Obligatorio 1 - Diseño de aplicaciones 2
Descripción del diseño

Agustín Hernandorena (233361)
Joaquín Lamela (233375)

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

Índice

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.

Para esta primera entrega del sistema, se implementó una API REST que ofrece todas las operaciones requeridas por el cliente. Toda interacción con el repositorio de datos se realiza mediante un API REST, la que ofrece operaciones para resolver todo lo necesario y el back end donde se implementa la lógica de negocio. Es decir, se implementó todo el backend de la aplicación (lógica de negocio), pero no el frontend de la misma (interfaz de usuario), la cual se implementará para la siguiente entrega.

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.

Para llevar adelante esto, realizamos un diseño que permite contemplar el modelado de una solución de persistencia adecuada para el problema utilizando Entity Framework (Code First).

El sistema que desarrollamos, pone foco en los siguientes requerimientos funcionales:

- **Búsqueda de puntos turísticos por región y por categoría**

  Los turistas, pueden acceder al sitio web y explorar los lugares que deseen visitar según la región de nuestro país y a través diferentes categorías definidas.

  Según lo acordado con el cliente, las regiones no cambian con el tiempo, y son las siguientes:

  - Región metropolitana.
  - Región Centro Sur.
  - Región Este.
  - Región Litoral Norte.
  - Región “Corredor Pájaros Pintados”.

  Dentro de cada una de las regiones, se representan los puntos turísticos, los cuales pertenecen a diferentes categorías (deben tener al menos una). Estas categorías, a diferencia de las regiones, pueden variar en el tiempo, por lo tanto, el sistema provee la funcionalidad de que un usuario administrador pueda introducir nuevas categorías cuando así lo desee.

  Finalmente, cada punto turístico tiene un nombre, una descripción de máximo 2000 caracteres y una imagen asociada.

- **Elegir un punto turístico y realizar una búsqueda de hospedajes**

  Los turistas, dado un punto turístico seleccionado previamente pueden realizar una búsqueda de hospedajes. Para realizar dicha búsqueda se debe ingresar: el punto turístico seleccionado, la fecha de check-in, fecha de check-out y la cantidad de huéspedes.
  La cantidad de huéspedes, se debe discriminar siguiendo las siguientes categorías:

  - Cantidad de adultos (13 años o más).
  - Cantidad de niños (2 a 12 años).
  - Cantidad de bebés (menos de 2 años).

  Una vez que el turista ingresa los datos mencionados previamente, se le muestra un listado de los hospedajes que se encuentran disponible para el periodo seleccionado. Cada uno de ellos cuenta con: un nombre, la cantidad de estrellas (1 a 5), el punto turístico al cual pertenece, una dirección, una o varias imágenes, el precio por noche, el precio total para el período seleccionado y una descripción de las características y servicios ofrecidos por el hospedaje.
  Para calcular el precio total se consideran la cantidad de huéspedes y las categorías de los mismos (adultos, niños o bebés), ya que a los niños se le cobra un 50% y a los bebés un 25% del valor del día asociado.

- **Realizar una reserva de un hospedaje**
  Los turistas pueden confirmar una reserva de un hospedaje (previamente seleccionado). Los datos de la reserva se completan automáticamente con los datos de búsqueda (check-in, check-out, cantidad de huéspedes), además el turista debe ingresar su nombre, apellido y el e-mail. Una vez realizada la reserva, estos reciben un código único que identifica la reserva, además de un número telefónico y un texto de información del contacto.

- **Consultar el estado actual de una reserva**
  Un turista puede consultar el estado actual de una reserva a partir de su número. El estado de una reserva está compuesto por: un texto que identifica el estado de la reserva, un nombre y una descripción.

Todos los requerimientos mencionados previamente, hacen referencia a funcionalidades que pueden llevar a cabo los turistas, pero también el sistema ofrece una serie de funcionalidades a ser utilizadas por los administradores. Nuestra solución pone foco en las siguientes funcionalidades:

- Iniciar sesión en el sistema usando su e-mail y contraseña.
- Dar de alta un nuevo punto turístico, para una región existente.
- Dar de alta un nuevo hospedaje o borrar uno existente, para un punto turístico existente.
- Modificar la capacidad actual de un hospedaje.
- Cambiar el estado de una reserva, indicando una descripción.
- Realizar el mantenimiento de los administradores del sistema.

Una vez especificadas las funcionalidades que nuestro sistema provee tanto al usuario (turista) como al administrador, nos centraremos en describir cómo es la solución que diseñamos.

Es por eso, que en una primera instancia, pensamos en cómo nuestro sistema se divide en agrupaciones lógicas, como las mismas se organizan y la jerarquía que existe entre ellas. Para modelar estos conceptos, realizamos un diagrama de paquetes, una estructura que nos permite visualizar cómo nuestro sistema está dividido en agrupaciones lógicas, como estas se organizan y sus jerarquías.

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).

Como se puede observar la solución está compuesta por los paquetes: *BusinessLogic, BusinessLogicException, BusinessLogicInterface, BusinessLogicTest, DataAccess, DataAccessInterface, DataAccessTest, Domain, DomainException, Filters, Model, RepositoryException, WebApi y WebApiTest*.

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.

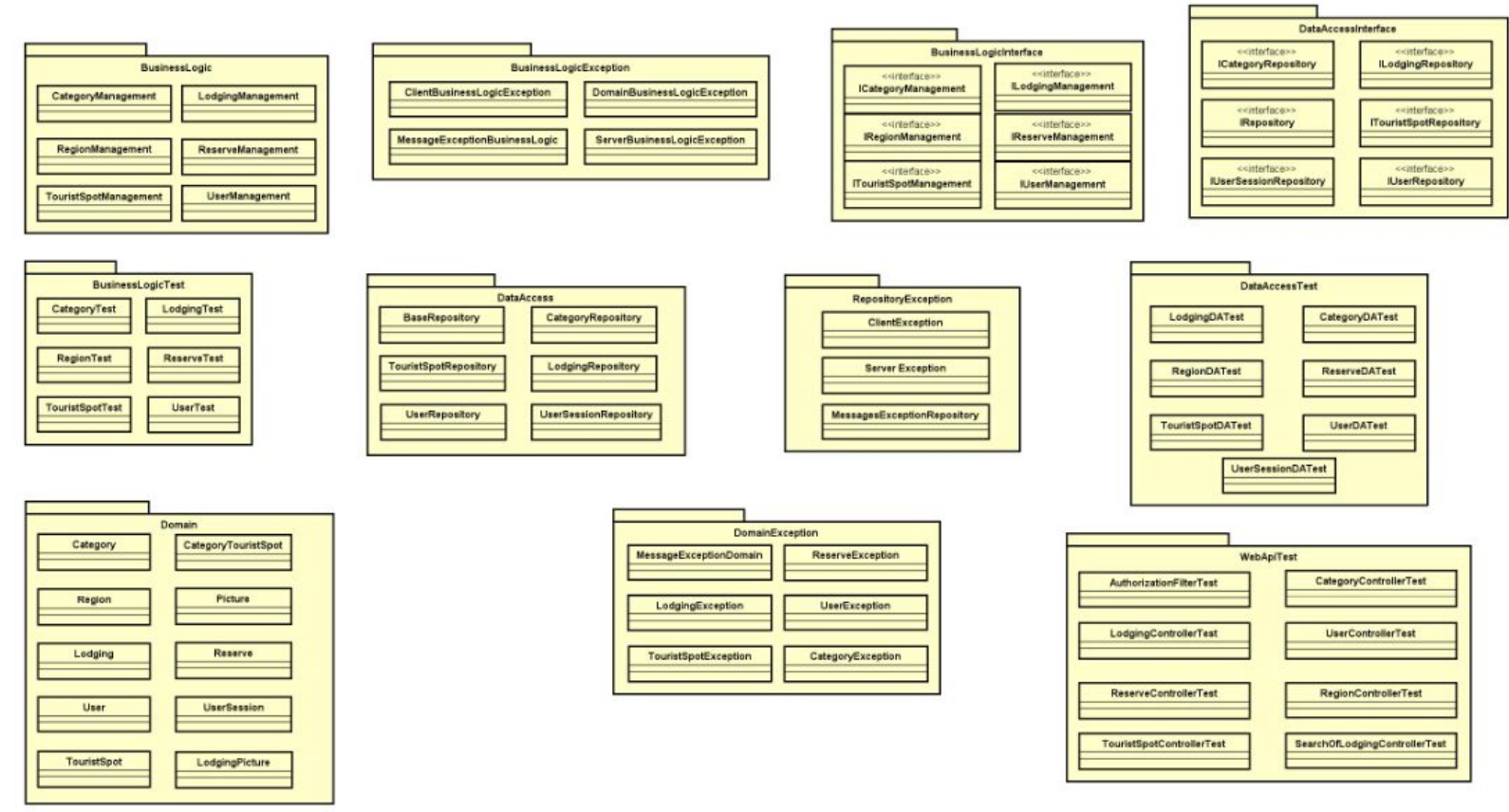


Diagrama de descomposición de los namespaces del proyecto (parte 1).

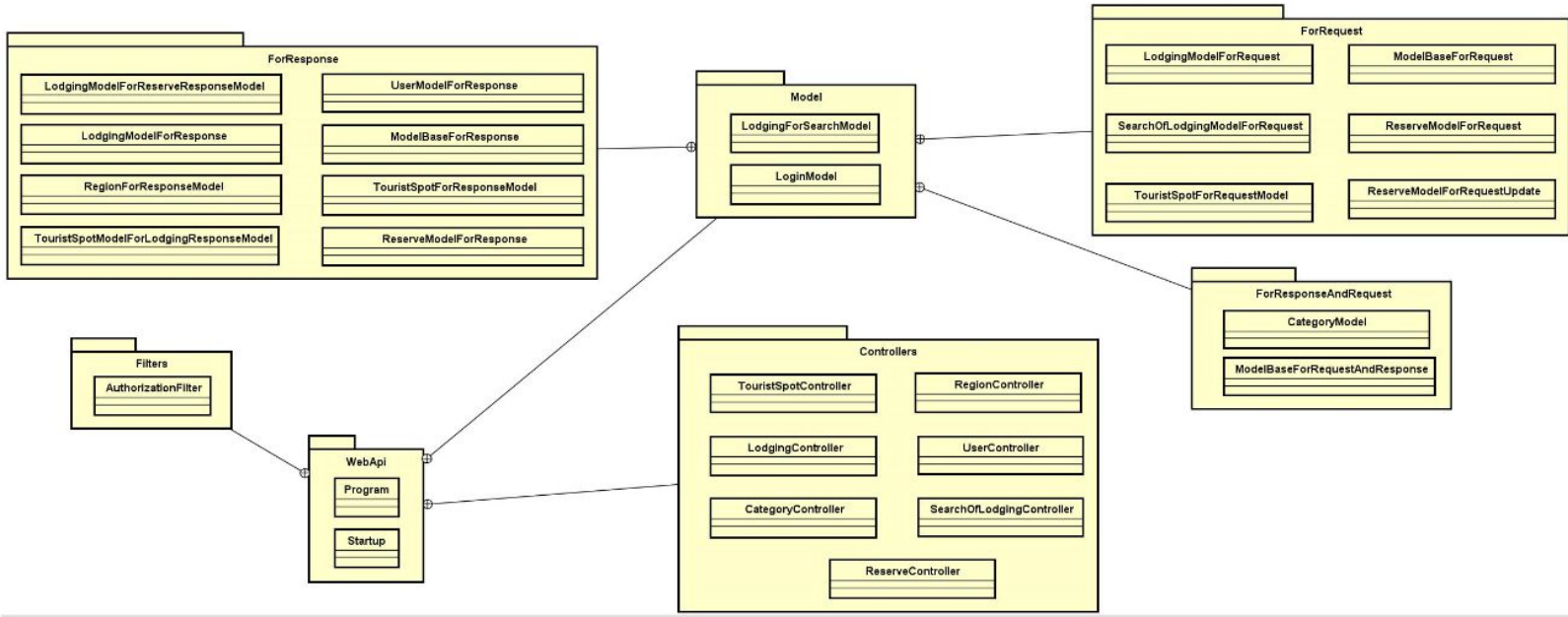
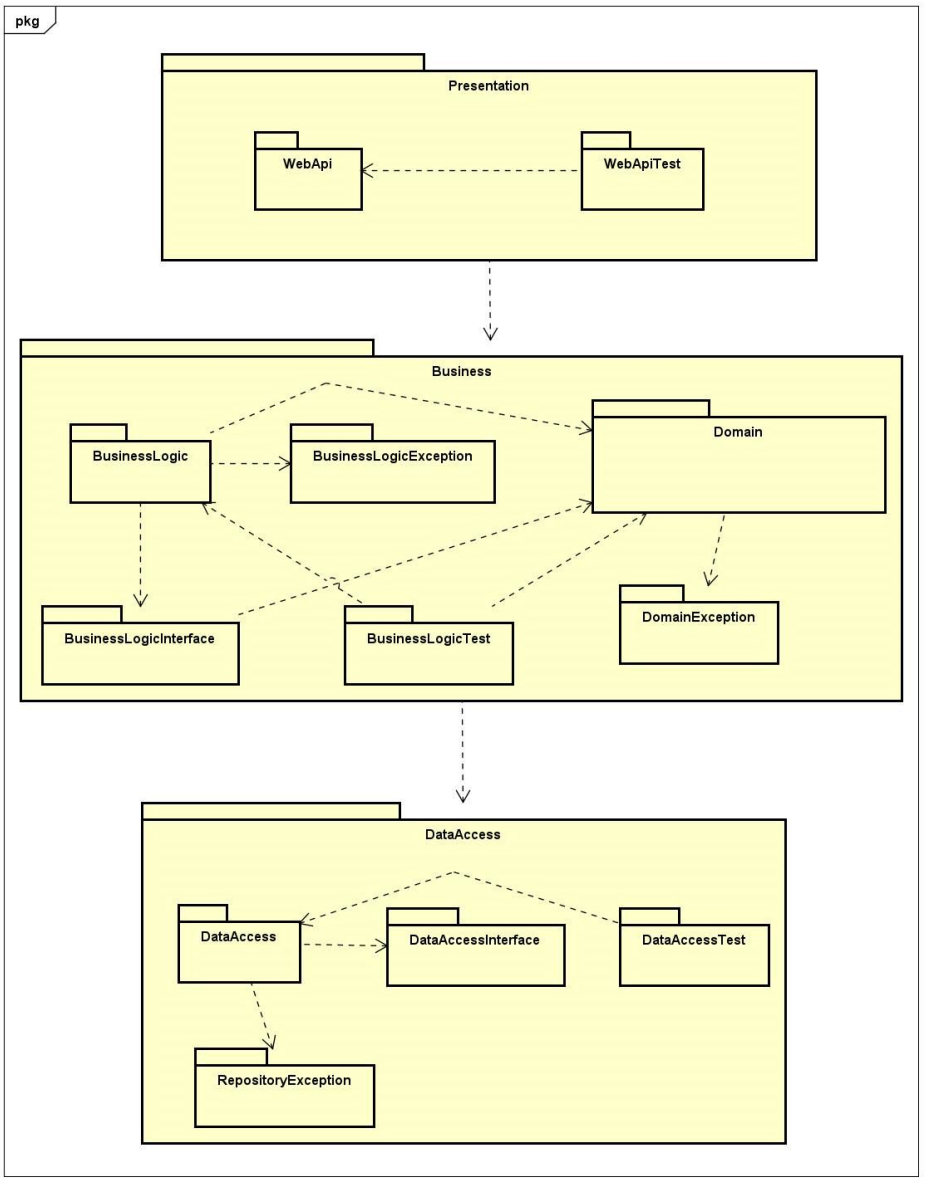


Diagrama de descomposición de los namespaces del proyecto (parte 2).

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)



Como se observa en el diagrama de paquetes precedente, nuestra solución está compuesta por tres capas (horizontales), en donde cada una de ellas cumple un papel específico dentro de la aplicación.

En nuestro caso la capa *Presentation*, 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.

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.

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. Por ejemplo, los componentes de la capa de presentación solo se ocupan de la lógica de presentación, mientras que los componentes que residen en la capa de lógica de negocios solo se ocupan de la lógica de negocios. Este tipo de clasificación de componentes facilita la creación de roles y modelos de responsabilidad efectivos en su arquitectura, y también facilita el desarrollo, la prueba, el control y el mantenimiento de aplicaciones que utilizan esta arquitectura debido a las interfaces de componentes bien definidas y el alcance limitado de los componentes.

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. 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*). El patrón DTO tiene como finalidad de crear un objeto plano (POJO) con una serie de atributos que puedan ser enviados o recuperados del servidor en una sola invocación, de tal forma que un DTO puede contener información de múltiples fuentes o tablas y concentrarlas en una única clase simple.
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 basicamente esta 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, asi como tambien 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 I*DataAccesInterface*, 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 del sistema, ya que si el dia de mañana, cambia la forma en que se almacena la información, simplemente habría que agregar un nuevo objeto que implemente la interfaz IRepository (y por tanto las operaciones), pero no sería 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 a 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 exista 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 que 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).

Justificación del diseño para cada paquete

Dentro del paquete *WebApi,* tenemos el paquete *Controllers*, cuyo diagrama de clases se presenta a continuación. Este paquete está compuesto por las clases: *CategoryController*, *RegionController*, *LodgingController*, *TouristSpotController*, *ReserveController*, *SearchOfLodgingController* y *UserController*. 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, obtenerlas todas y, por último, dar de alta una en el sistema.

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.

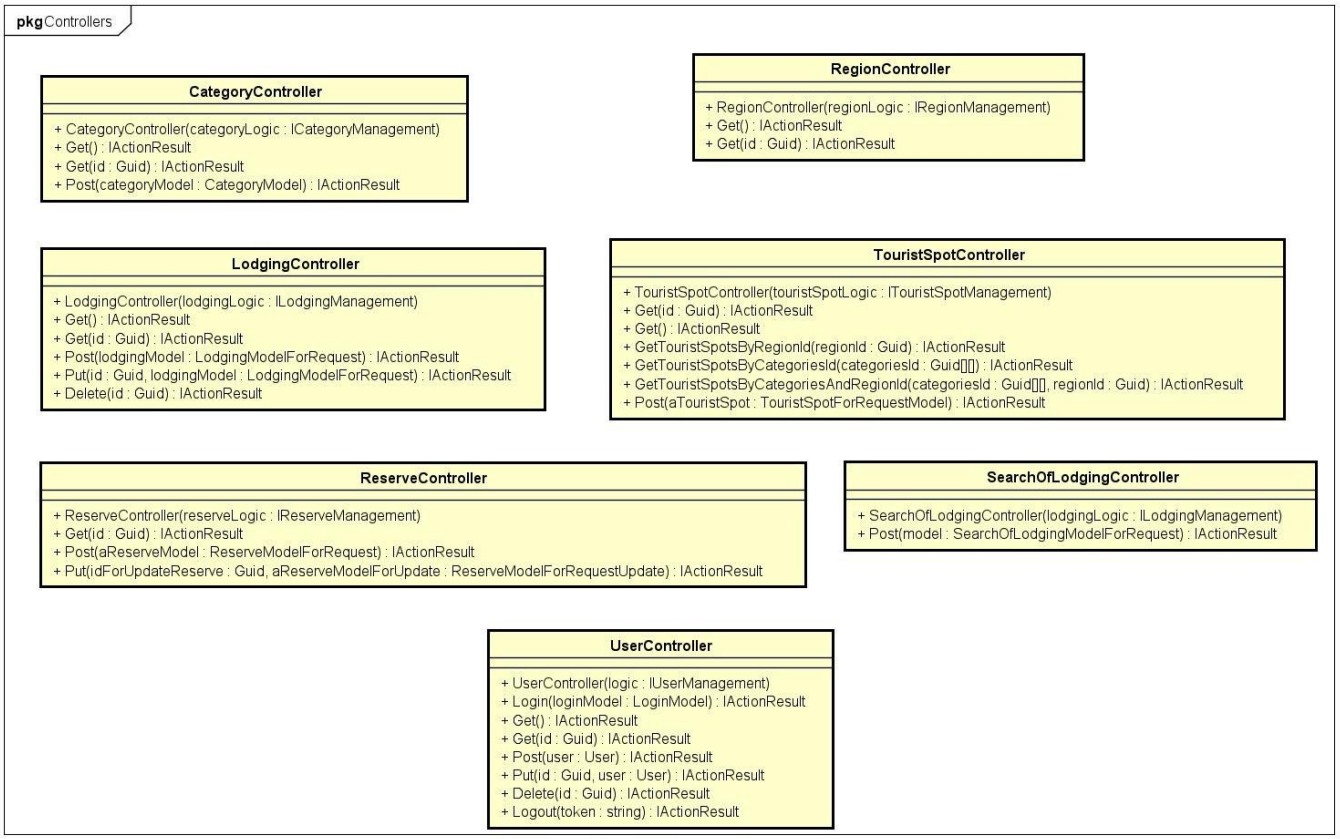
La entidad *LodgingController*, tiene operaciones relacionadas con los hospedajes, y básicamente, cuenta con las operaciones CRUD relativas a ellos (create, read, update y delete). Cabe aclarar que el cliente solicitó que el sistema debe poder permitir actualizar solo la capacidad de un hospedaje, pero nosotros creímos que podría ser adecuado que se puedan actualizar además otros campos del hospedaje, ya que si nos situamos en el contexto de uso de la aplicación, creemos que puede ser probable que el hospedaje cambie algunos de sus datos con el tiempo (nombre, cantidad de estrellas, descripción, dirección), y por lo tanto, el sistema debe ofrecer algún mecanismo de actualización en caso de que ocurran estos cambios.

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 encuentran en una región dado un identificador de la misma, obtener aquellos que tengan las categorías dadas por medio de una lista de identificadores de categorías, 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 y bebés), y el punto turístico previamente seleccionado.

La clase ReserveOfController, contiene operaciones relacionadas a las reservas de hospedajes, entre las que se incluyen: obtener una por un identificador, dar de alta una en un sistema dado un modelo que contiene: el hospedaje a reservar, fecha de check in, fecha de check out, cantidad de huéspedes (adultos, niños y bebés), 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.

Por último, tenemos el *UserController* que 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*.



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

La idea detrás del diseño de una API es poder exponer una interfaz que sea amigable para los desarrolladores que se integran a ella y a su vez que sea lo suficientemente robusta para evitar que surjan errores en la interacción cliente-servidor o frontend-backend. Por ende, todas las decisiones que tomamos velan por cumplir estos objetivos. Los DTOs por lo general nos ayudan a esto, a que tengamos un control estricto sobre lo que “aceptamos nos pasen” (requests) y sobre lo que “queremos devolver” (responses).

Los *DTO (Data Transfer Object)* los 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 usaramos directamente las entidades del dominio para retornar los datos en las *requests* lo que ocurriría es que terminamos 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.*

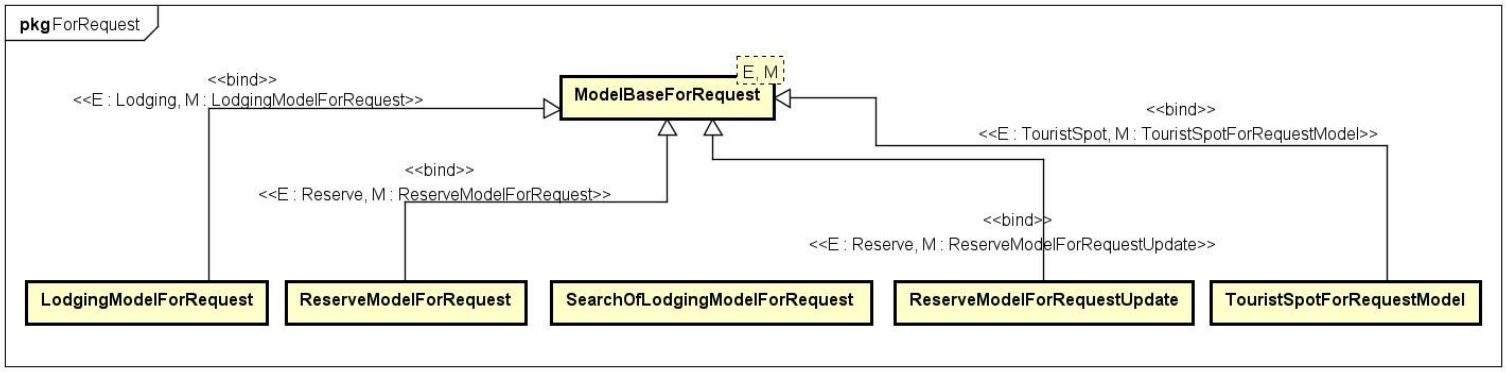
De esta forma tenemos dos tipos de DTOs en la API: los que modelan requests (por ej: *TouristSpotForRequestModel*) y los que modelan responses (por ej: *TouristSpotForResponseModel*).

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 ante un *HTTP GET* a un determinado endpoint. Un claro ejemplo de esto, es en el funcionamiento del login, cuando se realiza un *HTTP POST* para loguearse, sería deseable tener un modelo que incluya el *email* y la *password*, sin embargo, para dar respuesta a un *HTTP GET* a un endpoint de usuarios no voy a querer retornar su *password* por obvias razones de seguridad.

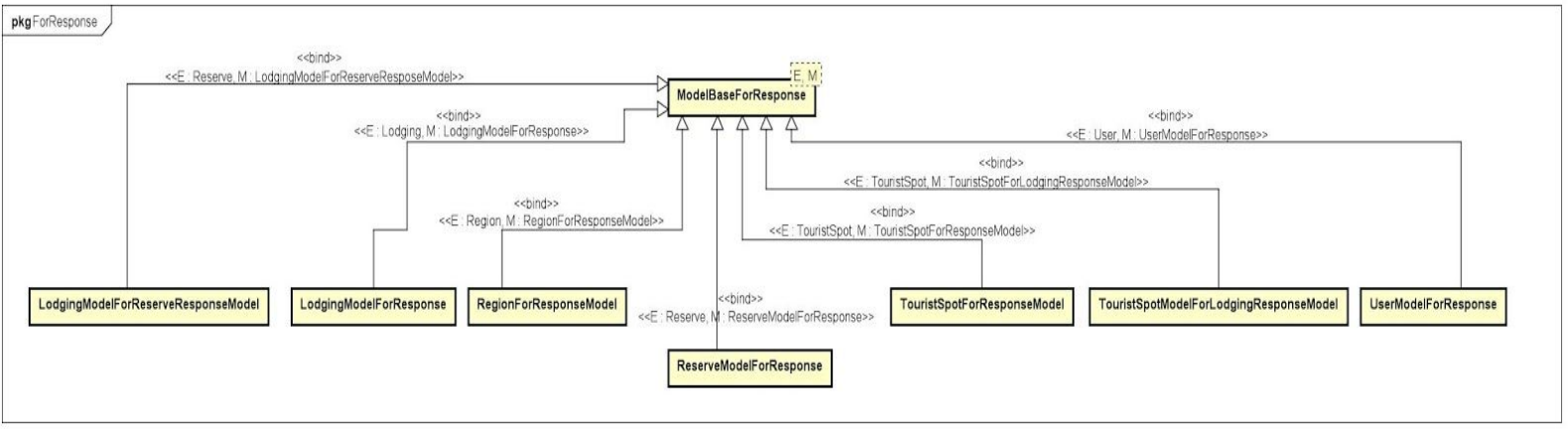
Veamos un diagrama de clases que modela estos dos tipos de DTOs mencionados previamente:

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.

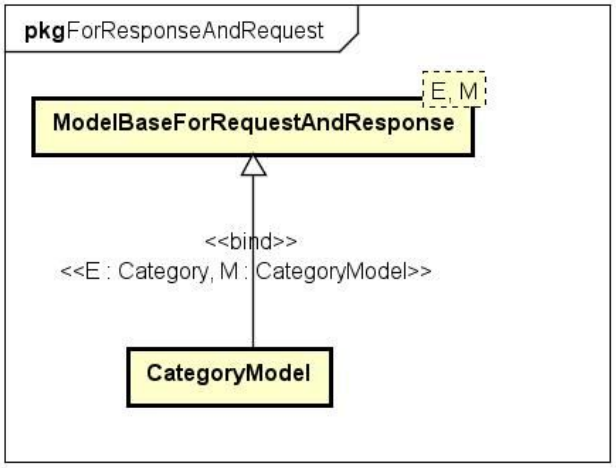
Otro aspecto a destacar en el diagrama, y que fue otras de las decisiones de diseño que tomamos, fue la de tener varios modelos para una misma entidad dependiendo de la operación que se esté llevando a cabo con esa entidad, podemos notar que contamos con las clases: *ReserveModelForRequest* y *ReserveModelForRequestUpdate*, que si bien ambos son modelos para la entidad *Reserve,* cuentan con diferente información, porque no queremos mostrar los mismos datos cuando se crea una reserva que cuando se actualiza, esto por los datos que se pueden actualizar únicamente.



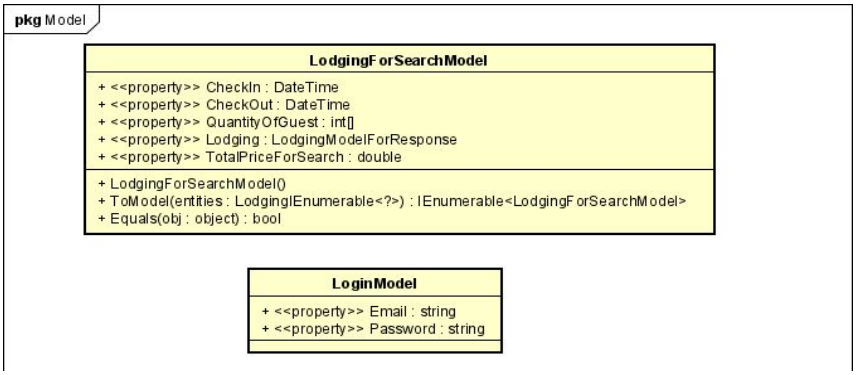
De forma análoga, tendremos lo mismo pero para las *responses*, en el siguiente diagrama podemos observar que todas las clases presentes heredan de una clase base genérica llamada *ModelBaseForResponse,* que tiene un método que toma una lista de elementos de una cierta entidad del dominio (E) y la convierte 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*, que tiene tanto el método para transformar una entidad en un modelo, como el que permite transformar un modelo en una entidad. En particular, en nuestra solución, este tipo de *DTO* lo utilizamos para las categorías puesto que los datos que se ingresan relativos a ellas son los mismos que los que se exponen.



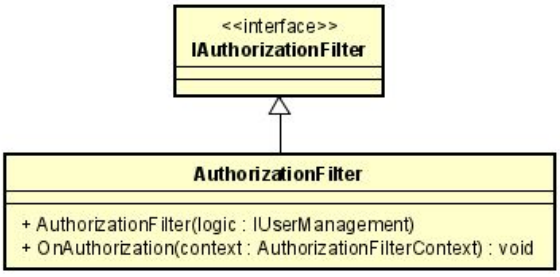
También encontramos que hay dos entidades dentro del paquete Model, que no se encuentran diagramadas dentro de ninguno de los paquetes anteriormente mencionados. Esto debido a que son modelos, los cuales no necesitan heredar de ninguna de las clases base, ya que no hay una necesidad previa de transformar 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.



Siendo así que como se puede notar, el modelo de inicio de sesión, no contiene ningun metodo, solamente tiene dos properties, esto debido a que únicamente se utiliza este modelo para realizar una operación específica que no está diagramada como una entidad existente, sino que a partir de dichos datos se realiza una operación de logueo, que tiene como resultado una sesión en el sistema.

Mientras que por otro lado, el modelo de hospedaje para búsqueda, no responde a la entidad hospedaje existente, esto debido a que tiene menos datos que la misma, y es únicamente utilizada cuando se realiza una búsqueda de hospedajes. En particular está diagramada de esta manera y tiene una operación *ToModel*, ya que al buscar hospedajes para cierto punto turístico se obtiene una lista de hospedajes. De tal manera que la transformación de una entidad existente en la base de datos, al modelo en especifico, no responde al criterio utilizado por la clase base *BaseModelForResponse*.

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, en ese sentido, decidimos implementar un filtro de autorización *(AuthorizationFilter),* que lo utilizamos para aplicar la política de autorización y seguridad. Entonces, en aquellas operaciones que requieran autenticación, colocando un tag del filtro de autorización, se verificará que el administrador esté logueado correctamente, y en caso de que no lo esté no podrá desarrollar la operación, esto debido a que estos filtros se ejecutan antes que cualquier otro y permiten evitar llegar al controller en caso de no cumplir con las políticas de seguridad.



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

cumpiendo 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.

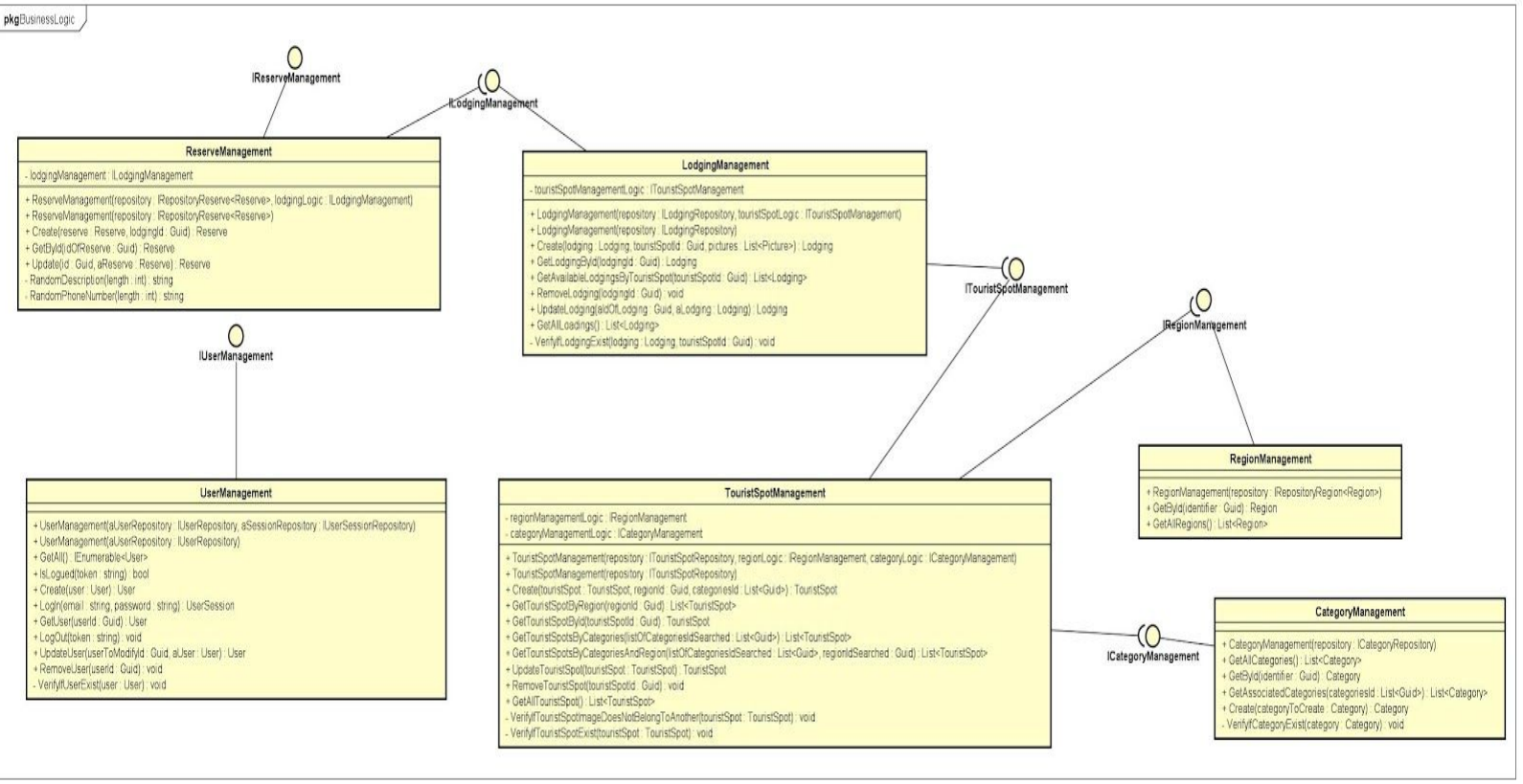
Es por esto, que como se podrá notar en los nombres tanto de las interfaces de las reglas de negocio, cómo la implementación de las mismas se posee en su nombre el término "Management". Haciendo énfasis en esto debemos especificar que una de las decisiones tomadas fue evitar el uso de una única clase conocida como "System" o "Sistema" en español. Por lo cual para cada una de las entidades que se encuentran en el Dominio, se le realizó una interfaz del manejador que le permite gestionar los eventos pertinentes a esta entidad, con la correspondiente implementación. Tal cómo se menciona en el párrafo anterior, esto se realiza de esta forma, para evitar depender de las implementaciones específicas, y estar dependiendo únicamente de los servicios expuestos, de manera que si se cambia la implementación, los servicios no se vean afectados por dicho cambio. Ya que se dependerá únicamente de la exposición de los servicios, y no de cómo ellos están implementados.
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 serian:

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. Es por esto que decidimos crear para cada una de las clases del dominio una interfaz de las reglas de negocio, y también una implementación de las mismas, en las cuales las únicas responsabilidades que tienen, es manejar los objetos del tipo del objeto del dominio y encargarse acerca de sus eventos.

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 presentación, 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.

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**.

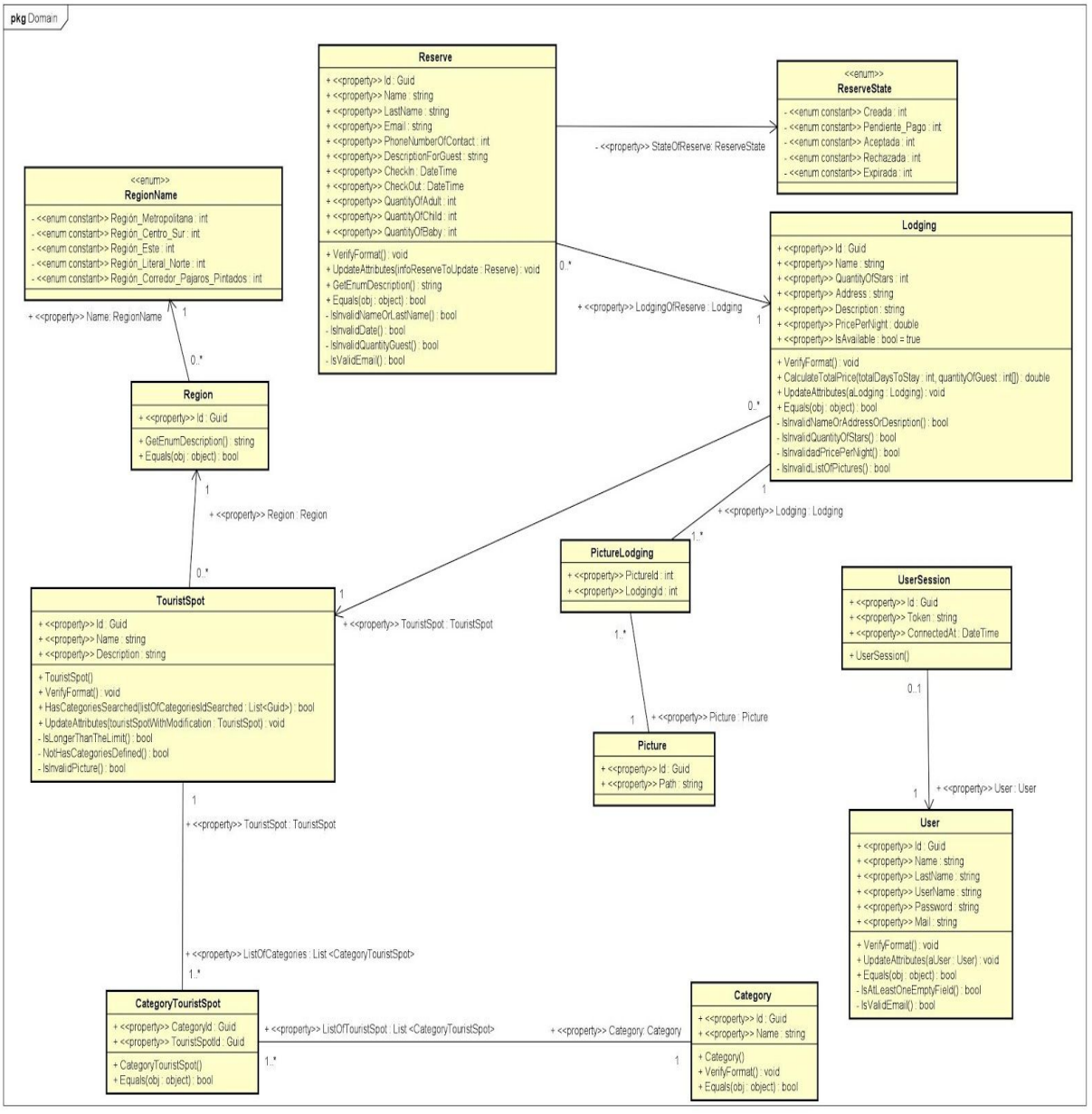
- Alta cohesión: Es la medida en la que un módulo de un sistema tiene una sola responsabilidad. Por ende, un módulo con alta cohesión será aquel que guarde una alta relación entre sus funcionalidades, manteniendo el enfoque a su único propósito. Nótese cómo este principio se relaciona de manera casi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

Dominio de la solución del problema

Siendo las clases existentes: Reserve, Lodging, Region, TouristSpot, Category, User, UserSession, Picture, CategoryTouristSpot y PictureLodging.

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 comportamiento[2].



[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, 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 conllevaria 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*.

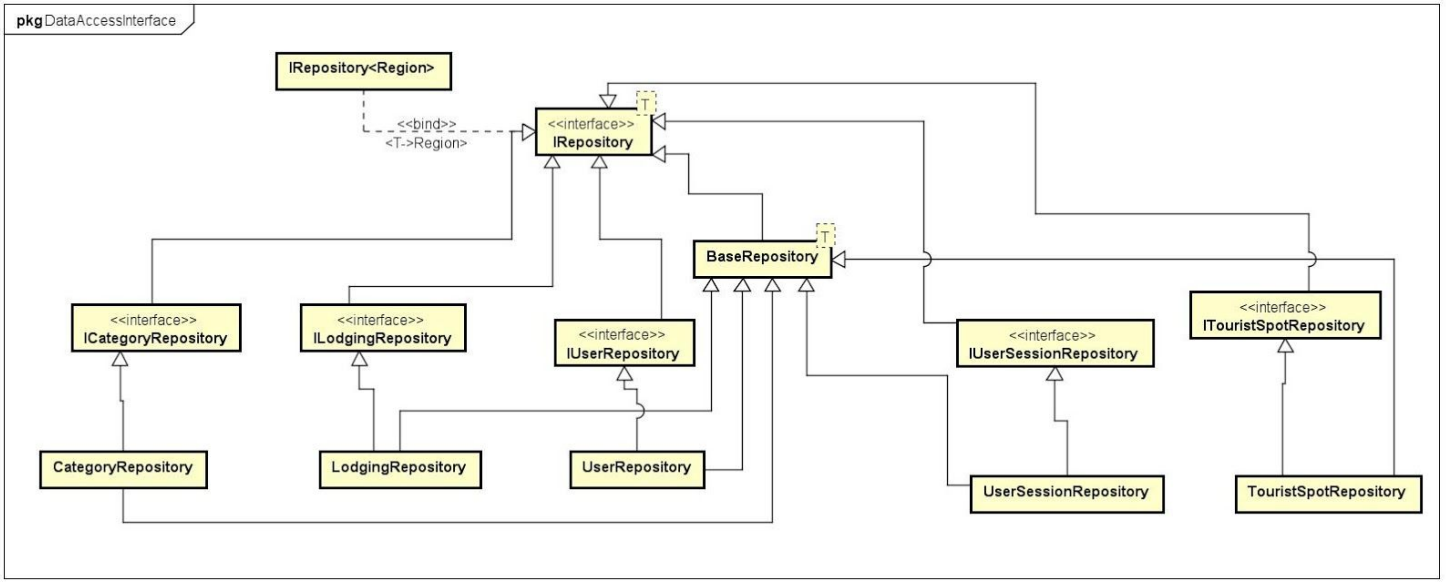
Pero con solo tener las operaciones CRUD no nos alcanza para desarrollar nuestra solución, en muchas ocasiones queremos obtener un determinado registro dada una condición, obtener registros por un determinado atributo que no sea el identificador, entre otros.

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.

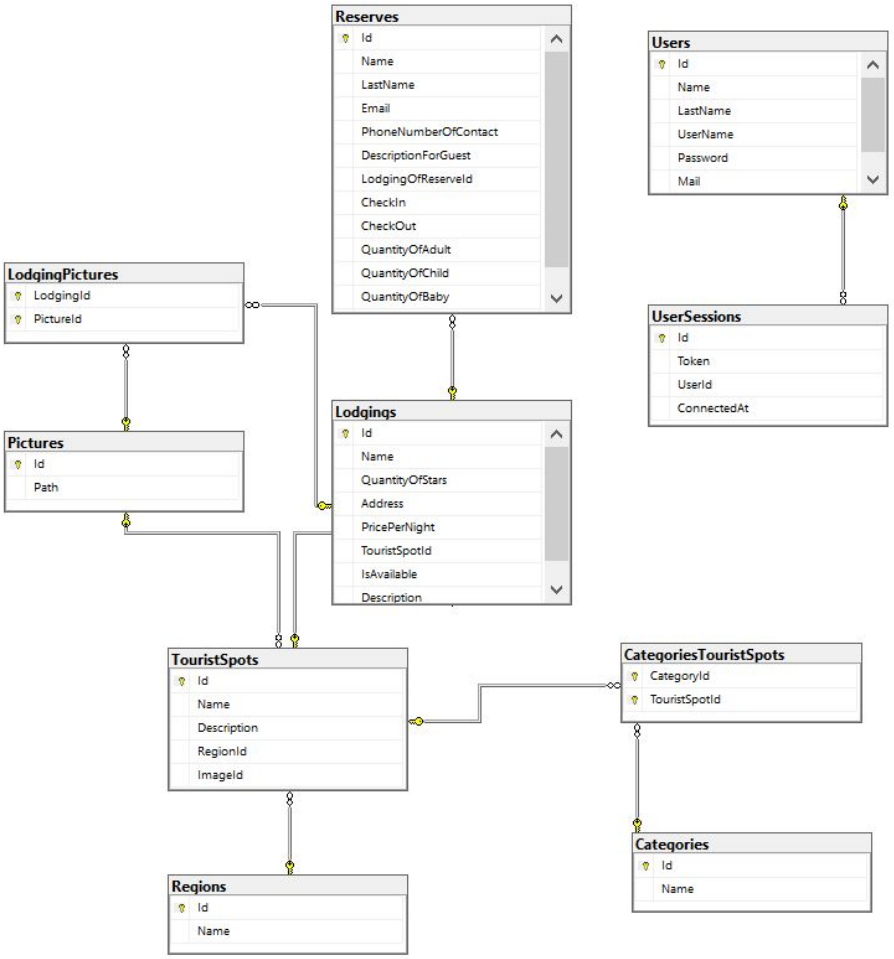
La otra opción disponible que fue por la que nos inclinamos es la de implementar para cada entidad que requiera de operaciones particulares, un repositorio particular con su correspondiente interfaz que exponga las operaciones del repositorio. Para cada entidad, lo diseñamos mediante una interfaz en 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 realizó el siguiente diagrama de clases (a alto nivel). 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



La solución consta de 10 tablas, en 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.

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

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.

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*.

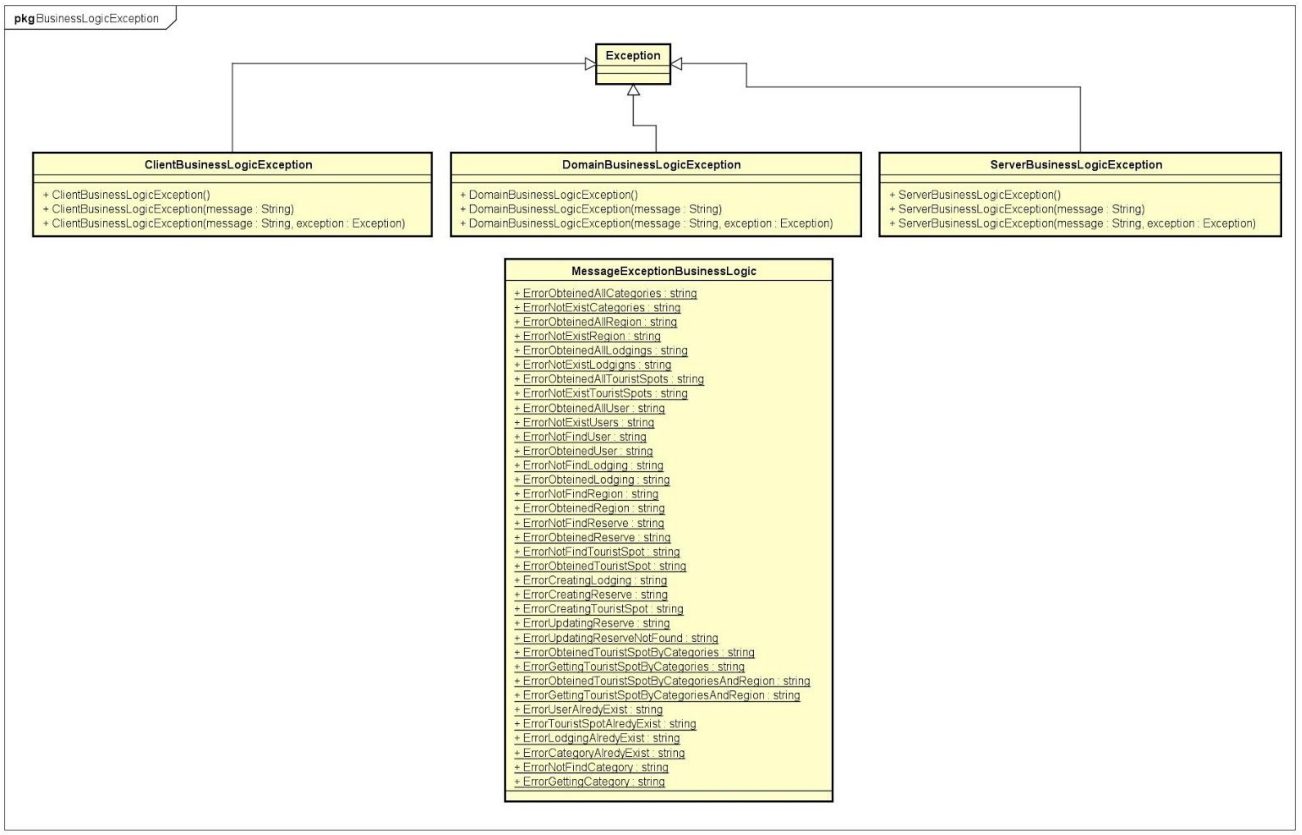
La *ClientBusinessLogicException* maneja las excepciones que de alguna manera fueron causadas por el cliente, entre las que se encuentran: obtener un registro por un id inexistente, intentar eliminar un objeto inexistente, entre otros. Estas excepciones son capturadas desde los *controllers* de la *WebApi* 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 (con algunos de los campos introducidos de forma incorrecta). Estas excepciones son capturadas desde los *controllers* de la *WebApi* lanzando códigos de error 400 (*Bad Request*).

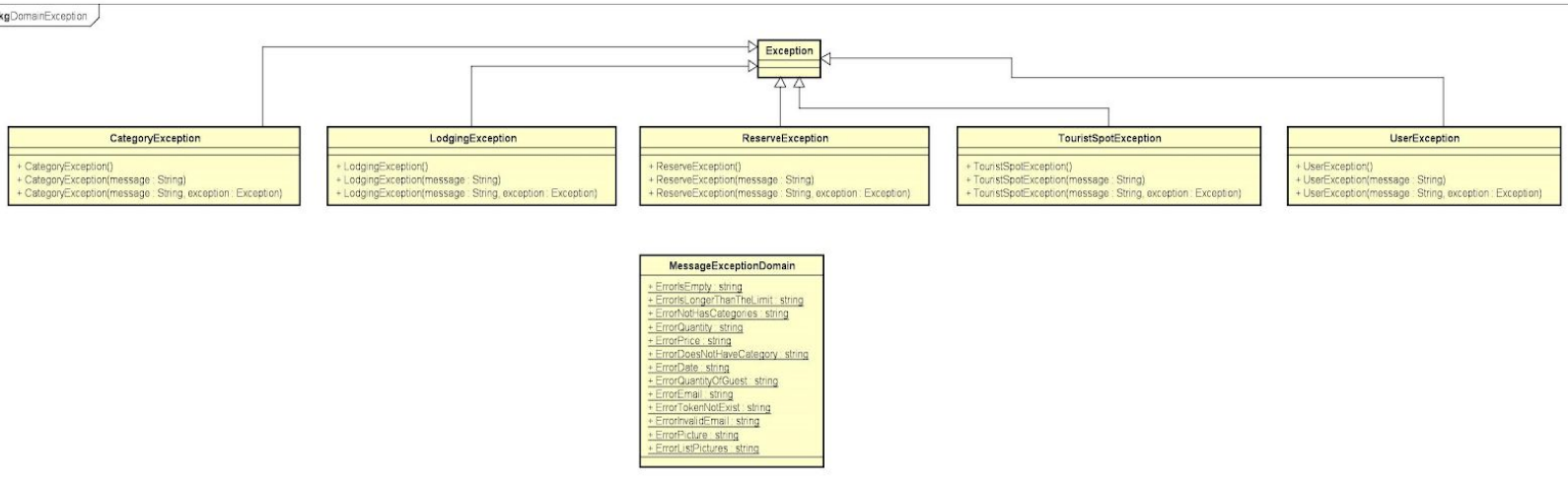
La *ServerBusinessLogicException* maneja las excepciones relacionadas a errores internos del servidor (como puede ser una desconexión de la base de datos). Estas, son capturadas desde los controllers de la *WebApi*, lanzando un código de error 500 (*Internal Server Error*).

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 mantenible en el tiempo, ya que si el dia 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

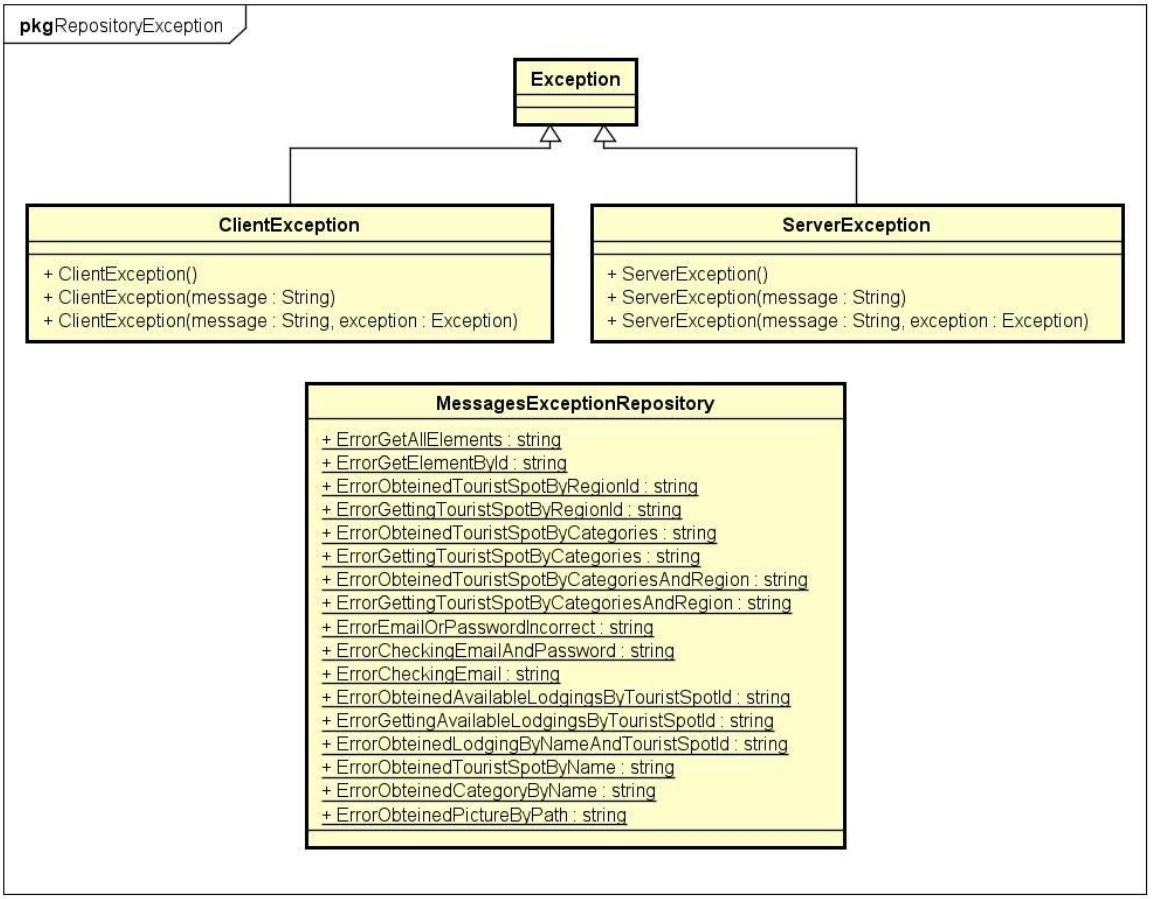
(que es lo que ocurriría si no contáramos con una clase como la *MessageExceptionBusinessLogic).*



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. El mismo cuenta por una clase por cada una de las entidades del dominio de nuestro sistema: *CategoryException, LodgingException, ReserveException, TouristSpotException, UserException*, 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 más arriba).

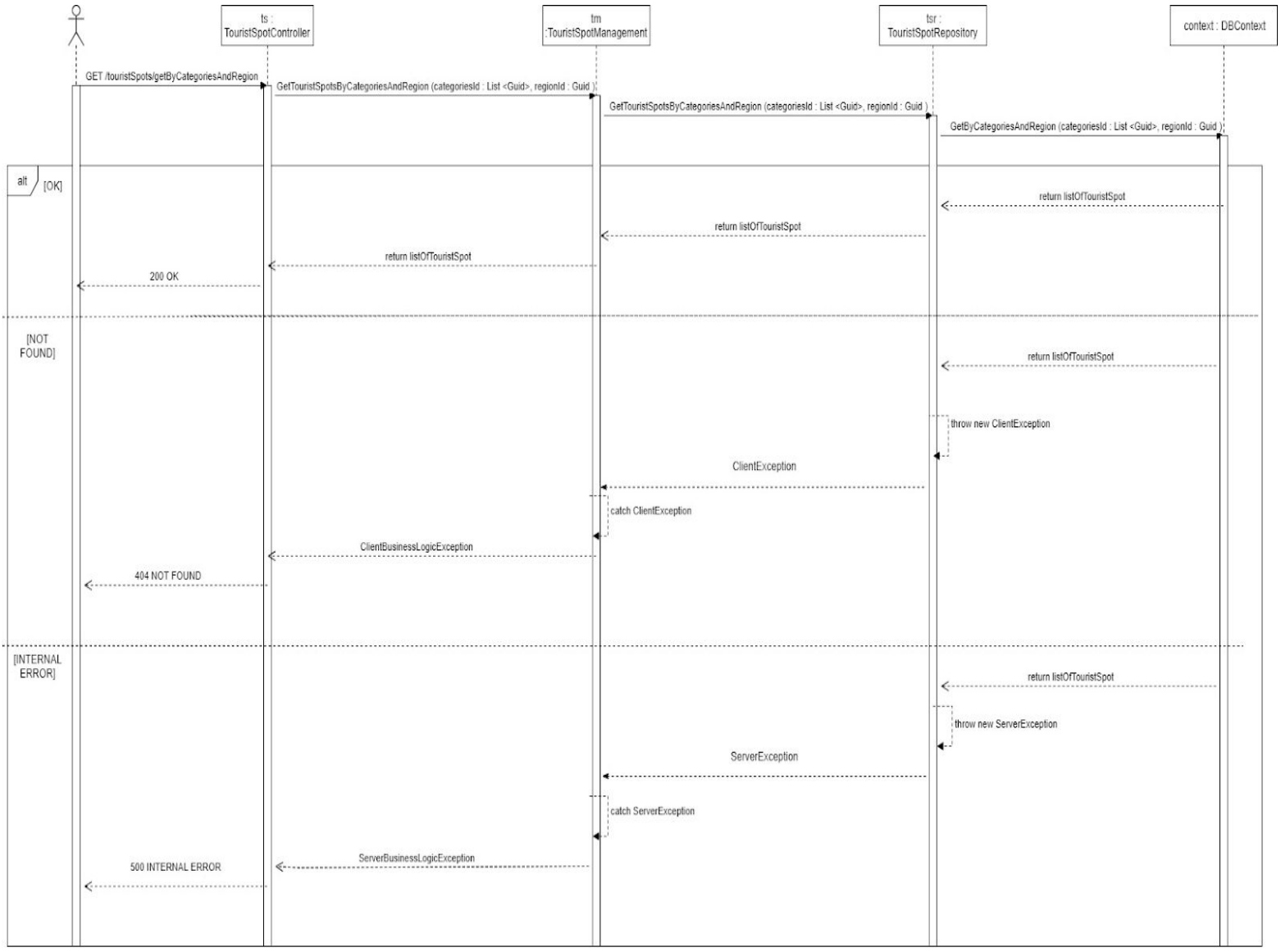


Finalmente, tenemos el paquete *RepositoryException* que se encarga de manejar las excepciones relacionadas a la base de datos, tanto aquellas causadas por errores del cliente (*ClientException*) como las causadas por errores internos del servidor (*ServerException*). Luego tenemos una clase que contiene los mensajes de error que se mostraran al lanzar las excepciones (*MessagesExceptionRepository*).



Diagramas de secuencia

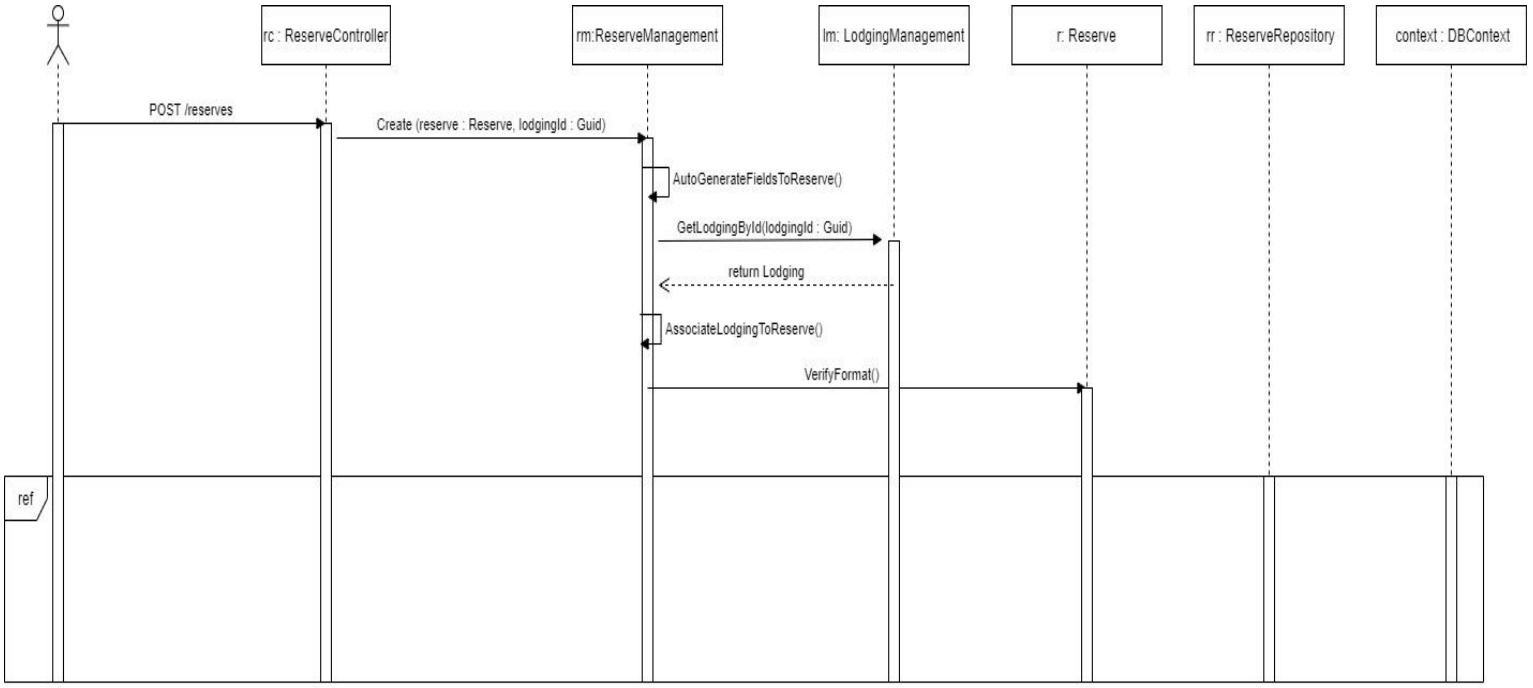
Obtener puntos turísticos por región y categorías



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.

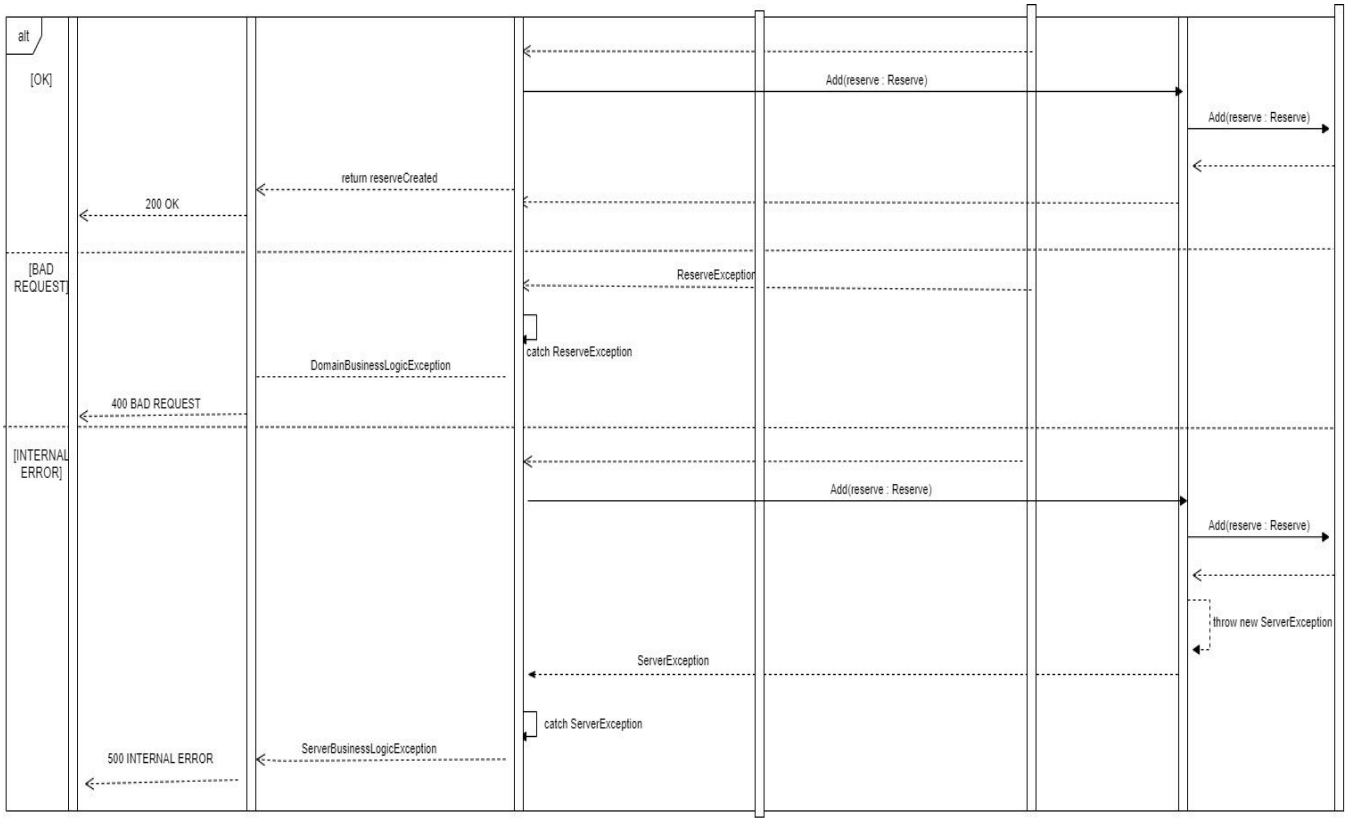
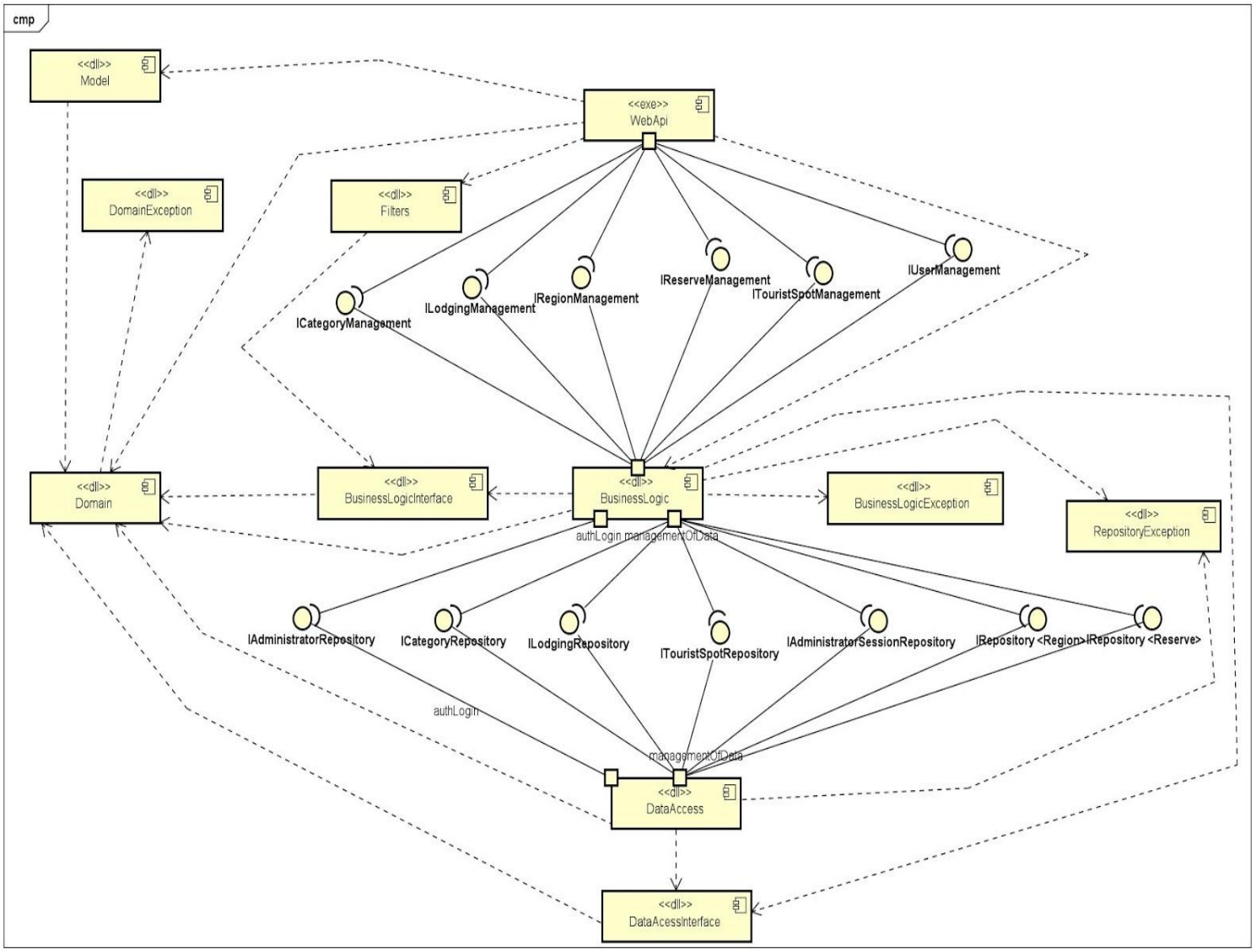


Diagrama de componentes



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 sea crea un componente de excepciones para cada una de las diferentes capas.