logueado.
Parameters	No recibe parametros.
Responses	<p>A) 200: Ocurre cuando la peticion es exitosa y se devuelven todos los usuarios.</p> <p>B) 401: Ocurre cuando se intenta realizar la operacion sin estar logueado.</p> <p>C) 404: Ocurre cuando no existe el recurso solicitado, es decir, no hay usuarios existentes.</p> <p>D) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	Recibe un header adicional a los por defecto con el token de la sesion del administrador que se encuentra logueado y desea realizar la operacion.

Nota: El nombre del header que contiene el token de autenticación se llama: **token**.

GET	/api/users/{id}
Resource	Users
Description	Realiza la busqueda de un usuario en el sistema a partir del id ingresado, y en caso de que exista lo retorna siempre y cuando el solicitante este logueado.
Parameters	Recibe el id del usuario a buscar.
Responses	<p>A) 200: Ocurre cuando la peticion es exitosa y se devuelve el usuario que posee el id parametro.</p> <p>B) 401: Ocurre cuando se intenta realizar la operacion sin estar logueado.</p> <p>C) 404: Ocurre cuando no existe el recurso solicitado, es decir, no existe un usuario con el id buscado.</p> <p>D) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	Recibe un header adicional a los por defecto con el token de la sesion del administrador que se encuentra logueado y desea realizar la operacion.

Nota: El nombre del header que contiene el token de autenticación se llama: **token**.

POST	/api/users
Resource	Users
Description	Crea y retorna una nuevo usuario con la informacion contenida en el body (el cual es obligatorio). Esto se realiza siempre y cuando los campos sean correctos y el solicitante este logueado.
Parameters	No recibe parametros.
Body	{ "name": "string", "lastName": "string", "userName": "string", "password": "string", "mail": "string" }
Responses	<p>A) 201: Ocurre cuando la peticion es exitosa y se crea el usuario con la informacion contenida en el body. Una vez creada, el mismo se retorna.</p> <p>B) 400: Ocurre cuando el servidor no puede procesar la peticion debido a algo que es percibido como un error del cliente, y muestra un mensaje indicando el error ocurrido.</p> <p>C) 401: Ocurre cuando se intenta realizar la operacion sin estar logueado.</p> <p>D) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	Recibe un header adicional a los por defecto con el token de la sesion del administrador que se encuentra logueado y desea realizar la operacion.

Nota: El nombre del header que contiene el token de autenticación se llama: **token**.

DELETE		/api/users/{id}
Resource	Users	
Description	Elimina un usuario cuyo id coincide con el pasado por parametro. Esto lo realiza siempre y cuando haya un administrador logueado en el sistema.	
Parameters	Recibe el id del usuario a eliminar.	
Responses	A) 204: Ocurre cuando la petición es exitosa y se ha eliminado correctamente el usuario. No muestra contenido. B) 401: Ocurre cuando se intenta realizar la operación sin estar logueado. C) 404: Ocurre cuando no existe el recurso solicitado, en este caso, cuando no existe el usuario a eliminar D) 500: Ocurre cuando sucede un problema interno en el servidor.	
Headers	Recibe un header adicional a los por defecto con el token de la sesión del administrador que se encuentra logueado y desea realizar la operación.	

POST		/api/users/login
Resource	Users	
Description	Crea y retorna una nueva sesión de usuario con la información del usuario a loguear contenida en el body (el cual es obligatorio). Esto se realiza siempre y cuando los campos sean correctos.	
Parameters	No recibe parámetros.	
Body	<pre>{ "email": "string", "password": "string" }</pre>	
Responses	A) 201: Ocurre cuando la petición es exitosa y se crea la sesión de usuario con la información contenida en el body. Una vez creada, la misma se retorna. B) 400: Ocurre cuando el servidor no puede procesar la petición debido a algo que es percibido como un error del cliente, y muestra un mensaje indicando el error ocurrido. C) 500: Ocurre cuando sucede un problema interno en el servidor.	
Headers	No recibe ningún header adicional a los por defecto.	

PUT	/api/users/{id}
Resource	Users
Description	Modifica y retorna el usuario cuyo id coincide con el parametro, actualizando sus datos con la informacion contenida en el body (el cual es obligatorio). Esto se realiza siempre y cuando haya un administrador logueado y los campos sean correctos.
Parameters	Recibe el id del usuario a actualizar.
Body	<pre>{ "id": "3fa85f64-5717-4562-b3fc-2c963f66afab", "name": "string", "lastName": "string", "userName": "string", "password": "string", "mail": "string" }</pre>
Responses	<p>A) 201: Ocurre cuando la petición es exitosa y se actualiza el usuario con la información contenida en el body. Una vez actualizado, el mismo se retorna.</p> <p>B) 400: Ocurre cuando el servidor no puede procesar la petición debido a algo que es percibido como un error del cliente, y muestra un mensaje indicando el error ocurrido.</p> <p>C) 401: Ocurre cuando se intenta realizar la operación sin estar logueado.</p> <p>D) 404: Ocurre cuando no existe el recurso solicitado, en este caso, cuando no existe el usuario que se quiere actualizar.</p> <p>E) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	Recibe un header adicional a los por defecto con el token de la sesión del administrador que se encuentra logueado y desea realizar la operación.

Nota: El nombre del header que contiene el token de autenticación se llama: **token**.

DELETE	/api/users/logout
Resource	Users
Description	Elimina la sesión de usuario asociada al usuario que actualmente se encuentra logueado y realiza la petición.
Parameters	No recibe ningún parámetro.
Responses	<p>A) 200: Ocurre cuando la petición es exitosa y se ha eliminado correctamente la sesión de usuario. Se muestra un mensaje indicando que se ha cerrado correctamente la sesión actual.</p> <p>B) 401: Ocurre cuando se intenta realizar la operación sin estar logueado.</p> <p>C) 404: Ocurre cuando no existe el recurso solicitado, en este caso, cuando no existe el usuario a eliminar.</p> <p>D) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	Recibe un header adicional a los por defecto con el token de la sesión del administrador que se encuentra logueado y desea realizar la operación.

Nota: El nombre del header que contiene el token de autenticación se llama: **token**.

Review controller

GET	/api/touristSpots/reviews/{id}
Resource	Reviews
Description	Realiza la busqueda de una review en el sistema a partir del id ingresado, y en caso de que exista la retorna.
Parameters	Recibe el id de la review a buscar.
Responses	<p>A) 200: Ocurre cuando la peticion es exitosa y se devuelve la review que posee el id parametro.</p> <p>B) 404: Ocurre cuando no existe el recurso solicitado, es decir, no existe una review con el id buscado.</p> <p>C) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	No recibe ningun header adicional a los por defecto.

POST	/api/reviews
Resource	Reviews
Description	Crea y retorna una nueva review con la informacion contenida en el body (el cual es obligatorio). Esto se realiza siempre y cuando los campos sean correctos.
Parameters	No recibe parametros.
Body	<pre>{ "idOfReserveAssociated": "3fa85f64-5717-4562-b3fc-2c963f66afa6", "score": 4, "description": "Buena experiencia" }</pre>
Responses	<p>A) 201: Ocurre cuando la peticion es exitosa y se crea la review con la informacion contenida en el body. Una vez creada, la misma se retorna.</p> <p>B) 400: Ocurre cuando el servidor no puede processar la peticion debido a algo que es percibido como un error del cliente, y muestra un mensaje indicando el error ocurrido.</p> <p>C) 404: Ocurre cuando no existe el recurso solicitado, en este caso, cuando no existe la reserva asociada sobre la cual se quiere enviar la review.</p> <p>D) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	No recibe ningun header adicional a los por defecto.

Importer controller

GET	/api/imports/
Resource	Imports
Description	Devuelve todos los importadores existentes en el sistema.
Parameters	No recibe parametros.
Responses	<p>A) 200: Ocurre cuando la petición es exitosa y se devuelven todos los importadores.</p> <p>B) 404: Ocurre cuando no existe el recurso solicitado, es decir, no hay importadores existentes.</p> <p>C) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	Recibe un header adicional a los por defecto con el token de la sesión del administrador que se encuentra logueado y desea realizar la operación.

Nota: El nombre del header que contiene el token de autenticación se llama: **token**.

GET	/api/imports/parametersOfDllSelected
Resource	Imports
Description	Devuelve los parametros necesarios para que el dll pasado por parametro pueda realizar la importacion.
Parameters	Recibe el path del dll que se desea obtener los parametros necesarios.
Responses	<p>A) 200: Ocurre cuando la petición es exitosa y se devuelven la lista de parametros requeridas para el dll enviado.</p> <p>B) 500: Ocurre cuando sucede un problema interno en el servidor.</p>
Headers	Recibe un header adicional a los por defecto con el token de la sesión del administrador que se encuentra logueado y desea realizar la operación.

Nota: El nombre del header que contiene el token de autenticación se llama: **token**.

POST	/api/imports
Resource	Imports
Description	Crea los hospedajes contenidos en el archivo enviado para realizar la importacion. Esto se realiza siempre y cuando los campos sean correctos y el usuario se encuentre logueado como administrador
Parameters	Recibe el path del dll que se desea utilizar para realizar la importacion.
Body	[{ "name": "path", "type": "file", "value": "myDirectory/file.json" }]
Responses	A) 200: Ocurre cuando la peticion es exitosa y todos los hospedajes fueron importados de forma exitosa. B) 400: Ocurre cuando el servidor no puede processar la peticion debido a algo que es percibido como un error del cliente, y muestra un mensaje indicando el error ocurrido, esto ocurre cuando alguno de los hospedajes a importar tiene campos invalidos o la lista de hospedajes a importar es vacia. C) 500: Ocurre cuando sucede un problema interno en el servidor.
Headers	Recibe un header adicional a los por defecto con el token de la sesion del administrador que se encuentra logueado y desea realizar la operación.

Nota: El nombre del header que contiene el token de autenticación se llama: **token**.

Report controller

GET	/api/reports/report/
Resource	Reports
Description	Devuelve un reporte de tipo A con la cantidad de reservas de cada hospedaje tomando en cuenta el rango de fechas pasado por parametro. La generacion del reporte solo se da si el usuario se encuentra logueado como administrador en el sistema.
Parameters	Recibe el rango de fechas para los cuales se desea tomar en cuenta las reservas para la generacion del reporte (check-in y check-out), ademas del identificador del punto turistico de donde queremos obtener los hospedajes para el reporte.
Responses	A) 200: Ocurre cuando la peticion es exitosa y se devuelve el reporte con los hospedajes considerando el rango de fechas y el punto turistico pasado por parametro. B) 404: Ocurre cuando no existe el recurso solicitado, es decir, no existe el punto turistico pasado por parametro. C) 500: Ocurre cuando sucede un problema interno en el servidor.
Headers	Recibe un header adicional a los por defecto con el token de la sesion del administrador que se encuentra logueado y desea realizar la operacion.

Nota: El nombre del header que contiene el token de autenticación se llama: **token**.

Sección 1

Descripción resumida de cada una de las operaciones.

Para realizar las operaciones contenidas en las tablas que poseen un (*) es necesario estar logueado como administrador en el sistema. Si no de otra forma, no podrá realizarse esta operación debido a que saltará un código de estado 401, con un mensaje de error acerca de qué no se está logueado y no tiene autorización para realizar la operación si no se está logueado.

Regions controller

Resource	Get	Post	Put	Delete
/api/regions	Devuelve todas las regiones existentes	-	-	-
/api/regions/{id}	Devuelve la región identificada con el id buscado	-	-	-

Category controller

Resource	Get	Post	Put	Delete
/api/categories	Devuelve todas las categorías existentes	Crea una nueva categoría y la devuelve en caso de crearse (*)	-	-
/api/categories/{id}	Devuelve la categoría identificada con el id buscado	-	-	-

Tourist Spot controller

Resource	Get	Post	Put	Delete
/api/touristSpots	Devuelve todos los puntos turísticos existentes	Crea un nuevo punto turístico y lo devuelve en caso de crearse (*)	-	-
/api/touristSpots/{id}	Devuelve el punto turístico identificado con el id buscado	-	-	-
/api/touristSpots/getByRegion	Devuelve los puntos turísticos que se ubican en la región indicada en el parámetro.	-	-	-
/api/touristSpots/getByCategories	Devuelve los puntos turísticos que tienen las categorías indicadas por parámetro.	-	-	-
/api/touristSpots/getByCategoriesAndRegion	Devuelve los puntos turísticos que tienen la región y las categorías indicadas por parámetro.	-	-	-

Lodging controller

Resource	Get	Post	Put	Delete
/api/lodgings	Devuelve todos los hospedajes existentes.	Crea un nuevo hospedaje y lo retorna en caso de crearse (*)	-	-
/api/lodgings/{id}	Devuelve el hospedaje identificado con el id buscado.	-	Modifica el hospedaje identificado con el id seleccionado y lo retorna (*)	Elimina el hospedaje identificado con el id. (*)

SearchOfLodging controller

Resource	Get	Post	Put	Delete
/api/searchOfLodgings	-	Crea una búsqueda con los datos pasados, obteniendo cómo respuesta todos los hospedajes que cumplan.	-	-

Reserve controller

Resource	Get	Post	Put	Delete
/api/reserves	-	Crea una reserva con los datos pasados y la retorna en caso de crearla	-	-
/api/reserves/{id}	Devuelve la reserva identificada con el id buscado.	-	-	-
/api/reserves/{idForUpdateReserve}	-	-	Actualiza la reserva que tiene el id buscado con la información contenida en el body. (*)	-

User controller

Resource	Get	Post	Put	Delete
/api/users	Devuelve todos los usuarios existentes (*)	Crea un nuevo usuario (*)	-	-
/api/users/{id}	Devuelve el usuario identificado con el id buscado. (*)	-	Modifica el usuario identificado con el id seleccionado (*)	Elimina el usuario identificado con el id. (*)
/api/users/login	-	Inicia sesión el usuario identificado con los datos ingresados.	-	-
/api/users/logout	-	-	-	Cierra la sesión el usuario, ingresando el token de sesión. (*)

Review controller

Resource	Get	Post	Put	Delete
/api/reviews	-	Crea una reseña con los datos pasados, y luego de crearla la retorna	-	-
/api/reviews/{id}	Devuelve la reseña identificada con el id buscado.			

Import controller

Resource	Get	Post	Put	Delete
/api/imports	Devuelve el nombre de todos los importadores disponibles. (*)	Crea hospedajes con el archivo indicado luego de haber seleccionado el importador a utilizar. (*)	-	-
/api/imports/parametersOfDIISelected	Devuelve la lista de parámetros para el importador seleccionado. (*)			

Report controller

Resource	Get	Post	Put	Delete
/api/reports/report	Genera el reporte de tipo A para los parámetros pasados. (*)	-	-	-