

Obligatorio 1 - DA2

Tourism

Santiago Rügnitz (215381)

Martín Gutman (186783)

<https://github.com/ORT-DA2/DA2-Obligatorio>

Índice

Descripción general	3
Link al repositorio	3
Descripción y justificación del diseño	4
Descripción general	4
Diagrama de paquetes organizados en capas	5
Paquetes y sus responsabilidades	6
Web API	6
Descripción de jerarquía de herencias	8
Business Logic Interface	9
Business Logic	10
Exceptions	11
Domain	12
Data Access Interface	14
Data Access	15
Modelo de tablas de la estructura de base de datos	16
Colaboración de las funcionalidades claves	16
Agregado de administradores	17
Agregado de puntos turísticos	18
Búsqueda de puntos turísticos por región y categorías	19
Justificación de diseño	20
Manejo de excepciones	20
Explicación de mecanismos o algoritmos que tienen impacto en la mantenibilidad o desempeño de la solución	21
Descripción del mecanismo de acceso a datos utilizado	22
Diagrama de Implementación	23
Evidencia del Diseño y especificación de la API	24
Restricciones de REST	24
Nombramiento de las URIs	25
Mecanismo de autenticación	26
Códigos de error y mensajes retornados	26
Descripción de los recursos	27
Accommodations	28
Administrators	29
Categories	30
Regions	30
Reservations	31
Spots	32
Evidencia de uso de TDD	33
Clean Code	33
Nombramiento	33
Nombramiento de paquetes, clases y métodos	34

Funciones	36
Otras recomendaciones respetadas	37
Diseño emergente	38
Desarrollo con TDD (Test Driven Development)	39
Metodología seguida (Outside In)	39
Demostración de aplicación de la metodología	40
Ejecución correcta de las pruebas	41
Cobertura de código	42
Web API	42
Business Logic	43
Domain	43
Data Access	44
Funcionalidades “importantes”	45
Búsqueda de hospedajes para cierto punto turístico	46
Realizar una reserva de hospedaje	46
Dar de alta un nuevo hospedaje	46
Borrar un hospedaje	47
Modificar la capacidad actual de un hospedaje	47
Cambiar el estado de una reserva	47
Reporte de casos de prueba con Postman	48
Búsqueda de hospedajes	49
Realizar una reserva de un hospedaje	50
Dar de alta un nuevo hospedaje	54
Dar de baja de un hospedaje	57
Modificar la capacidad actual de un hospedaje	58
Cambiar el estado de una reserva	59
Anexo	61
Diagramas de interacción	61
Autenticación de agregado de administradores	61
Agregado de administradores luego de la autenticación	62
Autenticación de agregado de puntos turísticos	63

Descripción general

Este documento consta de la documentación de la aplicación Tourism. En el mismo se podrán encontrar cuatro secciones que refieren a distintos puntos considerados a la hora de realizar la aplicación.

Estos puntos son:

1. Descripción general del diseño

En este punto, se encontrará una descripción de cómo fue diseñada la aplicación, y cuales fueron los patrones y principios que fueron considerados en esta etapa del desarrollo.

También se podrán encontrar ciertos diagramas respectivos a vistas del modelo 4 + 1, las vistas en las que se hizo más énfasis, fueron, la vista lógica, con los diagramas de clases, componentes, paquetes, secuencias, entre otros y la vista de desarrollo, con el diagrama de componentes.

2. Evidencia de diseño y especificación de la API

En esta sección se trabajará con el diseño de la API, y como se diagramó y pensó para que cumpliera con ciertas restricciones que permitirían que fuera considerada como una API REST. También se podrá encontrar una documentación generada con swagger que describe el funcionamiento de los métodos y recursos que son expuestos a los clientes.

3. Desarrollo con TDD y Clean Code

Esta sección contiene información sobre las recomendaciones de Clean Code, y su impacto en la legibilidad y entendimiento del código. Se exponen las recomendaciones tomadas y se muestran capturas de pantalla de ello.

En dicha sección, también se podrá encontrar la validación del trabajo guiado por las pruebas. Se encontrarán capturas que demuestran el uso de TDD y su respectivo enfoque, entre las cuales se destacan, el resultado y la organización de las pruebas, resultado del análisis de cobertura de código, entre otras.

4. Reporte de casos de pruebas con Postman

En esta sección, se puede encontrar información acerca de los casos de prueba realizados en Postman, haciendo énfasis en aquellos relacionados con las funcionalidades marcadas como prioritarias. Se incluye el archivo con la colección de pruebas y el entorno para una correcta ejecución de las pruebas.

Link al repositorio

<https://github.com/ORT-DA2/DA2-Obligatorio>

Descripción y justificación del diseño

Descripción general

El diseño de la aplicación se llevó a cabo siguiendo los patrones y principios generales del diseño.

Para comenzar, la misma fue diseñada para que seguir el principio de *“inyección de dependencias”*. Este fue el punto de partida para el desarrollo y el que permitió idear las capas que tendría la aplicación, tomando en cuenta justamente, que entre las capas se genere una dependencia respecto de paquetes de interfaces bien defendidas, y no respecto de las implementaciones.

Este primer punto fue importante para todo el desarrollo de la aplicación porque permitió una evolución independiente respecto de cada una de las capas. También implicó un enfoque específico para TDD debido a que las pruebas requirieron, dadas estas características, el uso de Mock, justamente con el fin de que se pueda dar esta evolución independiente.

Continuando, también se construyó la aplicación en base a los principios SOLID. Entre ellos, los más importantes y que más claramente se vieron aplicados, fueron el Single Responsibility Principle, ya que se intentó que todas las clases y métodos tuvieran un solo eje de cambio, y el principio de abierto y cerrado, Open Closed Principle, debido a que se ideó un código abierto a la modificación y cerrado al cambio.

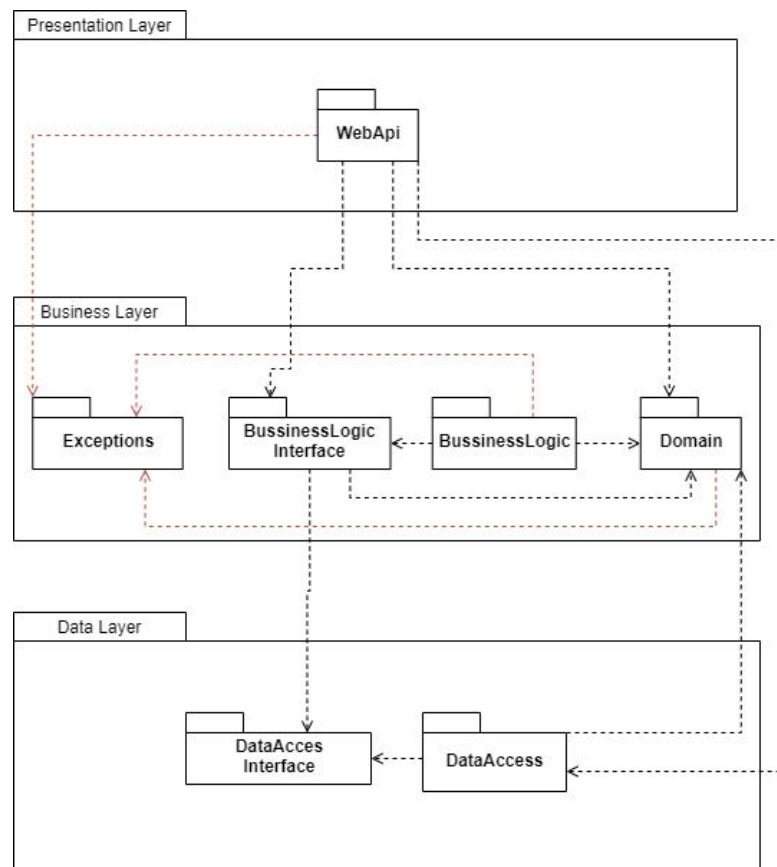
Por último, el más notorio de los principios SOLID que se siguieron, fue el Interface Segregation Principle, ya que no se tenían interfaces con muchos métodos y cuyos métodos fueron usados por distintos clientes, sino que se tenían interfaces específicas para cada uno de los clientes que las requirieron.

También se diseñó la aplicación siguiendo Principios GRASP, tales como Alta Cohesión y Bajo Acoplamiento, además de Ley de Demeter, Experto, entre otros.

Por último, se destaca, como se verá más adelante, el uso del patrón repositorio para el almacenamiento de los datos en la base de datos, lo cual permitió seguir los principios ya mencionados, además de que otorgó una gran flexibilidad a esta parte del código.

Esta idea de diseño llevó al siguiente diagrama de paquetes organizados en capas, los cuales se implementaron de la siguiente manera.

Diagrama de paquetes organizados en capas



Dadas las explicaciones anteriores, se decidió dividir a la aplicación en tres capas. La capa de presentación o frontEnd, que engloba las funcionalidades requeridas para que el usuario pueda llevar a cabo las acciones especificadas en las reglas de negocio.

Esta capa depende de la capa BusinessLayer, que es la capa que se encarga de modelar el dominio de la aplicación, implementar las reglas de negocio y exponer interfaces para que sean usadas por la API. La misma presenta los paquetes de dominio, que modelan los datos de las distintas entidades, interfaces definidas que anexan a esta capa con la superior, implementaciones de las mismas, y manejo de excepciones. A su vez, esta capa depende de un paquete de interfaces que exponen los métodos la capa de Datos.

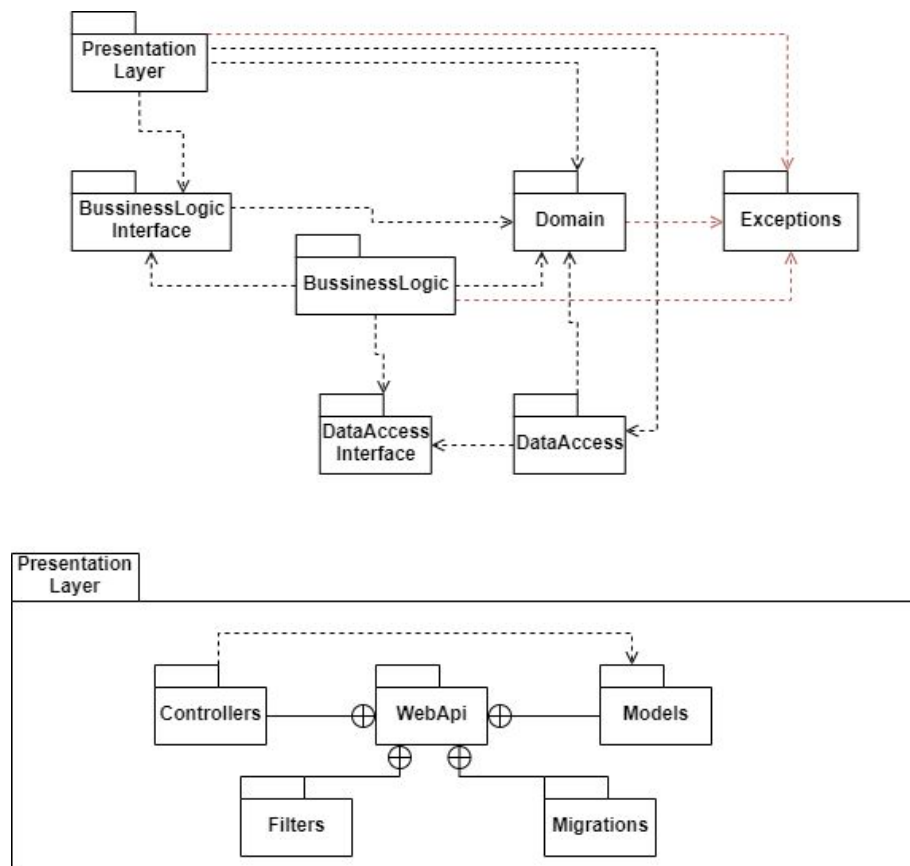
La capa de datos o Data Layer, es la que se encarga del manejo del almacenamiento de las entidades que están modeladas en el dominio, en la base de datos y se encarga del manejo de las operaciones que se realizan sobre la misma para trabajar con las entidades almacenadas. Presenta dos paquetes, uno que se encarga de dichas funcionalidades, y otro que expone una interfaz para que sea usada en la capa de reglas de negocio.

Se debe destacar, que la implementación y la diagramación de los paquetes se realizó siguiendo el patrón de “inyección de dependencias”. Este patrón indica que la dependencia entre capas no debe ser de implementaciones, sino que debe darse una dependencia por medio de paquetes de interfaces (contratos), bien definidos. Estos paquetes de interfaces son los ya mencionados `DataAccessInterface` y `BusinessLogicInterface`, quienes permiten la

correcta aplicación de este patrón. Esto permitió un trabajo más paralelizable en cada una de las capas, además de disminuir el acoplamiento entre las mismas y permitir una evolución independiente de cada una (siempre y cuando la interfaz no se viera modificada).

Esta implementación requirió un sistema de pruebas que utilizaba test doubles para funcionar. En este caso, por medio del uso de Mocks. Se puede encontrar el trabajo y realización de las pruebas en la [sección de "TDD"](#).

Paquetes y sus responsabilidades



En este primer diagrama, se pueden apreciar los distintos paquetes y clusters de los mismos, además de las dependencias que se generaron entre ellos.

Web API

El cluster de paquetes a destacar es el que se muestra denotado en el diagrama como Presentation Layer.

Este conjunto de paquetes engloba al paquete de **WebAPI**, este paquete es el que se encarga de todas las funcionalidades que expone la aplicación hacia el usuario (frontend), brinda métodos para que se pueda trabajar con los controladores y se requiere de las interfaces provistas por el backend para poder relacionarse con el mismo.

Este paquete tiene dentro 4 paquetes. Estos paquetes permiten a la API trabajar con los tres niveles de abstracción que hay en la solución.

El primer paquete a destacar, es el de **Filtros**, cuyas funcionalidades están muy arraigadas a los comportamientos que se pueden dar en distintas partes de la consultas y respuestas que se dan hacia y desde la API. Tienen dos funcionalidades marcadas que son las de autorizar las funcionalidades de los administradores, quienes quienes deben estar logueados para desarrollarlas; entre estas funcionalidades se destacan AVM de los propios administradores, alta de puntos turísticos, etc.

Se destaca también la funcionalidad de trabajar con las excepciones que generan los métodos invocados por los controladores, este filtro se encarga de identificarlas, y mandarlas al usuario de una manera adecuada, entendible y correcta. Esta última funcionalidad permite que no se tenga que repetir código en los controladores para que se encarguen de manejar estas excepciones, además de delegar de forma adecuada las responsabilidades de cada uno de los paquetes con el fin de cumplir el principio *“SOLID”*, *“Single Responsibility Principle”*, que indica que cada elemento del código tiene que tener solo un motivo para cambiar, lo que no se cumpliría en caso de que los controladores manejaran los errores y la autenticación de administradores.

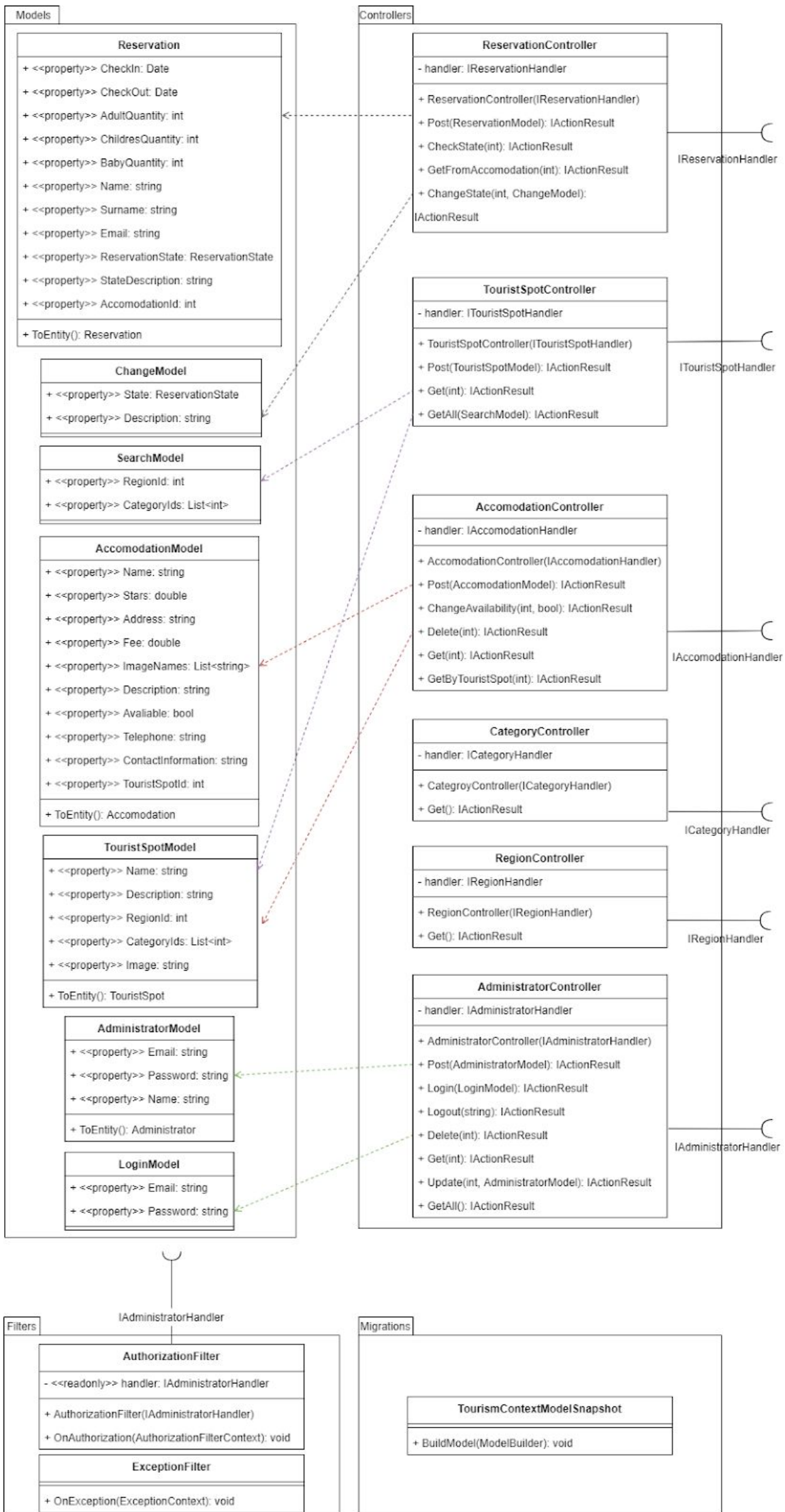
El paquete que trabaja a más alto (junto con el de los filtros) nivel es el de los **Controladores**, que son los que se encargan de manejar las funcionalidades expuestas a los usuarios. Permiten las consultas pasando parámetros en la URI, Header y Body y delega las funcionalidades al backend usando los modelos; creando una dependencia con el paquete que los alberga. Además, de depender del paquete que expone las funcionalidades de la capa de negocio, BusinessLogicInterface.

El paquete de **Modelos**, es el que se usa para manejar las requests que llegan a la API, trabajar con los datos que entran a la aplicación y transformarlos para que lleguen al backend, o hacer operaciones con los mismos. La funcionalidad principal de los modelos es la de recibir los datos de las distintas entidades y convertirlos a entidades soportadas por el backend (usando el método ToEntity()).

Por último, se pueden encontrar los paquetes de **Migrations**, la funcionalidad principal de este paquete es el modelado de la base de datos. Permite que la misma se cree y que trabaje de manera adecuada teniendo en cuenta la persistencia establecida en el dominio y en el paquete de DataAccess.

Se destaca además, que el paquete Web API es el que se encarga de la ejecución de la misma, y de la determinación de la implementación que van a tomar las distintas interfaces del proyecto en una ejecución dada. Esta tarea se lleva a cabo en la clase Startup, y es la que se da como resultado de la aplicación del patrón de “inyección de dependencias”.

La disposición de las clases dentro de los paquetes, métodos y dependencias quedan representados en el siguiente diagrama de clases. En el mismo, se pueden apreciar las dependencias, interfaces requeridas y dependencias que se dan internamente en el paquete de WebAPI.



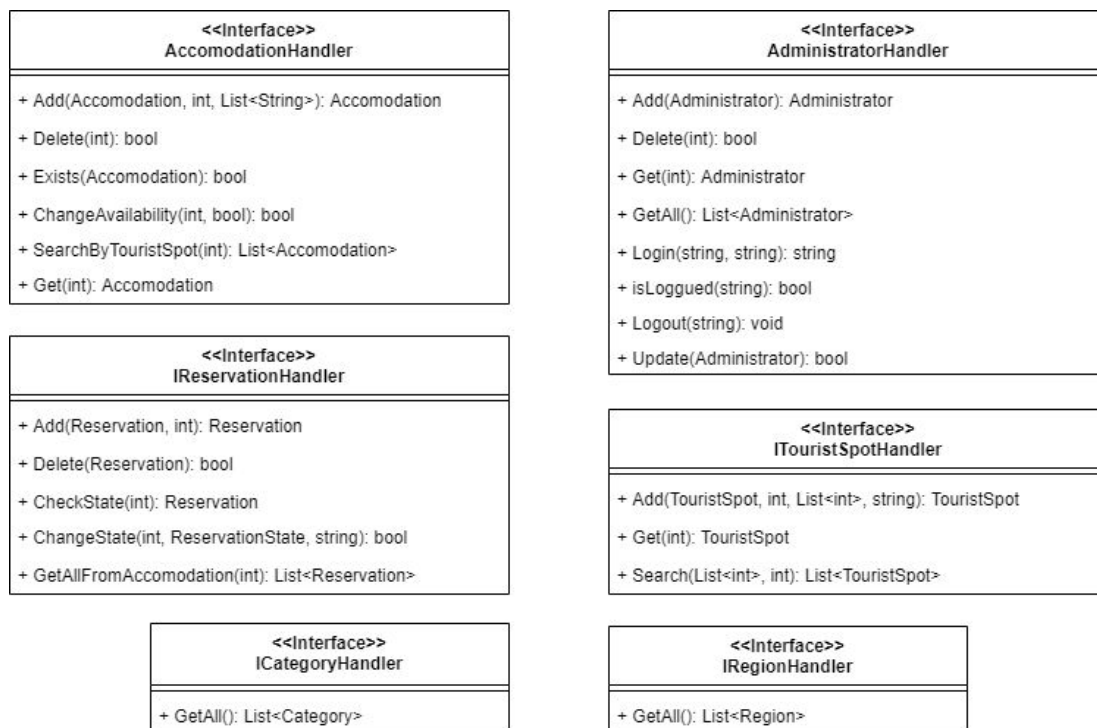
Como se puede apreciar en el diagrama, el paquete de Web Api requiere de 6 interfaces. Estas interfaces son las que permiten a la API trabajar de manera adecuada. Las mismas residen en el paquete de Business Logic Interface.

Business Logic Interface

Como se estableció, este paquete es el que expone 6 interfaces que son utilizadas por la Web API. Estas interfaces son las que se encargan de inyectar las dependencias entre las capas que albergan los paquetes mencionados y funcionan como contrato entre ellos.

Además, con la construcción de este paquete se logró aplicar el principio “*SOLID*”, “*Open Closed Principle*”, que indica que el código es cerrado al cambio y abierto a la extensión. Porque dado el uso de interfaces, se pasa a depender de abstracciones y no de implementaciones, por lo que se podría agregar una nueva forma de implementar el comportamiento requerido por las interfaces sin modificar el código existente. O hacer otros cambios similares.

También se siguió el “*Interface Segregation Principle*”, ya que se trabajó con distintas interfaces que permiten que distintos clientes pudieran depender de interfaces independientes, pequeñas y con pocos métodos. Esto permitió que no se dependiera de interfaces monolíticas y que no eran utilizadas al 100% y no se expusieron más métodos de los que se usarían, mejorando así la mantenibilidad y facilidad de cambio de la aplicación.



Estas interfaces son implementadas por las clases del paquete BusinessLogic.

Business Logic

Este paquete es el que se encarga de implementar las interfaces que residen en Business Logic Interfaces, definiendo su comportamiento y definiendo las distintas acciones que debe tomar la aplicación para cumplir con las reglas de negocio especificadas.

Las clases de este diagrama implementan las interfaces ya mencionadas, deben utilizar otras interfaces que pertenecen al paquete Business Logic Interfaces. La razón de que se requieran estas interfaces, reside en que hay ciertas entidades cuyo comportamiento requiere del uso de otros manejadores, por lo que requieren una interfaz de ellos para poder trabajar.

El ejemplo más claro de esto, es que la clase `TouristSpotHandler`, utiliza interfaces que exponen las funcionalidades de los manejadores de regiones y de categorías, ya que a la hora de agregar puntos turísticos, se necesita, por ejemplo, saber si, dado un identificador de región, si la misma se encuentra en la base de datos o no. Algo similar se da en la búsqueda, a la que se le pasa un identificador de regiones e identificadores de categorías, lo que requiere de verificaciones similares.

Las distintas clases de este paquete se realizaron siguiendo el principio “*SOLID*”, “*Single Responsibility Principle*”, ya que las clases que se encuentran en el mismo presentan un solo eje de cambio, mejorando así la mantenibilidad, estabilidad y facilidad de modificación de la clase.

Se aprecia que no se tienen clases monolíticas que exponen muchos métodos y que tengan por consiguiente, muchos ejes de cambio, sino que por el contrario, se tienen clases pequeñas, con un solo eje de cambio y por consiguiente, con una *alta cohesión*, principio GRASP que también se aplicó en los distintos paquetes de la solución, y que, nuevamente, contribuye, al entendimiento, modificación y reutilización de las distintas clases.

Lo establecido se puede apreciar en el diagrama de clases que se muestra a continuación.

Se destaca que este paquete es utilizado, dado la uniformidad de la implementación, por muchos paquetes a lo largo de toda la aplicación.

Está compuesto por dos clases, que modelan el comportamiento del programa frente a dos errores. Estos errores se relacionan directamente con los errores que son expuestos al usuario, y que están manejados en la Web API, específicamente, en el filtro de excepciones. Las clases y sus constructores se muestran a continuación.

BadRequestException	NotFoundException
+ BadRequestException() + BadRequestException(string) + BadRequestException(string, Exception) + BadRequestException(SerializationInfo, StreamingContext)	+ NotFoundException() + NotFoundException(string) + NotFoundException(string, Exception) + NotFoundException(SerializationInfo, StreamingContext)

Como se puede apreciar, los nombres de las clases corresponden con errores conocidos en los estándares de HTTP, BadRequest, se relaciona directamente con el código 400, y NotFound, se relaciona con el 404.

Estas excepciones se usan a lo largo de toda la aplicación y la relación entre los nombres y los códigos de error, se explicita en el ya mencionado filtro de excepción de la API.

Domain

Para completar la capa de negocio, se muestra el paquete Domain. Como ya se ha establecido, este paquete es el que se encarga de modelar las entidades que luego serán guardadas en la base de datos.

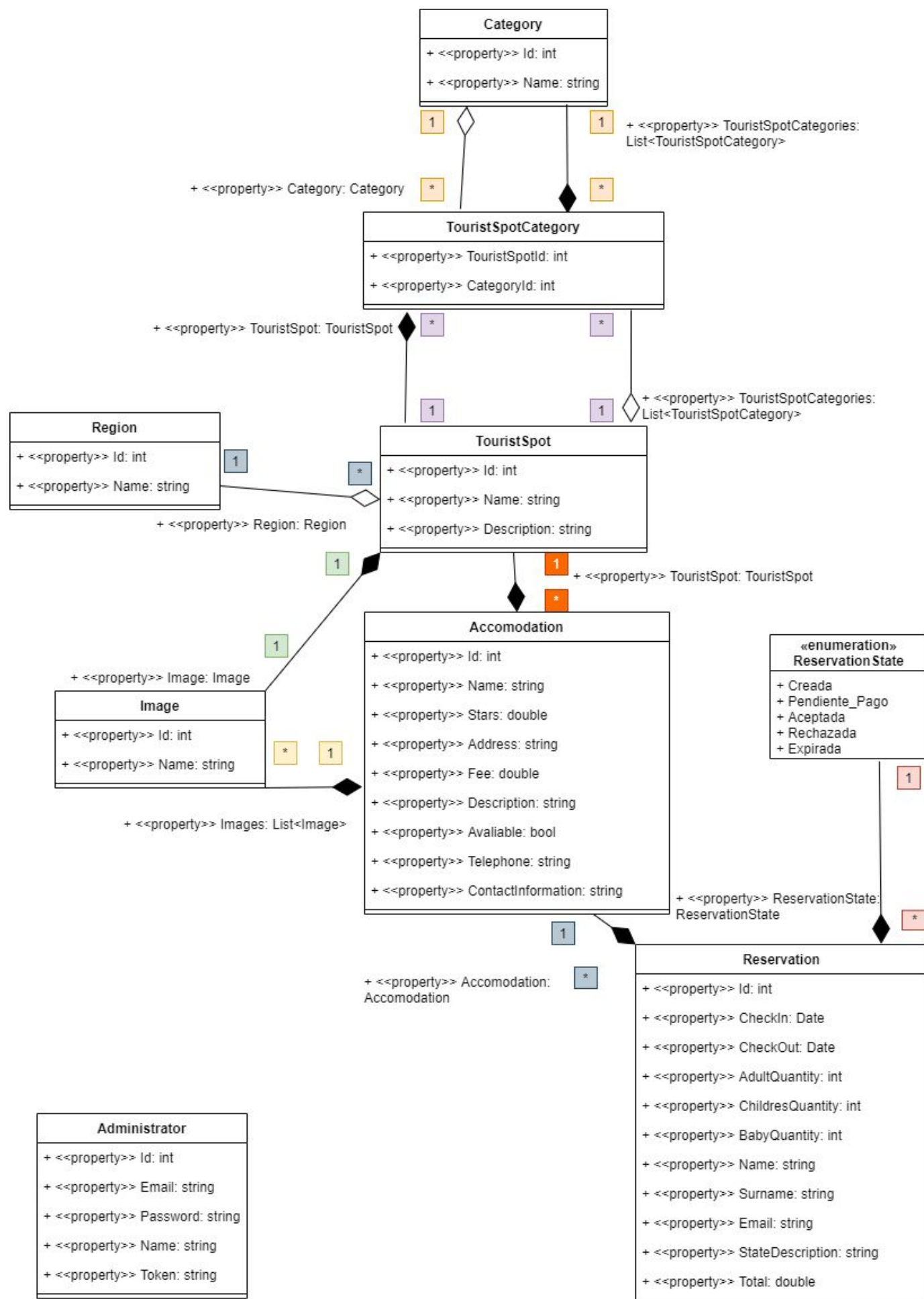
En particular, ocurre que muchos paquetes de la aplicación dependen de este, dado que representa el diagramado de las entidades que están presentes en el modelado del negocio, y por tanto, muchos paquetes lo utilizan para llevar a cabo sus funcionalidades.

De igual forma, se destaca que este paquete, solo depende de un paquete que es el paquete de excepciones, que se usa para realizar ciertas validaciones a cerca de los modelados, por ejemplo, para verificar que los nombres de administradores no sean vacíos, o que las listas de categorías asociadas a puntos turísticos no sean vacías.

Se debe destacar que existe la clase Tourist Spot Category. Esta clase es necesaria para el correcto almacenamiento de la relación que se da entre las clases Tourist Spot y Category. Fue implementada de forma mandatoria para que el framework, entity framework de .NET Core, lo pueda manejar de una forma correcta.

Esto se debe hacer porque la relación entre estas clases es Many-to-Many, lo que significa que una categoría puede estar asociada a muchos puntos turísticos, y un punto turístico puede asociarse a varias categorías, como se aprecia en las cardinalidades representadas en el diagrama.

Se muestra a continuación el diagrama de clases para este paquete, que muestra lo establecido.



Para continuar, se establece que las responsabilidades del almacenamiento de datos en la base de datos, de las entidades del domain, están a cargo de la capa de Almacenamiento, o Data Layer.

Esta capa consta de dos paquetes, uno que se encarga del almacenamiento propiamente dicho, y otro que se encarga de exponer las funcionalidades a las capas superiores.

Data Access Interface

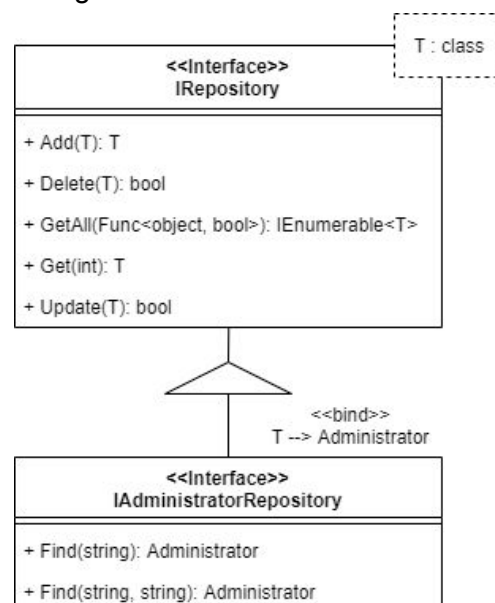
Este paquete es el que se encarga de exponer las funcionalidades a las capas superiores, funciona de manera similar al paquete Business Logic Interface. Es el que expone las interfaces de Data Access, requeridas por la lógica de negocio (Business Logic), y permite la inyección de dependencias entre las capas que corresponden a las clases mencionadas.

Al igual que el paquete mencionado, este paquete permite la inyección de dependencias, además del cumplimiento del principio “*SOLID*”, “*Open Closed Principle*”, ya que de manera similar a como ya se estableció, se depende de interfaces que posibilitan la extensión sin la necesidad del cambio. También se siguió el “*Interface Segregation Principle*”, destacado en el paquete Business Logic Interface.

Las clases de este paquete, fueron implementadas de forma que se pueda aplicar el *patrón repositorio*. Patrón que se explica más adelante, pero que permite un comportamiento genérico para todas las entidades que lo usen, sin necesidad de implementar una clase por cada una de las entidades con las que se debe hacer operaciones sobre la base de datos.

Se destaca además, el uso de la herencia de interfaces en el caso particular del administrador, ya que las operaciones sobre administrador requieren un login, por lo que se exponen estos métodos.

Se expone a continuación el diagrama de clases.



Estas interfaces se implementan en el paquete de Data Access.

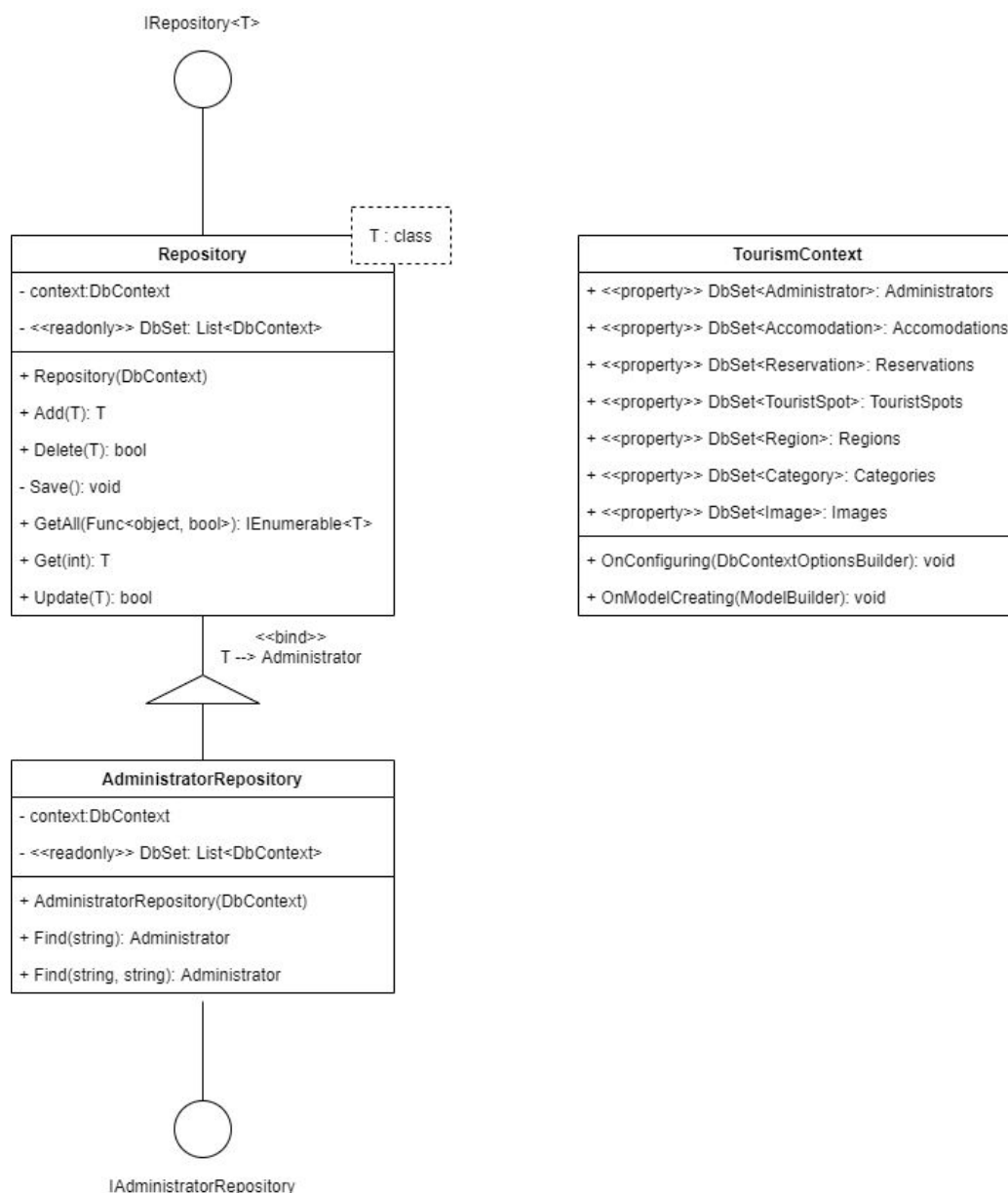
Data Access

Como se estableció, este paquete es el que se encarga de dar una implementación a las interfaces que se establecen en el paquete de Data Access Interface.

Las clases de este paquete le asignan un comportamiento concreto a estas.

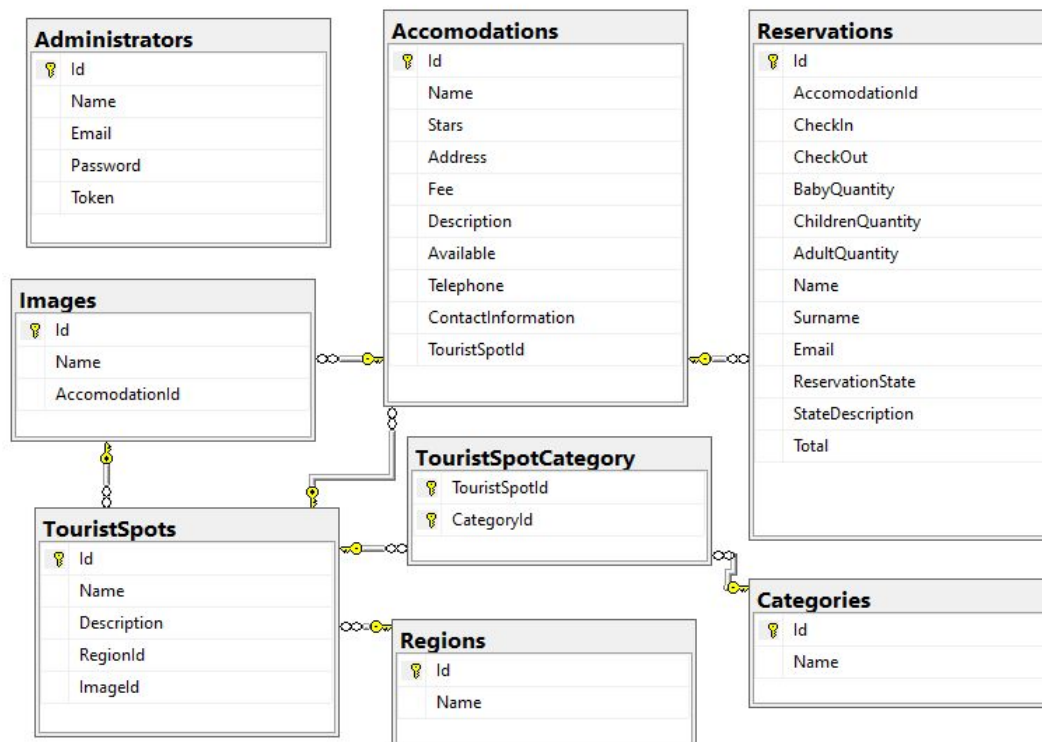
Se debe destacar, que el modelado de estos paquetes permitió, como ya se destacó, el trabajo con el *patrón repositorio*, este patrón se implementa en este paquete, en especial, en la clase Repository<T>. Esta es una clase genérica que implementa los métodos de CRUD de forma genérica (agregando un método GetAll que filtra por una función lambda).

Esto permite que sin importar la entidad que se quiera mantener, se implemente un comportamiento para hacerlo sobre la base de datos. Este patrón mejora claramente la centralización de la lógica de base de datos en un punto común y contribuye a una arquitectura extensible a nuevas entidades que se deban mantener en la base de datos (contribuyendo entonces al principio de abierto y cerrado). Se destacan también la implementación de la ya mencionada herencia, y las interfaces implementadas por las clases de este paquete. Se muestra el diagrama a continuación.



Modelo de tablas de la estructura de base de datos

Se muestra el modelo de las tablas a continuación.



Colaboración de las funcionalidades claves

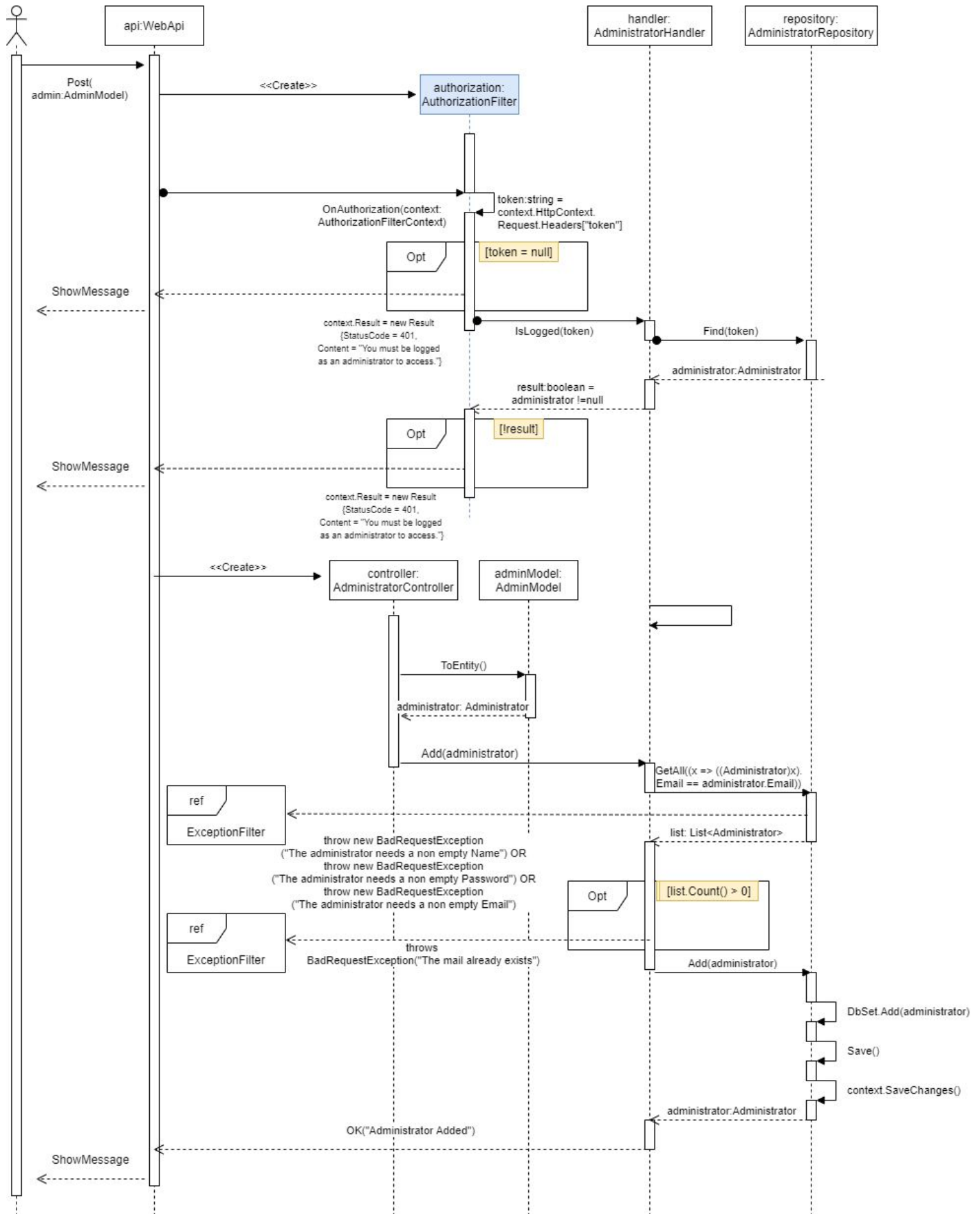
Las funcionalidades seleccionadas para hacer diagramas de colaboración son, el agregado de administradores, dado el login necesario y las condiciones a chequear. El agregado de puntos turísticos, dada nuevamente la necesidad de estar logueado y los casos borde sobre las regiones y las categorías. Y por último, la búsqueda de puntos turísticos por regiones y categorías, sus recorridas que constituyen a las interacciones. Se puede apreciar que para estos métodos se da comunicación entre clases pertenecientes a todas las capas, desde la capa de acceso a datos, hasta la Web API, y se muestra la secuencia de mensajes presentes para hacer posible la realización de la funcionalidad.

Se debe destacar que en caso de que en la ejecución se de una excepción, se utilizará el un campo con una guarda ref, que referencia a un diagrama de comunicación del método `OnException`, que reside en el filtro de excepciones de la API. Este filtro se encuentra modelado en la sección [“Manejo de excepciones”](#).

Se destaca también que no se mostrarán en los diagramas, los métodos implementados en Repository, debido a que los mismos son autoexplicativos, además de que su inclusión complejiza la lectura de los diagramas, además de que solo constan de llamadas a métodos internos o de clases que no son mantenidas por los desarrolladores. Como ejemplo de lo establecido, se muestra esta interacción en el agregado de Administradores (se muestra la interacción del repositorio).

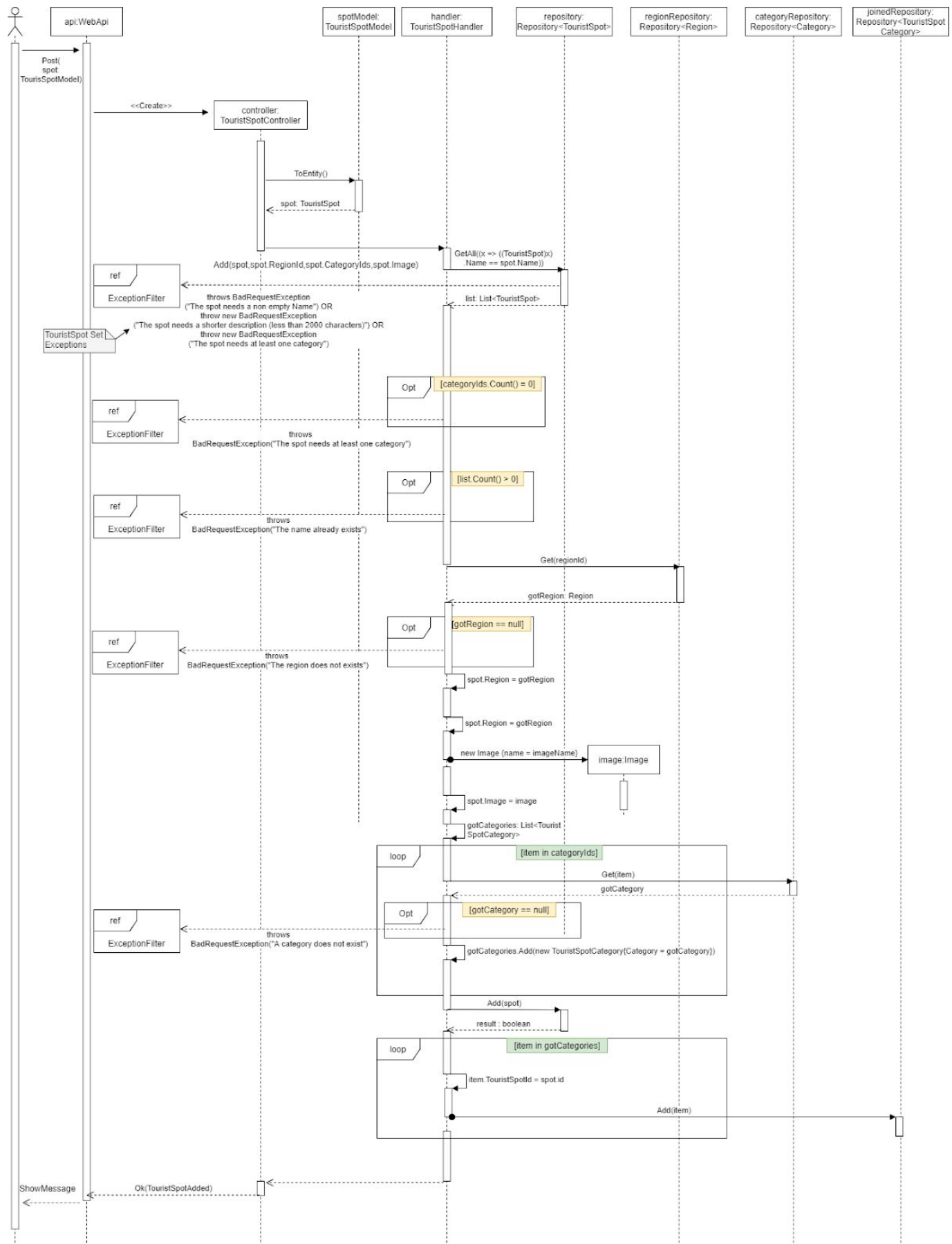
Agregado de administradores

Se muestra esta funcionalidad en un diagrama de interacción. [En el anexo](#), se pueden encontrar dos diagramas que parten el siguiente en dos, en uno se especifica la secuencia de verificación de usuario logueado, y en el otro, se especifica la interacción para efectivamente agregar al administrador.



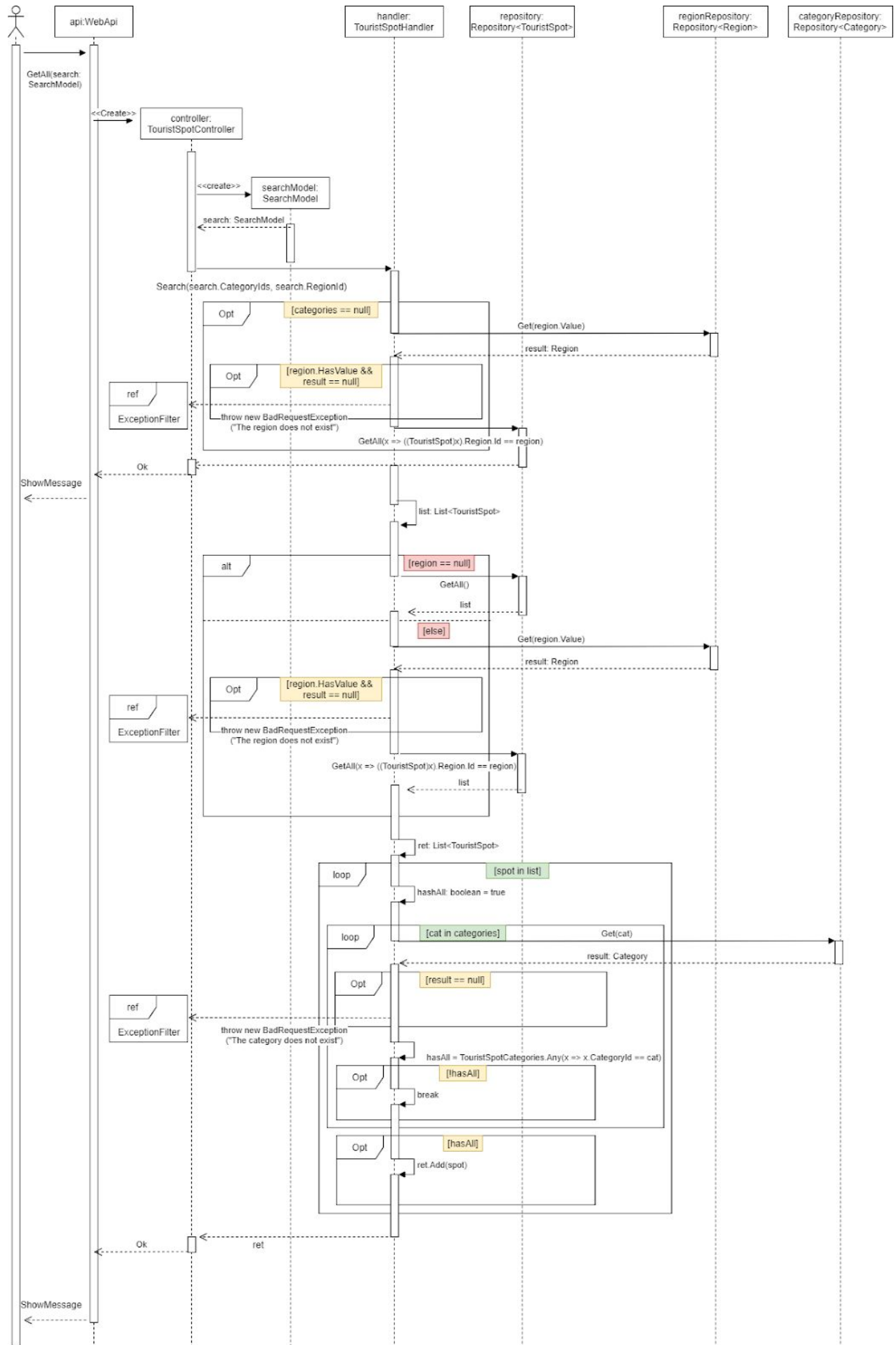
Agregado de puntos turísticos

Se muestra esta funcionalidad en un diagrama de interacción. [En el anexo](#), se pueden encontrar el diagrama que corresponde con la autenticación, este es muy similar a al diagrama que autentica al usuario en el agregado de administradores (solo cambia el método con el que se invoca).



Búsqueda de puntos turísticos por región y categorías

Se muestra esta funcionalidad en un diagrama de interacción.



Justificación de diseño

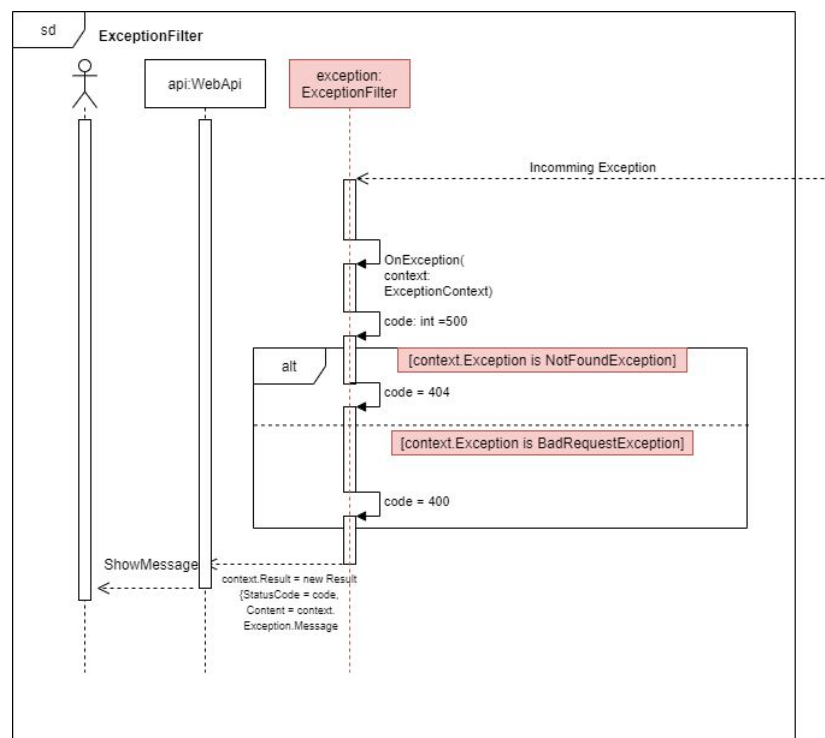
Manejo de excepciones

Como ya se mencionó en el paquete de excepciones, el comportamiento de las mismas reside y se engloba en dicho paquete.

Las excepciones generadas son de dos tipos, Bad Request Exception, y Not Found Exception. Estas excepciones son invocadas cuando se dan distintos errores en la ejecución y son capturadas en el filtro de excepción. Como ya se estableció, fueron denominadas en relación a los códigos de error 400 y 404 respectivamente.

Este filtro que se encuentra en la Web API, se encarga de verificar si se dio una excepción en el proceso de request y response de la API, y muestra un mensaje al usuario indicando lo que sucedió.

El diagrama de interacción que muestra cómo actúa este filtro, y como se vió anteriormente, es invocado en los métodos cuyo diagrama de comunicación se expuso, es el siguiente:



Se aprecia que las excepciones manejadas en el paquete, son las que se capturan en este filtro, y que posteriormente se le muestran al usuario con el código de error correspondiente.

En caso de que se de una excepción que no haya sido controlada en la aplicación, se mostrará un error cuyo código es el 500. Esto se puede generar, por ejemplo, por fallas no controladas en el repositorio.

Explicación de mecanismos o algoritmos que tienen impacto en la mantenibilidad o desempeño de la solución

Los mecanismos que se aplicaron que tienen impacto en la mantenibilidad son alguno de los ya mencionados. Entre los más importantes, se encuentran:

- El uso de interfaces y la inyección de dependencias
 - Este mecanismo aplicado para permitir una evolución independiente de los distintos módulos que presentaba la aplicación, también favorece al principio de abierto y cerrado, ya que en caso de por ejemplo, querer agregar una nueva implementación del guardado de datos, solo requeriría implementar este componente y modificar el program que establece la implementación que va a tomar cada interfaz en tiempo de ejecución. Esto favorece claramente a la mantenibilidad de la aplicación y a su desempeño en cambios futuros.

De la mano con este punto, se encuentra el probado de la aplicación con el uso de test doubles, que si bien está muy arraigado a la inyección de dependencias, también hace que las pruebas sean independientes y favorece la construcción de módulos intercambiables sin tener que cambiar los otros módulos.

- Guardado de Regiones y Categorías en la base de datos
 - A lo largo de todo el desarrollo, se evitó el uso de los magic numbers y magic strings, esto se ve claramente representado en el almacenamiento de las categorías y regiones en la base de datos. Entidades que si bien no estaban mantenidas en esta entrega, se almacenaron con el fin de evitar grandes modificaciones futuras en caso de que se vean modificados los nombres de las categorías, que se agreguen más, o lo mismo para las regiones.
- Estado de reserva como un enum
 - De la mano con el punto anterior, el estado de reserva se generó como un enum, evitando nuevamente magic strings, promoviendo un mejor entendimiento del código y esperando agregados y modificaciones futuras.
- Filtros de autenticación y de excepción
 - Estos filtros se consideran muy importantes a la hora de destacar la mantenibilidad y rendimiento del código. Como ya se explicó, los mismo permitieron la facilidad de que no se tenga que repetir código para las acciones que se tenían que autenticar (filtro de autenticación), y en general, para todos los métodos de los controladores, obteniendo las excepciones invocadas (filtro de excepción). Este es un punto muy importante ya que en el caso de que algún controlador requiera de una autenticación, o se creen nuevos métodos que requieran del manejo de excepciones, no se tendrá que rehacer este código, ya que, ya se encuentra operativo, y solo se necesita agregar la anotación.
- Patrón Repositorio
 - Como ya se destacó anteriormente, el manejo de las entidades y sus interacciones con la base de datos se realizó de forma genérica, utilizando el patrón repositorio. Esto permite que en caso de que en el futuro se quieran almacenar nuevas entidades, el CRUD ya se encuentre definido, facilitando un agregado de esta índole.

Descripción del mecanismo de acceso a datos utilizado

Con el fin de poder almacenar los datos, se usó Entity Framework Core aplicando Code First para crear la base de datos relacional en la que se persistirán los objetos del dominio.

Al igual que ocurre con el Framework que existe para .NET Standard. Entity Framework Core trabaja de formas distintas a la hora de su aplicación. Las metodologías a seguir podrían ser, Data Annotations, o Fluent API.

En el caso de esta entrega, para definir las claves principales de las tablas y las relaciones entre estas, además de el comportamiento en general de la base de datos, se usó Fluent API, ya que permite la centralización de la lógica de base de datos en un punto solo, y no se tiene un enfoque distribuido como ocurre con las Data Annotations.

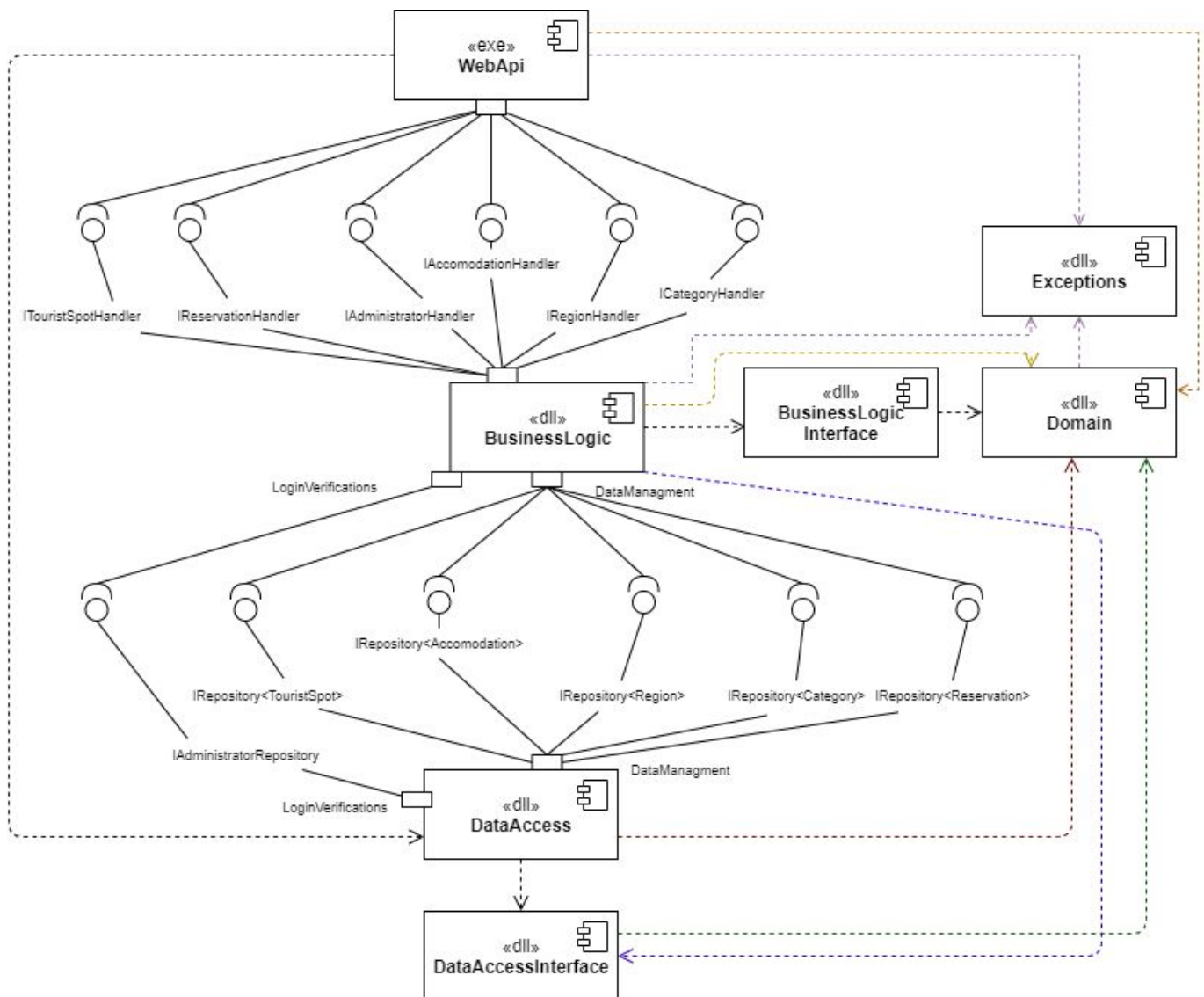
Para la relación many to many entre los puntos turísticos y las categorías fue necesario crear una tabla auxiliar con su respectiva clase en el dominio. Cada objeto de dicha clase tiene una referencia a cada uno y sus respectivos ids, sirviendo como asociación entre ambos.

Como se detalló en la sección Paquetes y sus responsabilidades, y en particular, la sección [“Data Access”](#), para el acceso a la base de datos se usó una interfaz genérica repositorio para cada tipo de objeto a persistir, con la excepción de los administradores que su repositorio requirió que se agreguen funciones a la interfaz mediante una herencia.

Esta idea se implementó, como ya se había establecido, siguiendo el patrón repositorio ya mencionado.

Diagrama de Implementación

El diagrama de implementación muestra los componentes que tiene el proyecto a la hora de ser desplegado en un diagrama externo. Este diagrama muestra la interacción de los componentes y las interfaces que expone y requiere cada uno de ellos.



Se deben destacar del diagrama anterior, las interfaces expuestas y requeridas, que en tiempo de ejecución, se dan por medio de los paquetes Web API, Business Logic, y Data Access. De los cuales, Data Access, y Business Logic, son los que implementan las interfaces de Data Access interface, y Business Logic interface respectivamente. Se destaca además, que las relaciones de dependencia, sin ser respecto de las interfaces, son iguales que las expuestas en el diagrama de paquetes, que se encuentra al comienzo del análisis de diseño.

Evidencia del Diseño y especificación de la API

Para el diseño de la API se siguieron las restricciones definidas para las REST API.

Restricciones de REST

REST, es un estilo arquitectónico para las API, que permite APIs fluidas, livianas y muy performantes. Este estilo arquitectónico está definido por ciertas restricciones, entre las que se destacan las que fueron aplicadas más claramente en este proyecto:

- El server no guarda estados, todas las peticiones son independientes entre sí y contienen toda la información necesaria para realizarse.
 - Un ejemplo de la aplicación de esta restricción, es que siempre que se requiera autenticación para realizar una acción, como son el agregado de puntos turísticos o AVM de administradores, es necesario enviar un token en el header.
Esto se hizo debido a que, como no hay guardado de estados, no se registra a un administrador como activo o logueado, sino que cada vez que quiera hacer una acción de las mencionadas, se debe autenticar.
- Arquitectura cliente-servidor, ambos son independientes y solo dependen de la interfaz definida para comunicarse entre ellos.
 - Para el desarrollo de esta aplicación, esto es evidente ya para el despliegue de esta versión, llegado el deadline establecido, se desarrolló únicamente el servidor sin importar la implementación futura del cliente.
- Finalmente, aunque no es el caso actual, la API permite una arquitectura por capas de servidores invisible para el cliente cumpliendo con la restricción de ser un sistema en capas.
 - Si bien para esta versión, no se desarrolló siguiendo esta idea, se destaca que dado el desarrollo e implementación de la misma, se podría dar una extensión que lo permitiera de manera sencilla.
- El último criterio de REST a trabajar es el de interfaz uniforme.
 - Esta restricción fue implementada para la API desarrollada, siguiendo las siguientes características:
 - Un recurso debe tener una sola URI y un método para obtener información relacionada o adicional.
 - Un recurso no debe ser muy grande, sino que debe tener métodos para obtener toda la información en caso de quererse.
 - Se debe seguir estándares de nombramiento, formato de links y formato de datos.
 - La aplicación de estos estándares se ven expuestos a continuación.

Nombramiento de las URIs

Como se destacó, uno de los puntos más importantes para el uso de la API, es el correcto nombramiento de las URIs, ya que contribuye a las ventajas expuestas por REST, y permite el cumplimiento de la característica de interfaz uniforme.

Las características tomadas en cuenta para esto fueron las siguientes:

En primera instancia, se destaca el uso de sustantivos en plural para representar las colecciones.

Continuando, se estableció que todas las colecciones se encuentran al mismo nivel, ya que, aunque exista dependencia entre ellas la alternativa propuesta hubiese generado demasiados niveles en las colecciones.

Ej: *region/id/spots/id/accommodations/id/reservations/id*.

Esto hubiera dificultado en gran nivel el uso de la API y el entendimiento de la misma.

Esta implementación también presenta la ventaja de que al hacer cada colección al mismo nivel queda claro que el id es único dentro de la colección, mientras que con una colección anidada se podría dar a entender que el id es único dentro del recurso que la contiene.

Esta implementación también cumple con la característica de interfaz uniforme. Ya que siempre que se quiera acceder a una lista de recursos particular (por ejemplo los puntos turísticos de una región) se usa un GET con parámetros para filtrar la colección, y no se usa una vez GET, otra vez FETCH, etc.

Otra característica a destacar de esto, es que, a excepción del login y logout de administradores que tienen su propia uri, todas las funcionalidades que ofrece la api están identificadas por la acción de la request y el recurso al que se le aplica. Si bien se sabe que no se debían seguir los estándares de este nombramiento marcados por HTTP, se consideró como una buena opción para esta aplicación, por lo que se utilizaron:

- POST sobre una colección para crear un nuevo elemento
- PUT sobre un elemento de una colección para actualizarlo
- DELETE sobre un elemento de una colección para borrarlo
- GET sobre una colección o elemento para obtener sus datos.

Mecanismo de autenticación

La autenticación fue un mecanismo necesario para ciertas acciones que debía realizar la aplicación.

Para las funcionalidades específicas de administrador se espera recibir en el header un token con un string identificador de un usuario logueado. En caso de no recibir un token o recibir un token inválido se retorna un error 401.

El token se obtiene en el body de la respuesta de un login exitoso y queda habilitado hasta que se hace un logout con dicho token en el header. Repetir un login generará un nuevo token deshabilitando el anterior.

Este mecanismo se implementó mediante un filtro de autorización, que permitió no repetir código en las acciones que se marcaban como que necesitaban una autenticación. Además de poder filtrar una consulta antes de que llegue al controlador, y en caso de que no trajera datos correctos. En este caso, los de autenticación, permitiría que no llegara al mismo.

Códigos de error y mensajes retornados

Los códigos de error utilizados en la API, siguen también los estándares HTTP, ya que es un estándar universal y entendible por todo el mundo. Dado que se intentó no sobrecargar a la API con mensajes de error innecesarios y que fueran tan específicos que pasarían a ser inentendibles, se utilizaron 5 códigos de retorno:

- 200: se usa para datos o acciones ejecutadas correctamente
- 400: se usa para datos inválidos. Aplica para datos inválidos en body o parámetros, no enviar body y errores de tipo en el body, parámetros o identificadores de uri
- 401: cuando se requiere una autenticación para realizar la acción pero no se incluye un token válido en el header.
- 404: identificadores en la uri que no se corresponden con un elemento de la colección.
- 500: errores internos del servidor. Aplica para cualquier excepción no controlada que no forma parte del paquete de excepciones.

Además, se siguieron las recomendaciones del pasaje de contexto y referencia junto con los errores y mensajes expuestos al cliente. Este contexto, determina el correcto funcionamiento de la aplicación, porque se generó el error, y mensajes que permiten identificarlo, y no volverlo a cometer.

Descripción de los recursos

Otras de las características de las API REST es que se tienen documentaciones concretas y específicas sobre el funcionamiento de la misma.

Para facilitar la lectura de la documentación se usó la herramienta swagger para crear una documentación electrónica. Dicha documentación se encuentra en `swagger/index.html` desde la raíz de la API, Ej: <https://localhost:44313/swagger/index.html>

Swagger genera automáticamente una documentación interactiva que da una descripción de los los diferentes endpoints con sus parámetros y bodies y permite probarlos. A continuación se agrega para cada recurso la descripción, sus headers y los posibles códigos de estado; ya que estos no están contemplados en la documentación de swagger.

Accommodations

Representa la colección de los hospedajes.

POST	/accommodations
Headers necesarios	token
Responses	<ul style="list-style-type: none">• 200 - Accommodation added• 400 - The accommodation needs a non empty Name• 400 - The Accommodation stars needs to be between 1 and• 400 - The accommodation needs a non empty address• 400 - The accommodation needs at least one image• 400 - The Accommodation fee needs to be more than 0• 400 - The image needs to be non empty• 400 - The tourist spot does not exist• 401 - You must be logged as an administrator to access

GET	/accommodations
Headers necesarios	
Responses	<ul style="list-style-type: none">• 200 - Devuelve el objeto en el body• 404 - The accommodation does not exist

PUT	/accommodations/{id}
Headers necesarios	token
Responses	<ul style="list-style-type: none">• 200 - Availability changed• 401 - You must be logged as an administrator to access• 404 - The accommodation does not exist

DELETE	/accommodations/{id}
Headers necesarios	token
Responses	<ul style="list-style-type: none">• 200 - Accommodation deleted• 401 - You must be logged as an administrator to access• 404 - The accommodation does not exist

GET	/accommodations/{id}
Headers necesarios	
Responses	<ul style="list-style-type: none">• 200 - Devuelve el objeto en el body• 404 - The accommodation does not exist

Administrators

Representa la colección de los administradores, también tiene las acciones de logout y login.

POST	/administrators
Headers necesarios	token
Responses	<ul style="list-style-type: none">• 200 - Administrator updated• 400 - The mail already exists• 400 - The administrator needs a non empty Name• 400 - The administrator needs a non empty Email• 400 - The administrator needs a non empty Password• 401 - You must be logged as an administrator to access

GET	/administrators
Headers necesarios	token
Responses	<ul style="list-style-type: none">• 200 - Devuelve la lista en el body• 401 - You must be logged as an administrator to access

POST	/administrators/login
Headers necesarios	
Responses	<ul style="list-style-type: none">• 200 - Devuelve el token en el body• 400 - Wrong email or password

DELETE	/administrators/logout
Headers necesarios	token
Responses	<ul style="list-style-type: none">• 200 - Logged out successfully• 401 - You must be logged as an administrator to access

GET	/administrators/{id}
Headers necesarios	token
Responses	<ul style="list-style-type: none">• 200 - Devuelve el objeto en el body• 401 - You must be logged as an administrator to access• 404 - There is no administrator with that id

DELETE	/administrators/{id}
Headers necesarios	token
Responses	<ul style="list-style-type: none">• 200 - Administrator deleted

	<ul style="list-style-type: none"> • 401 - You must be logged as an administrator to access • 404 - There is no administrator with that id
--	--

PUT	/administrators/{id}
Headers necesarios	token
Responses	<ul style="list-style-type: none"> • 200 - Administrator updated • 400 - The administrator needs a non empty Name • 400 - The administrator needs a non empty Email • 400 - The administrator needs a non empty Password • 401 - You must be logged as an administrator to access • 404 - There is no administrator with that id

Categories

Representa la colección de las categorías.

GET	/categories
Headers necesarios	
Responses	<ul style="list-style-type: none"> • 200 - Devuelve la lista en el body

Regions

Representa la colección de las regiones.

GET	/regions
Headers necesarios	
Responses	<ul style="list-style-type: none"> • 200 - Devuelve la lista en el body

Reservations

Representa la colección de las reservas.

POST	/reservations
Headers necesarios	
Responses	<ul style="list-style-type: none">• 200 - Devuelve el número de reserva en el body• 400 - There is no accomodation with that id• 400 - The Check In Date needs to be after today• 400 - The Check Out Date needs to be after Check In• 400 - The baby quantity must be 0 or more• 400 - The children quantity must be 0 or more• 400 - The reservation needs at least one adult guest• 400 - The reservation needs a non empty name• 400 - The reservation needs a non empty email• 400 - The reservation needs a non empty surname• 400 - Invalid state

GET	/reservations
Headers necesarios	
Responses	<ul style="list-style-type: none">• 200 - Devuelve la lista en el body• 400 - The accomodation does not exist

GET	/reservations/{id}
Headers necesarios	
Responses	<ul style="list-style-type: none">• 200 - Reservation state updated• 404 - The reservation does not exist

PUT	/reservations/{id}
Headers necesarios	token
Responses	<ul style="list-style-type: none">• 200 - Reservation state updated• 400 - Invalid state• 401 - You must be logged as an administrator to access• 404 - The reservation does not exist

Spots

Representa la colección de los puntos turísticos.

GET	/spots
Headers necesarios	
Responses	<ul style="list-style-type: none">• 200 - Devuelve la lista en el body• 400 - The region does not exist• 400 - A category does not exist

POST	/spots
Headers necesarios	token
Responses	<ul style="list-style-type: none">• 200 - Tourist spot added• 400 - The spot needs at least one category• 400 - The name already exists• 400 - The region does not exist• 400 - The image needs to be non empty• 400 - A category does not exist• 401 - You must be logged as an administrator to access

GET	/spots/{id}
Headers necesarios	
Responses	<ul style="list-style-type: none">• 200 - Devuelve el objeto en el body• 404 - The Tourist Spot does not exist

Evidencia de uso de TDD

Clean Code

Para comenzar se debe establecer que Clean Code, como lo describe Robert Martin en su libro, es lo que permite que el código sea legible, entendible y más fácil de modificar.

Martin expone una lista de recomendaciones que se deben llevar a cabo para que el código sea limpio. Estas recomendaciones se encuentran englobadas según el nivel de especificidad del código, por lo que se encuentran recomendaciones de paquetes, clases, métodos y nombramiento de variables.

Para demostrar el uso de Clean Code y que el código se implementó en base a las recomendaciones dadas, se explorarán estas recomendaciones y se mostrarán capturas de pantalla o evidencias demostrando que fueron seguidas.

Nombramiento

Los nombres tanto de variables, métodos, clases y paquetes es un punto importante a la hora de trabajar con clean code. Se destaca que se siguieron las recomendaciones y los estándares de nombramiento de C# para la denominación de las distintas abstracciones.

Un claro ejemplo de esta aplicación se da en el método Search en la clase tourist spot, que define varias variables y llama a distintos métodos. En este método, se puede ver una correcta denominación de variables y métodos.

```
public List<TouristSpot> Search(List<int> categories = null, int? region = null)
{
    if (categories == null)
    {
        if (region.HasValue && regionRepository.Get(region.Value) == null)
        {
            throw new BadRequestException("The region does not exist");
        }
        return spotsRepository.GetAll(x => ((TouristSpot)x).Region.Id == region).ToList();
    }

    List<TouristSpot> spotsListWithRegion;
    spotsListWithRegion = GetListByRegion(region);

    List<TouristSpot> returningSpotList = new List<TouristSpot>();

    foreach (var spot in spotsListWithRegion)
    {
        var hasAll = HasAllCategories(categories, spot);
        if (hasAll) returningSpotList.Add(spot);
    }

    return returningSpotList;
}
```

Entre las denominaciones se pueden destacar:

- Search para el método
- categories para la lista de categorías
- region para la región
- Get y GetAll para métodos del repositorio
- hasAll para una variables que indica si un punto tiene todas las categorías de una lista, cuyo valor se ve asignado por una función denominada HasAllCategories
- returningSpotList para el retorno del método

Este nombramiento sigue los estándares de C# y las recomendaciones que indica Martin que son adecuadas en su libro. Otros puntos que se siguieron dentro de estas recomendaciones son los siguientes:

- Nombres que revelen la intención como hasAll en el método anterior
- Nombres que coinciden con su inicialización
- Se hicieron distinciones respecto a la función que contienen. Por ejemplo, spotListWithRegion y returningSpotList en el ejemplo anterior.
- Se usaron nombres sencillos y pronunciables
- No se usaron letras o números único para ningún nombre
- No se usó decodificación ni prefijos
- Se evitó que quienes lean el código deban hacer mapas mentales.
 - Si bien no se trabajó con for anidados, si se trabajó con foreach, y se intentó de que el nombre de los items fuera nemotécnico con el fin de que no se deban hacer mapas mentales.
 - Un ejemplo de esto es el de spot, que representa un tourist spot en el ejemplo anterior, y no se le puso item.

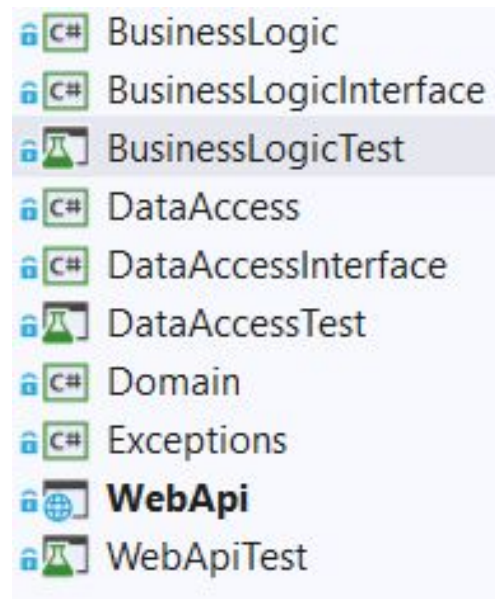
Nombramiento de paquetes, clases y métodos

De manera similar a lo destacado anteriormente, se establece que se utilizaron los estándares de C# y recomendaciones de Martin para el nombramiento de estas abstracciones. Entre las recomendaciones y estándares principales se encuentran:

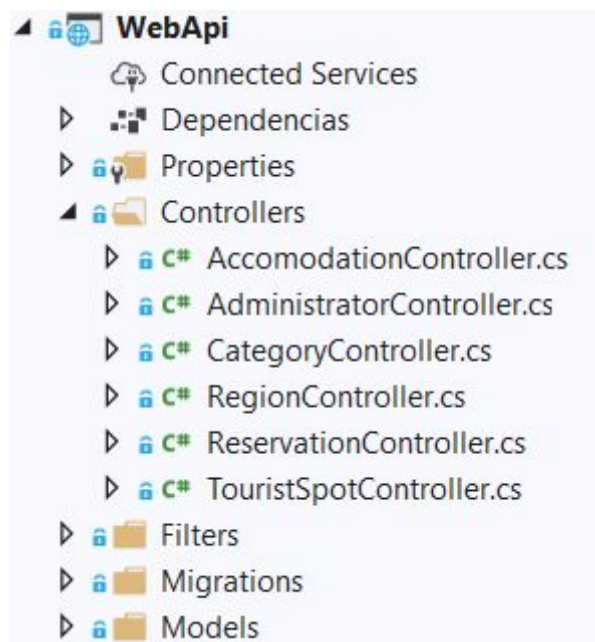
- Paquetes comienzan con mayúscula
- Clases comienzan con mayúscula y no es un verbo
- Métodos siempre utilizan un verbo
- Métodos consistentes entre sí (se usa una sola palabra para traer, en este caso get)

Tomemos por ejemplo el paquete BusinessLogic, que representa a las reglas de negocio, que dentro del mismo se encuentra la clase TouristSpotHandler, que dentro tiene los métodos: Add, Get, Search, HasAllCategories, GenerateCategoriesList y GetListByRegion, lo que sigue los estándares y metodologías recomendadas.

Se agrega a continuación, una captura para demostrar el correcto nombramiento de los paquetes pertenecientes a la solución.



Se muestra también un ejemplo de nombramiento de paquetes que se encuentran dentro de otros y clases dentro de estos últimos con una captura que muestra como se denominaron las abstracciones en el paquete Web API.



Funciones

Las funciones también se implementaron siguiendo los estándares de C# y recomendaciones dadas.

Los puntos a destacar de esto son:

- Nombramiento ya establecido
- Los niveles de indentación no suben de dos
 - Como ejemplo se toma el ya documentado search en el manejador de puntos turísticos. Esta función es la que abarca más niveles de indentación (junto con otras que abarcan la misma cantidad), y abarca hasta dos niveles
- Funciones pequeñas
 - Nuevamente, las funciones más grandes se encuentran en el manejador de puntos turísticos, y dentro de él, la función más grande no supera las 30 líneas de código (función add). Número que es muy bueno considerando que esta es la función más longeva de toda la solución
- Las funciones hacen una sola cosa
- No manejan dos niveles de abstracción. Por ejemplo, no acceden a la base de datos y calculan valores
- El código se puede leer top to bottom. Las funciones que llaman a otras funciones auxiliares se encuentran una detrás de otra. Un ejemplo claro de esto, es nuevamente, la función Search, que tiene las funciones privadas llamadas dentro de ella, debajo de ella. Se muestra una captura a continuación:

```
public List<TouristSpot> Search(List<int> categories = null, int? region = null)
{
    if (categories == null)
    {
        if (region.HasValue && regionRepository.Get(region.Value) == null)
        {
            throw new BadRequestException("The region does not exist");
        }
        return spotsRepository.GetAll(x => ((TouristSpot)x).Region.Id == region).ToList();
    }

    List<TouristSpot> spotsListWithRegion;
    spotsListWithRegion = GetListByRegion(region);

    List<TouristSpot> returningSpotList = new List<TouristSpot>();

    foreach (var spot in spotsListWithRegion)
    {
        var hasAll = HasAllCategories(categories, spot);
        if (hasAll) returningSpotList.Add(spot);
    }

    return returningSpotList;
}
```

1 referencia | Martin, Hace 44 minutos | 1 autor, 1 cambio

```
private bool HasAllCategories(List<int> categories, TouristSpot spot)
```

```

private bool HasAllCategories(List<int> categories, TouristSpot spot)
{
    var hasAll = true;
    foreach (var category in categories)
    {
        if (categoryRepository.Get(category) == null)
        {
            throw new BadRequestException("A category does not exist");
        }
        hasAll = spot.TouristSpotCategories.Any(x => x.CategoryId == category);
        if (!hasAll) break;
    }

    return hasAll;
}

1 referencia | Martin, Hace 47 minutos | 1 autor, 1 cambio
private List<TouristSpot> GetListByRegion(int? region)
{
    List<TouristSpot> spotsList;

    if (region == null)
    {
        spotsList = spotsRepository.GetAll().ToList();
    }
    else
    {
        if (region.HasValue && regionRepository.Get(region.Value) == null)
        {
            throw new BadRequestException("The region does not exist");
        }
        spotsList = spotsRepository.GetAll(x => ((TouristSpot)x).Region.Id == region).ToList();
    }

    return spotsList;
}

```

Otras recomendaciones respetadas

1. Solo se tienen comentarios autogenerados y comentarios que se utilizan para la generación automática de la documentación con la herramienta swagger
2. Formateo
 - a. Se siguió la “metáfora del periódico”. En las clases, primero se definen las variables de instancia, luego los métodos públicos y por último los privados
 - b. Se dejaron espacios verticales y horizontales con el fin de mejorar la legibilidad del código y su entendimiento
3. Manejo de errores
 - a. Se usaron excepciones
 - b. Se pasa un contexto de la excepción, junto con un mensaje descriptivo y autoexplicativo
4. Se implementaron las pruebas unitarias siguiendo la sigla FIRST o ATRIP. Rápidas, independientes, se auto validan, repetibles y en tiempo.

Diseño emergente

Por último, una de las características que Martin expone como importantes y que contribuyen al clean code es el uso o la implementación de la aplicación usando una metodología de diseño emergente. El diseño emergente consta de empezar con un diseño simple e ir generando al crear, correr y refactorizar código a partir de las pruebas, un diseño más complejo, sin duplicaciones y mejorando la calidad del código.

Este diseño, como explica Kent Beck en su libro, se expone en los siguientes pasos:

- Correr todas las pruebas
- Evitar duplicaciones
- Expresar la intención del programador
- Minimizar el número de clases y métodos
- Correr nuevamente las pruebas

Este texto está sacado del libro de la metodología de Beck, TDD, metodología que Martin recomienda y que indica contribuye al clean code. Como se expone en la siguiente sección, TDD o Test Driven Development, es la metodología que se usó para el desarrollo de toda la aplicación, exponiendo resultados positivos, no solo a la cobertura, mantenibilidad, y diseño, sino también al clean code. Ya que al seguir esta metodología, que consta de los estados Red, Green, Refactor, se pudieron llevar a cabo procesos que mejoraban continuamente la claridad del código. Entre ellos, la extracción de métodos, extracción de clases, renombramiento, uso de polimorfismo, etc.

Un ejemplo claro de este uso es el del filtro de autorización en autor, que permitió no repetir código en el controlador de administradores y que fue identificado en uno de los pasos de refactor luego de hacer varias pruebas para los métodos de este controlador.

Lo mismo ocurrió con el filtro de excepciones, que permitió no tener que hacer un try-catch-finally cada vez que se obtenía una excepción, sino que se englobó el comportamiento en este filtro. Nuevamente, este refactor se hizo a partir de la repetición de código que se generaba luego de escribir las pruebas utilizando esta metodología, dando paso nuevamente, a un nuevo arreglo que contribuía al entendimiento del código.

Como se vió, esta metodología contribuyó, además de a todas las ventajas que se expondrán a continuación, al clean code de la aplicación.

Desarrollo con TDD (Test Driven Development)

El desarrollo de la aplicación se realizó de forma íntegra utilizando la metodología TDD o desarrollo guiado por las pruebas. Por definición, esta metodología de desarrollo consta de tres estados, red, green y refactor, en los que el estado red indica la construcción de la prueba, pero sin éxito por el momento, green, indica que se hizo el código necesario para que la prueba sea correcta (utilizando un código lo más simple posible) y refactor, que es el paso en el que se mejora el diseño que se tiene de la aplicación, evitando la duplicación de código y mejorando el nivel de claridad del código, lo que como ya se estableció, contribuye al Clean Code.

Las ventajas que se pudieron y se pueden ver gracias a la aplicación de esta metodología son las siguientes:

- Permite introducir cambios asegurando que la funcionalidad no se ve modificada
- Contribuye al Clean Code
- Permite programar de una manera más segura y más rápida en cuanto al agregado de funcionalidades y código
- Contribuye al código con cobertura alta, lo que, a su vez, evita que se generen bugs inesperados o no contemplados.

Para el caso del desarrollo de este proyecto, se pudieron ver las ventajas mencionadas de la aplicación de esta metodología.

Metodología seguida (Outside In)

La aplicación de esta metodología en este proyecto fue siguiendo la metodología Outside In. Esta metodología consta de probar parte por partes la aplicación, utilizando ciertos elementos que no pertenecen al dominio para facilitar la prueba y generar un comportamiento y/o un estado esperado. Estos objetos son los test doubles y para el caso de este proyecto, se utilizaron Mocks, que son un tipo de test double que permite setear el comportamiento esperado frente a la invocación de cierta función.

Dada la aplicación de la inyección de dependencias, el enfoque que se debía seguir para con esta metodología debía ser el especificado (Outside In), ya que se quería poder trabajar con una aplicación que podría evolucionar de forma independiente (cada capa), y no depender de toda una implementación para poder probar el comportamiento del programa. Por lo que el método Inside Out no era adecuado ya que el mismo consta de probar la aplicación por funcionalidad, abarcando todos los niveles de abstracción que puede implicar, y sin realizar ningún mock ni test double.

La aplicación de esta metodología (Outside In) tiene sus ventajas y sus desventajas, los beneficios son los ya mencionados, evolución independiente, que no se necesiten todas las clases para probar los módulos, entre otros. Pero la principal desventaja es que se debe saber de antemano cuales son los métodos expuestos por cada uno de los módulos que se comunican con el módulo probado, lo que requiere interfaces bien definidas para ello.

Además, otra desventaja importante es que las pruebas de aceptación deben ser completas y correctas, debido a que en caso contrario, solo se estaría probando el comportamiento de cada módulo por separado y no el conjunto.

Dadas estas desventajas, se intentaron construir interfaces bien definidas para la comunicación entre módulos, y se tiene un set de pruebas de integración extenso y completo (colección en Postman).

Demostración de aplicación de la metodología

Para demostrar el uso de la misma, se mostrará evidencia de ciertos commits en los que se demostraba la aplicación de esta metodología, además de capturas de pantalla que muestran la cobertura de código en general, y para las funcionalidades principales de la aplicación en particular.

Para comenzar, se destaca la construcción y correcta ejecución de las pruebas. La solución consta de 3 proyectos de prueba, uno para cada capa de la aplicación. Como ya se destacó en la sección de Clean Code, las mismas se realizaron siguiendo la sigla ATRIP o FIRST.

Para cada uno de los paquetes de prueba, se generaron clases de prueba con el fin de chequear las funcionalidades de las distintas clases y claramente, dentro de ellas se tenían distintos métodos.

Dada la metodología Outside In, se destaca que en el paquete de pruebas de la Web API, se realizaron Mocks que especificaban el comportamiento de las interfaces que exponía la capa de negocio, y asimismo, las pruebas de la capa de negocio, tenían mocks que simulaban el comportamiento de las clases e interfaces expuestas por el repositorio. Se muestra lo dicho en las siguientes capturas:

Pruebas de la capa de negocio

```
[TestMethod]
[ExpectedException(typeof(BadRequestException),
"The Accomodation fee needs to be more than 0")]
public void AddAccomodationWith0Fee()
{
    touristSpotMock.Setup(x => x.Get(touristSpot.Id)).Returns(touristSpot);
    accomodationMock.Setup(x => x.Add(accomodation)).Returns(accomodation);

    accomodation.Fee = 0;
    var res = handler.Add(accomodation, touristSpot.Id, imageNames);
}
```

Pruebas de la capa de presentación (Web API)

```
public void GetAccommodationByIdOk()
{
    var mock = new Mock<IAccommodationHandler>(MockBehavior.Strict);
    var controller = new AccommodationController(mock.Object);
    var accommodationModel = new AccommodationModel()
    {
        Name = "Prueba",
        Stars = 3.0,
        Address = "Cuareim",
        ImageNames = new List<string> { "image" },
        Fee = 200,
        Description = "",
        Available = true,
        Telephone = "00000",
        ContactInformation = "",
        TouristSpotId = 1
    };

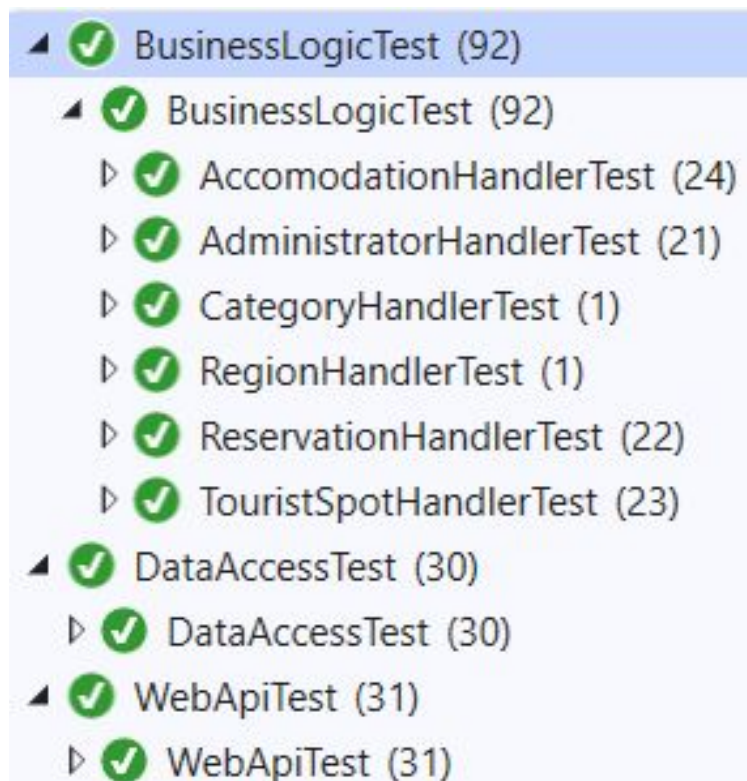
    mock.Setup(x => x.Get(It.IsAny<int>())).Returns(accommodationModel.ToEntity());

    var result = controller.Get(1);
    var okResult = result as OkObjectResult;
    var value = okResult.Value as bool?;

    mock.VerifyAll();
}
```

Ejecución correcta de las pruebas

Las pruebas generadas tienen un 100% de éxito, como se puede ver en la captura a continuación. Se aprecian también las distintas clases generadas para probar la aplicación.



Cobertura de código



El siguiente parámetro a verificar es la cobertura de código.

Al correr el analizador de código para la aplicación se obtiene el siguiente resultado:


Web API

Para la Web API, se obtienen distintos valores de la cobertura:

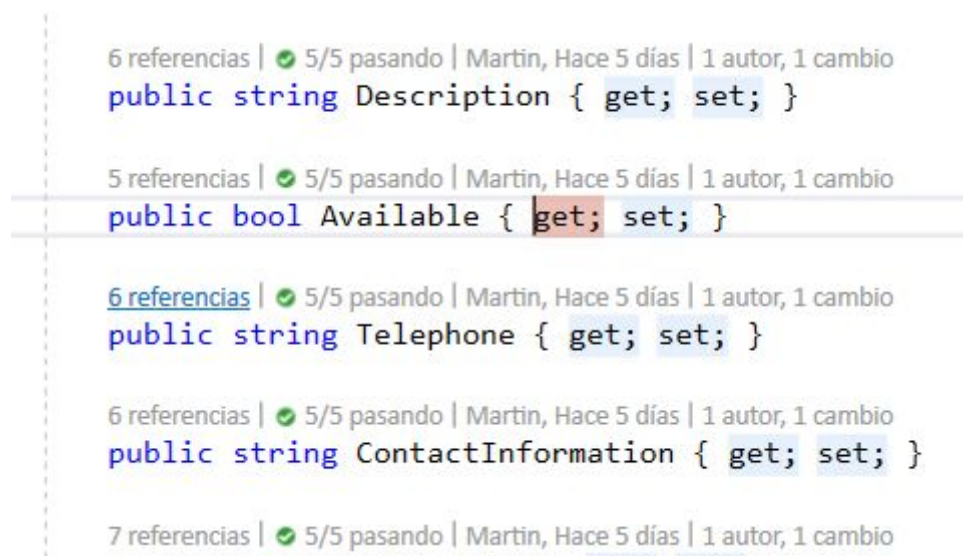
La cobertura de los controladores y de los filtros es del 100% como se puede ver a continuación:

▷  WebApi.Controllers	0	0,00 %	128	100,00 %
▷  WebApi.Filters	0	0,00 %	37	100,00 %

Por otro lado, la cobertura para los modelos no alcanza el 100%.

 WebApi.Models	1	0,83 %	120	99,17 %
---	---	--------	-----	---------

Esta falta en la cobertura reside en el uso de propiedades, y en la falta de pruebas para los métodos set o get de algunas de ellas:



Business Logic

Para la lógica de negocio, la cobertura es de 91,24% como se muestra en la siguiente captura.

businesslogic.dll	29	8,76 %	302	91,24 %
BusinessLogic	29	8,76 %	302	91,24 %

La cobertura en este caso no es del 100% por lo siguiente:

```
public List<TouristSpot> Search(List<int> categories = null, int? region = null)
{
    if (categories == null)
    {
        if (region.HasValue && regionRepository.Get(region.Value) == null)
        {
            throw new BadRequestException("The region does not exist");
        }
        return spotsRepository.GetAll(x => ((TouristSpot)x).Region.Id == region).ToList();
    }
}
```

Uno de los métodos del acceso a datos usado es el GetAll. Como ya se especificó, este método recibe una función lambda por parámetro y hace una consulta filtrando por esa condición.

A la hora de probar este método, la forma de realizar un Mock, es la siguiente:

```
mock.Setup(x => x.GetAll(It.IsAny<Func<object, bool>>())).
    Returns(new List<TouristSpot> { spot, spot2 });

List<TouristSpot> res = handler.Search(new List<int>(){ 1, 2 });

mock.VerifyAll();
Assert.AreEqual(1, res.Count());
Assert.AreEqual(spot2.Name, res[0].Name);
}
```

Dado que el Mock trabaja de esta manera, no se ingresa a la condición que se está efectivamente probando, sino que solo verifica que sea una función lambda, marcando entonces esa línea como no probada, y disminuyendo así la cobertura del código.

Domain

La cobertura de domain es relativamente baja (casi 80%), como se muestra a continuación:

domain.dll	56	20,66 %	215	79,34 %
Domain	56	20,66 %	215	79,34 %
Accomodation	9	16,98 %	44	83,02 %
Administrator	7	17,07 %	34	82,93 %
Category	14	77,78 %	4	22,22 %
Image	0	0,00 %	10	100,00 %
Region	12	75,00 %	4	25,00 %
Reservation	5	6,76 %	69	93,24 %
TouristSpot	8	15,69 %	43	84,31 %
TouristSpotCategory	1	12,50 %	7	87,50 %

Esto se da por algo similar a lo que pasaba con los modelos, en los que los métodos get y set no probados de las properties bajaban la cobertura. En este caso, es similar, pero se ve acentuado por los métodos equals y hashCode que no se han probado. Esto se nota claramente en Category y Region en los que los 14 y 12 bloques respectivamente, que no se prueban, corresponden a los mencionados.

Category	14	77,78 %	4	22,22 %
Equals(object)	8	100,00 %	0	0,00 %
GetHashCode()	4	100,00 %	0	0,00 %

Data Access

La cobertura de Data Access es de 97%. Nuevamente, ocurre algo similar a model y a domain, en los que la bajada de cobertura se da debido a los métodos get de las properties que no son probados de forma automática.

dataaccess.dll	7	2,83 %	240	97,17 %
DataAccess	7	2,83 %	240	97,17 %
AdministratorRepo...	0	0,00 %	23	100,00 %
AdministratorRepo...	0	0,00 %	8	100,00 %
AdministratorRepo...	0	0,00 %	3	100,00 %
Repository.<>c_D...	0	0,00 %	2	100,00 %
Repository<T>	0	0,00 %	28	100,00 %
TourismContext	7	3,83 %	176	96,17 %

Funcionalidades “importantes”

Para las funcionalidades destacadas como importantes, se analiza su cobertura de código y los commits que validan el uso de la metodología TDD. Para cada una de ellas, se agrega la captura de código y su cobertura (mostrada en celeste). La evidencia de los commits se puede encontrar en el repositorio, cuyo link se encuentra al principio del documento. En general, se destaca que para el acceso a base de datos, la cobertura es de un 100%, y al ser genérico, aplica para todas las funcionalidades a estudiar. Esto queda comprobado en la siguiente captura:

```
public T Add(T entity)
{
    DbSet.Add(entity);
    Save();
    return entity;
}

13 referencias | 7/7 pasando | Santiago Rugriza, Hace 11 días | 1 autor, 4 cambios
public bool Delete(T entity)
{
    DbSet.Remove(entity);
    Save();
    return true;
}

3 referencias | Santiago Rugriza, Hace 11 días | 1 autor, 4 cambios
private void Save()
{
    context.SaveChanges();
}

66 referencias | 46/46 pasando | Santiago Rugriza, Hace 6 días | 1 autor, 3 cambios
public IEnumerable<T> GetAll(Func<object, bool> p = null)
{
    if (p == null) return DbSet.ToList();

    return DbSet.ToList().Where(x => p.Invoke(x));
}

95 referencias | 71/71 pasando | Santiago Rugriza, Hace 13 días | 1 autor, 2 cambios
public T Get(int id)
{
    return DbSet.Find(id);
}

17 referencias | 15/15 pasando | Santiago Rugriza, Hace 11 días | 1 autor, 3 cambios
public bool Update(T entity)
{
    DbSet.Update(entity);
    Save();
    return true;
}

public Administrator Find(string email, string password)
{
    var admins = DbSet.ToList().Where(admin => admin.Email == email
    && admin.Password == password);

    if (admins.Count() == 0)
    {
        return null;
    }

    return admins.ToArray()[0];
}

10 referencias | 7/7 pasando | Martin, Hace 7 días | 1 autor, 1 cambio
public Administrator Find(string token)
{
    if (token == null)
    {
        return null;
    }

    var admins = DbSet.ToList().Where(admin => admin.Token == token);

    if (admins.Count() == 0)
    {
        return null;
    }

    return admins.ToArray()[0];
}
```

También se muestra una una captura de la cobertura de los filtros, que aplican para ciertas funcionalidades de las principales:

```
public void OnAuthorization(AuthorizationFilterContext context)
{
    string token = context.HttpContext.Request.Headers["token"];
    if (token == null)
    {
        context.Result = new ContentResult()
        {
            StatusCode = 401,
            Content = "You must be logged as an administrator to access."
        };
        return;
    }
    if (!handler.IsLogged(token))
    {
        context.Result = new ContentResult()
        {
            StatusCode = 401,
            Content = "You must be logged as an administrator to access."
        };
        return;
    }
}

public void OnException(ExceptionContext context)
{
    int code = 500;
    if (context.Exception is NotFoundException) code = 404;
    else if (context.Exception is BadRequestException) code = 400;

    context.Result = new ContentResult()
    {
        StatusCode = code,
        Content = context.Exception.Message
    };
}
```

Búsqueda de hospedajes para cierto punto turístico

```
public List<Accommodation> SearchByTouristSpot(int spotId)
{
    if (touristSpotHandler.Get(spotId) == null)
    {
        throw new BadRequestException("The spot does not exist");
    }

    return accommodationRepository.GetAll(x => ((Accommodation)x).TouristSpot.Id == spotId &&
        ((Accommodation)x).Available).ToList();
}

[HttpGet]
1 referencia | 1/1 pasando | Santiago Rugniz, Hace 2 días | 1 autor, 1 cambio
public IActionResult GetByTouristSpot(int spotId)
{
    return Ok(handler.SearchByTouristSpot(spotId));
}
```

Realizar una reserva de hospedaje

```
public Reservation Add(Reservation reservation, int accommodationId)
{
    var gotAccommodation = accommodationHandler.Get(accommodationId);
    if (gotAccommodation != null)
    {
        reservation.Accommodation = gotAccommodation;
        reservation.Total = CalculateTotal(reservation);
        return repository.Add(reservation);
    }
    else
    {
        throw new BadRequestException("There is no accommodation with that id");
    }
}

1 referencia | Santiago Rugniz, Hace 4 días | 1 autor, 1 cambio
private double CalculateTotal(Reservation reservation)
{
    double ret = 0;
    int days = (reservation.CheckOut - reservation.CheckIn).Days;
    ret += reservation.AdultQuantity + reservation.ChildrenQuantity*0.5 + reservation.BabyQuantity*0.25;
    ret *= days * reservation.Accommodation.Fee;
    return ret;
}

[HttpPost]
1 referencia | 1/1 pasando | Santiago Rugniz, Hace 1 día | 2 autores, 4 cambios
public IActionResult Post([FromBody] ReservationModel reservation)
{
    var res = handler.Add(reservation.ToEntity(), reservation.AccommodationId);
    return Ok("Reservation created, reservation number: " + res.Id);
}
```

Dar de alta un nuevo hospedaje

```
public TouristSpot Add(TouristSpot spot, int regionId, List<int> categoryIds, string imageName)
{
    if (categoryIds.Count == 0) throw new BadRequestException("The spot needs at least one category");

    if (spotsRepository.GetAll(x => ((TouristSpot)x).Name == spot.Name).
        ToList().Count() > 0)
    {
        throw new BadRequestException("The name already exists");
    }

    var gotRegion = regionRepository.Get(regionId);
    if (gotRegion == null)
    {
        throw new BadRequestException("The region does not exist");
    }

    spot.Region = gotRegion;
    spot.Image = new Image { Name = imageName };

    List<TouristSpotCategory> gotCategories = GenerateCategoriesList(spot, categoryIds);

    var addingResult = spotsRepository.Add(spot);

    foreach (var category in gotCategories)
    {
        category.TouristSpotId = spot.Id;
        joinedRepository.Add(category);
    }

    return addingResult;
}

[ServiceFilter(typeof(AuthorizationFilter))]
[HttpPost]
1 referencia | 1/1 pasando | Santiago Rugniz, Hace 3 días | 2 autores, 6 cambios
public IActionResult Post([FromBody] TouristSpotModel spot)
{
    handler.Add(spot.ToEntity(), spot.RegionId, spot.CategoryIds, spot.Image);
    return Ok("Tourist spot added");
}
```

Borrar un hospedaje

```
2 referencias | 1/1 pasando | Santiago Rognitz, Hace 13 días | 2 autores, 3 cambios  
public bool Delete(int id)  
{  
    var accomodation = Get(id);  
    if (accomodation == null) throw new NotFoundException("There is no accomodation with that id");  
  
    return accomodationRepository.Delete(accomodation);  
}
```

```
[ServiceFilter(typeof(AuthorizationFilter))]  
[HttpDelete("{id}")]  
1 referencia | 1/1 pasando | Santiago Rognitz, Hace 2 días | 1 autor, 3 cambios  
public IActionResult Delete( int id)  
{  
    handler.Delete(id);  
    return Ok("Accomodation deleted");  
}
```

Modificar la capacidad actual de un hospedaje

```
public bool ChangeAvailability(int id, bool availability)  
{  
    var accomodation = Get(id);  
    if (accomodation == null) throw new NotFoundException("There is no accomodation with that id");  
  
    accomodation.Available = availability;  
    return accomodationRepository.Update(accomodation);  
}
```

```
[ServiceFilter(typeof(AuthorizationFilter))]  
[HttpPut("{id}")]  
1 referencia | 1/1 pasando | Santiago Rognitz, Hace 2 días | 1 autor, 5 cambios  
public IActionResult ChangeAvailability( int id, [FromBody] bool available)  
{  
    handler.ChangeAvailability(id, available);  
    return Ok("Availability changed");  
}
```

Cambiar el estado de una reserva

```
public bool ChangeState(int idReservation, ReservationState state, string description)  
{  
    Reservation reservation = repository.Get(idReservation);  
    if (reservation == null)  
    {  
        throw new NotFoundException("The reservation does not exists");  
    }  
    reservation.ReservationState = state;  
    reservation.StateDescription = description;  
    return repository.Update(reservation);  
}
```

```
[ServiceFilter(typeof(AuthorizationFilter))]  
[HttpPut("{id}")]  
1 referencia | 1/1 pasando | Santiago Rognitz, Hace 1 día | 1 autor, 2 cambios  
public IActionResult ChangeState(int id, [FromBody] ReservationChangeModel change)  
{  
    handler.ChangeState(id, change.State, change.Description);  
    return Ok("Reservation state updated");  
}
```

Se aprecia como para todas las funcionalidades marcadas, el código está en celeste a excepción de las llamadas a GetAll, de los repositorios, que como ya se estableció, no se ha probado.

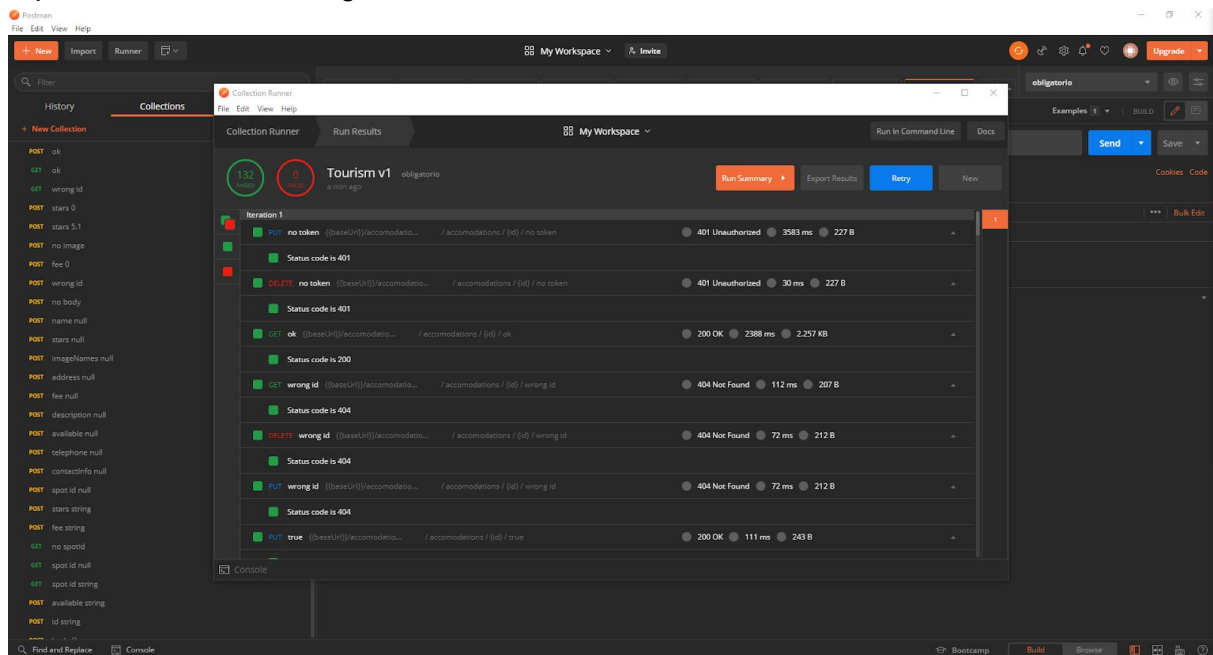
Reporte de casos de prueba con Postman

En la raíz del repositorio se encuentra un json para importar la colección de Postman. La colección se probó con el siguiente environment:

- baseUrl: https://localhost:44313
- token: fe2d952c-09a0-4d5a-ba42-178adeedc160
- logoutToken: 18d6d118-4508-4590-b0e5-30e5aba38c2e

las variables token se corresponden con tokens de administradores ya logueados en los datos de prueba.

para cada request de la colección se hizo un test para probar que se obtiene el status code esperado obteniendo el siguiente resultado:



Se agregan a continuación, los casos de prueba y escenarios para las cuales se probaron los distintos métodos de la aplicación. Para esta análisis, se muestran los casos de prueba de las funcionalidades que se consideraron como principales.

Búsqueda de hospedajes

Escenario	Nombre
Escenario 1	Id correcto
Escenario 2	Id inválido

Caso de prueba	Escenario	spotId	Resultado esperado
CP 1	Esc 1	V	200 - se recibe la lista
CP 2	Esc 2	NV	400 - Mensaje de error

Entrada	Clases válidas	Clases no válidas
spotId	Valor numérico correspondiente a un id de un punto turístico del sistema	Valores numéricos que no se corresponden con un punto, strings y null

CP	spotId	Resultado esperado	Resultado obtenido
1.1	1	200	OK
2.1	0	400	OK
2.2		400	OK
2.3	null	400	OK
2.4	string	400	OK

Realizar una reserva de un hospedaje

Escenario	Nombre
Escenario 1	modelo correcto
Escenario 2	modelo incorrecto o nulo

Entrada	Clases válidas	Clases no válidas
body	modelo válido	modelo invalido o vacío
name	string no vacío	null
adultQuantity	entero positivo	entero no positivo, null, no entero
childrenQuantity	entero no negativo	entero negativo, null, no entero
babyQuantity	entero no negativo	entero negativo, null, no entero
email	string no vacío	null
surname	string no vacío	null
reservationState	entero o string correspondiente a un elemento del enum	state fuera de rango, null, string invalido
telephone	string	
stateDescription	string	
checkIn	string con formato correcto	null, string mal formado
checkOut	string con formato correcto	null, string mal formado
accomodationId	entero correspondiente a una accomodation	null, no entero, entero que no se corresponde a una accomodation

Atributos del body que pueden ser invalidas:

1. name
2. adultquantity
3. childrenquantity
4. babyquantity
5. email
6. surname
7. reservationState
8. checkin

9. checkOut

10. accomodationId

Caso de prueba	Escenario	body	Atributos										Resultado esperado
			1	2	3	4	5	6	7	8	9	10	
CP 1	Esc 1	V	V	V	V	V	V	V	V	V	V	V	200 - se agrega la reserva
CP 2	Esc 2	NV	-	-	-	-	-	-	-	-	-	-	400 - Mensaje de error
CP 3	Esc 2	V	NV	V	V	V	V	V	V	V	V	V	400 - Mensaje de error
CP 4	Esc 2	V	V	NV	V	V	V	V	V	V	V	V	400 - Mensaje de error
CP 5	Esc 2	V	V	V	NV	V	V	V	V	V	V	V	400 - Mensaje de error
CP 6	Esc 2	V	V	V	V	NV	V	V	V	V	V	V	400 - Mensaje de error
CP 7	Esc 2	V	V	V	V	V	NV	V	V	V	V	V	400 - Mensaje de error
CP 8	Esc 2	V	V	V	V	V	V	NV	V	V	V	V	400 - Mensaje de error
CP 9	Esc 2	V	V	V	V	V	V	V	NV	V	V	V	400 - Mensaje de error
CP 10	Esc 2	V	V	V	V	V	V	V	V	NV	V	V	400 - Mensaje de error
CP 11	Esc 2	V	V	V	V	V	V	V	V	V	NV	V	400 - Mensaje de error
CP 12	Esc 2	V	V	V	V	V	V	V	V	V	V	NV	400 - Mensaje de error

(para las entradas no especificadas se usaron valores válidos)

CP	Entrada	Valor	Resultado esperado	Resultado obtenido
1.1	body	modelo	200	OK
1.2	name	string	200	OK
1.3	adultQuantity	1	200	OK
1.4	childrenQuantity	0	200	OK
1.5	babyQuantity	0	200	OK
1.6	email	string	200	OK
1.7	surname	string	200	OK
1.8	reservationState	0	200	OK
1.9	reservationState	Creada	200	OK
1.1 0	checkIn	01-01-2021	200	OK
	checkOut	01-02-2021		
1.1 1	accomodationId	1	200	OK
2.1	body		400	OK
2.2	body	{}	400	OK
3.1	name	null	400	OK
4.1	adultQuantity	0	400	OK
4.2	adultQuantity	null	400	OK
4.3	adultQuantity	string	400	OK
5.1	childrenQuantity	-1	400	OK
5.2	childrenQuantity	null	400	OK
5.3	childrenQuantity	string	400	OK
6.1	babyQuantity	-1	400	OK
6.2	babyQuantity	null	400	OK
6.3	babyQuantity	string	400	OK
7.1	email	null	400	OK
8.1	surname	null	400	OK

9.1	reservationState	-1	400	OK
9.2	reservationState	5	400	OK
9.3	reservationState	null	400	OK
9.4	reservationState	string	400	OK
10.1	checkIn	01-01-2020	400	OK
10.2	checkIn	>checkout	400	OK
10.3	checkIn	null	400	OK
10.4	checkIn	01#01#2021	400	OK
11	checkOut	01#02#2021	400	OK
11.1	checkOut	null	400	OK
12.1	accomodationId	0	400	OK
12.2	accomodationId	nul	400	OK
12.3	accomodationId	string	400	OK

Dar de alta un nuevo hospedaje

Escenario	Nombre
Escenario 1	modelo correcto
Escenario 2	modelo incorrecto o nulo

Entrada	Clases válidas	Clases no válidas
body	modelo válido	modelo invalido o vacío
name	string no vacío	null
address	string no vacío	null
imageNames	lista de strings no vacía	null, lista vacía
fee	double positivo	double no positivo, null, no double
description	string	
available	boolean	null, no boolean
telephone	string	
contactInformation	string	
touristSpotId	entero correspondiente a un punto	null, no entero, entero que no se corresponde a un punto

Caso de prueba	Escenario	body	name	stars	address	image names	fee	available	tourist spot id	Resultado esperado
CP 1	Esc 1	V	V	V	V	V	V	V	V	200 - se agrega el hospedaje
CP 2	Esc 2	NV	-	-	-	-	-	-	-	400 - Mensaje de error
CP 3	Esc 2	V	NV	V	V	V	V	V	V	400 - Mensaje de error
CP 4	Esc 2	V	V	NV	V	V	V	V	V	400 -

										Mensaje de error
CP 5	Esc 2	V	V	V	NV	V	V	V	V	400 - Mensaje de error
CP 6	Esc 2	V	V	V	V	NV	V	V	V	400 - Mensaje de error
CP 7	Esc 2	V	V	V	V	V	NV	V	V	400 - Mensaje de error
CP 8	Esc 2	V	V	V	V	V	V	NV	V	400 - Mensaje de error
CP 9	Esc 2	V	V	V	V	V	V	V	NV	400 - Mensaje de error

CP	body	name	stars	address	image names	fee	available	tourist spot id	Resultado esperado	Resultado obtenido
1.1	si	<string>	1.0	<string>	image1, image2	1.0	true	1	200	OK
2.1	no	-	-	-	-	-	-	-	400	OK
2.2	{ }	-	-	-	-	-	-	-	400	OK
3.1	si	null	1.0	<string>	<string>,<string>	1.0	true	1	400	OK
4.1	si	<string>	0	<string>	<string>,<string>	1.0	true	1	400	OK
4.2	si	<string>	5.1	<string>	<string>,<string>	1.0	true	1	400	OK
4.3	si	<string>	null	<string>	<string>,<string>	1.0	true	1	400	OK
4.4	si	<string>	string	<string>	<string>,<string>	1.0	true	1	400	OK
5.1	si	<string>	1.0	null	<string>,<string>	1.0	true	1	400	OK
6.1	si	<string>	1.0	<string>	null	1.0	true	1	400	OK
6.2	si	<string>	1.0	<string>		1.0	true	1	400	OK
7.1	si	<string>	1.0	<string>	<string>,<string>	0	true	1	400	OK
7.2	si	<string>	1.0	<string>	<string>,<string>	null	true	1	400	OK
7.3	si	<string>	1.0	<string>	<string>,<string>	string	true	1	400	OK

						g				
8.1	si	<string>	1.0	<string>	<string>,<string>	1.0	null	1	400	OK
8.2	si	<string>	1.0	<string>	<string>,<string>	1.0	string	1	400	OK
9.1	si	<string>	1.0	<string>	<string>,<string>	1.0	true	0	400	OK
9.2	si	<string>	1.0	<string>	<string>,<string>	1.0	true	null	400	OK
9.3	si	<string>	1.0	<string>	<string>,<string>	1.0	true	string	400	OK
9.4	si	<string>	1.0	<string>	<string>,<string>	1.0	true	1	400	OK

Dar de baja de un hospedaje

Escenario	Nombre
Escenario 1	id y auth válidos
Escenario 2	auth invalida
Escenario 3	id invalida

Entrada	Clases válidas	Clases no válidas
id	entero correspondiente a un hospedaje	null, no entero, entero que no se corresponde a un hospedaje
token	string correspondiente a un admin	null, string que no se corresponde con un admin

Caso de prueba	Escenario	token	id	Resultado esperado
CP 1	Esc 1	V	V	200 - se agrega el hospedaje
CP 2	Esc 2	NV	V	401 - Mensaje de error
CP 3	Esc 3	V	NV	404/400 - Mensaje de error

CP	token	id	Resultado esperado	Resultado obtenido
1.1	V	2	200	OK
2.1	no token	2	401	OK
2.2	string	2	401	OK
3.1	V	0	404	OK
3.2	V	string	400	OK

Modificar la capacidad actual de un hospedaje

Escenario	Nombre
Escenario 1	id, body y auth válidos
Escenario 2	auth invalida
Escenario 3	id invalida
Escenario 4	body invalido o nulo

Entrada	Clases válidas	Clases no válidas
id	entero correspondiente a un hospedaje	null, no entero, entero que no se corresponde a un hospedaje
body	boolean	null, no boolean
token	string correspondiente a un admin	null, string que no se corresponde con un admin

Caso de prueba	Escenario	token	id	body	Resultado esperado
CP 1	Esc 1	V	V	V	200 - se agrega el hospedaje
CP 2	Esc 2	NV	V	V	401 - Mensaje de error
CP 3	Esc 3	V	NV	V	404/400 - Mensaje de error
CP 4	Esc 4	V	V	NV	400 - Mensaje de error

CP	token	id	body	Resultado esperado	Resultado obtenido
1.1	V	2	true	200	OK
1.2	V	2	false	200	OK
2.1	no token	2	true	401	OK
2.2	string	2	true	401	OK
3.1	V	0	true	404	OK
3.2	V	string	true	400	OK

4.1	V	2	no body	400	OK
4.2	V	2	null	400	OK
4.3	V	2	string	400	OK

Cambiar el estado de una reserva

Escenario	Nombre
Escenario 1	id, body y auth válidos
Escenario 2	auth invalida
Escenario 3	id invalida
Escenario 4	body invalido o nulo

Entrada	Clases válidas	Clases no válidas
id	entero correspondiente a un hospedaje	null, no entero, entero que no se corresponde a un hospedaje
body	state correspondiente a un elemento del enum	state null, int/string que no se corresponde con un elemento del enum
token	string correspondiente a un admin	null, string que no se corresponde con un admin

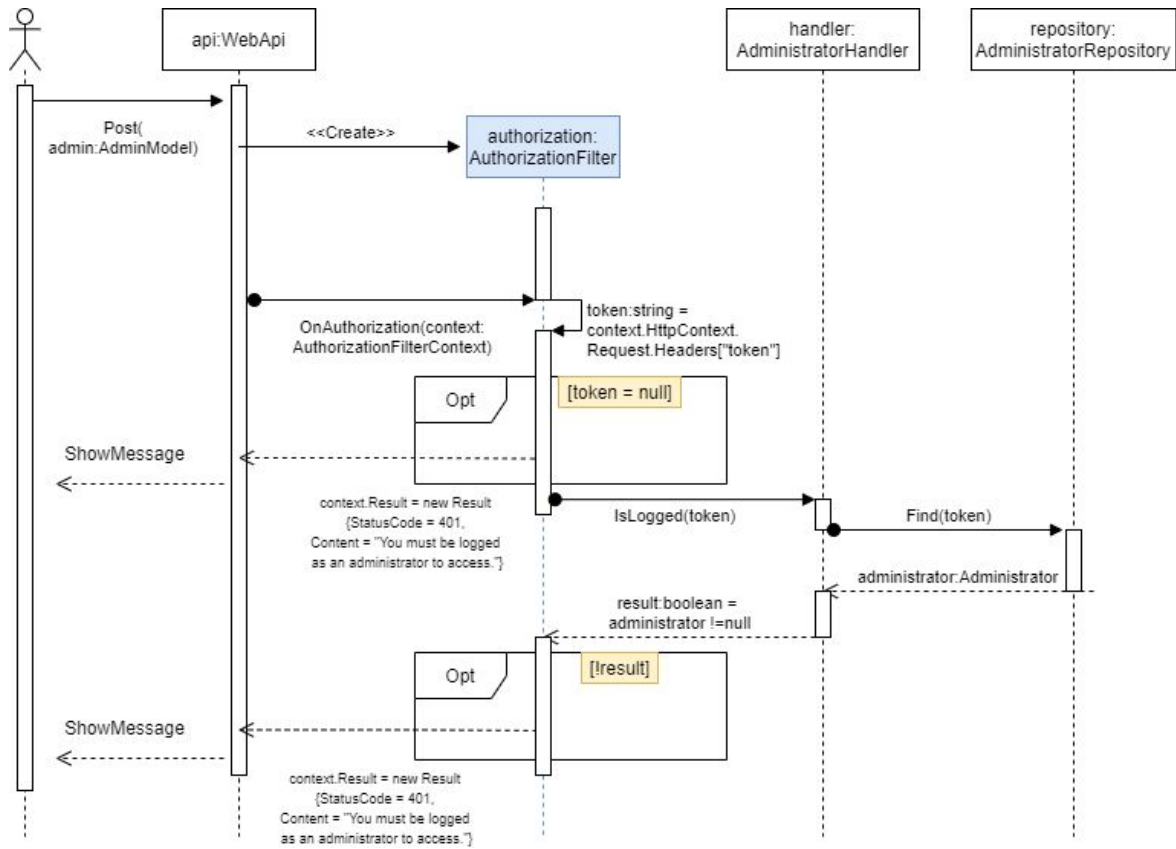
Caso de prueba	Escenario	token	id	body	Resultado esperado
CP 1	Esc 1	V	V	V	200 - se agrega el hospedaje
CP 2	Esc 2	NV	V	V	401 - Mensaje de error
CP 3	Esc 3	V	NV	V	404/400 - Mensaje de error
CP 4	Esc 4	V	V	NV	400 - Mensaje de error

CP	token	id	body	state	description	Resultado esperado	Resultado obtenido
1.1	V	1	si	1	<string>	200	OK
1.2	V	1	si	1	null	200	OK
1.3	V	1	si	Creada	<string>	200	OK
1.4	V	1	{}	0 (default)	null (default)	200	OK
2.1	no token	1	si	1	<string>	401	OK
2.2	string	1	si	1	<string>	401	OK
3.1	V	0	si	1	<string>	404	OK
3.2	V	string	si	1	<string>	400	OK
4.1	V	1	no			400	OK
4.2	V	1	si	-1	<string>	400	OK
4.3	V	1	si	5	<string>	400	OK
4.4	V	1	si	null	<string>	400	OK

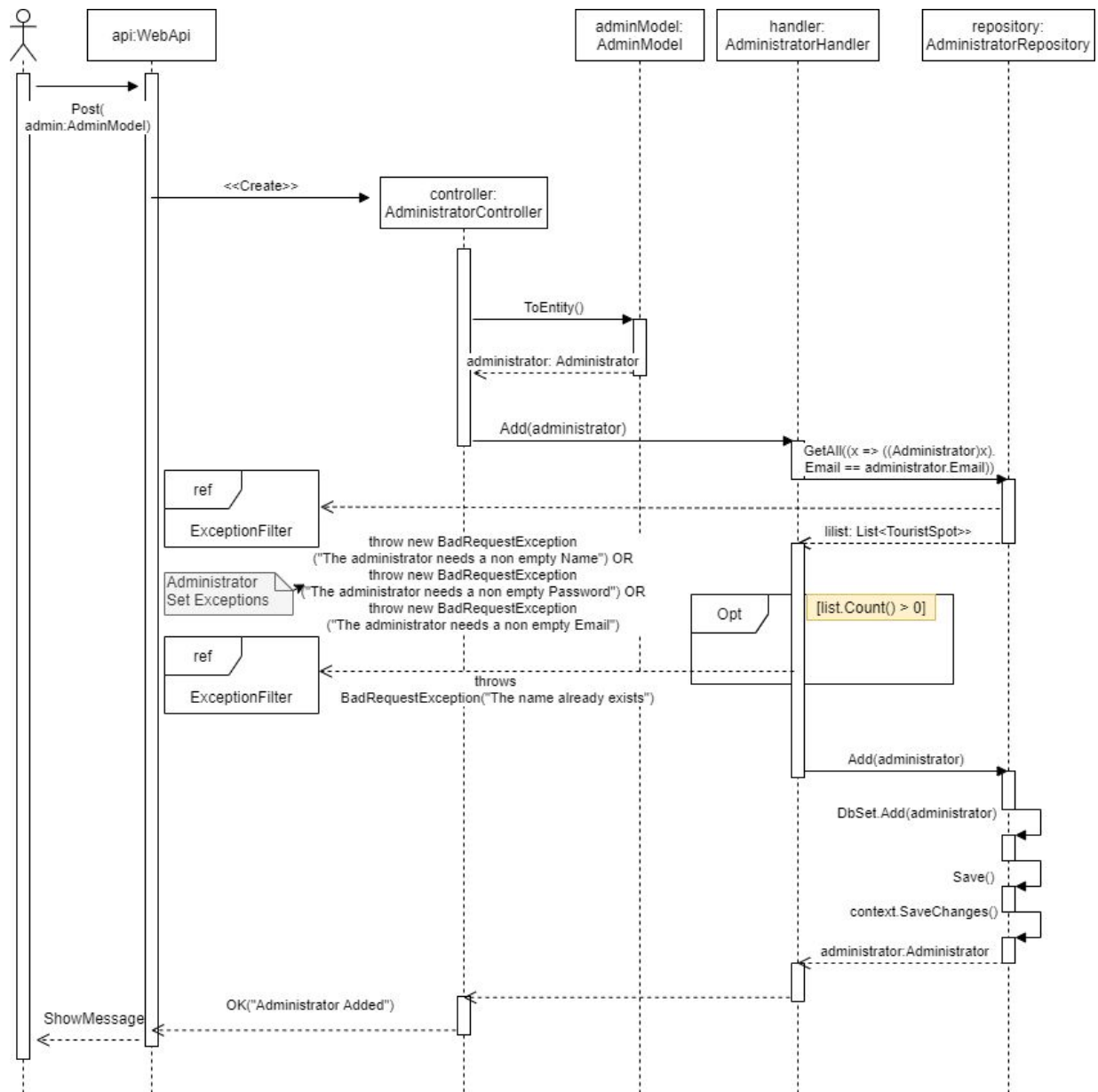
Anexo

Diagramas de interacción

Autenticación de agregado de administradores



Agregado de administradores luego de la autenticación



Autenticación de agregado de puntos turísticos

