Universidad ORT Uruguay

Facultad de Ingeniería

Obligatorio 2
Diseño de Aplicaciones 2

Descripción del diseño

Agustín Introini (211064)
Juan Ignacio Balian (211150)

Docentes:
Ignacio Valle, Daniel Acevedo

Repositorio en Github:
https://github.com/ORT-DA2/211150-211064-OBL

26 de noviembre de 2020

Declaraciones de autoría

Nosotros, Agustín Introini, Juan Balian, declaramos que el trabajo que se presenta en esta obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos Diseño de Aplicaciones 2;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Agustín Introini

Juan Ignacio Balian

Resumen

Trabajo realizado por Juan Ignacio Balian y Agustín Introini como segundo obligatorio de la materia Diseño de Aplicaciones 2. El sistema trata en esta versión de una API REST que ofrece las operaciones necesarias para el manejo de un sistema de reservas. El obligatorio fue construido aplicando TDD y teniendo en cuenta los principios de diseño SOLID, guias de diseño GRASP y estándares de Clean Code. También, esta versión del obligatorio incluye frontend desarrollado en Angular. Se utilizo Git para el control de versiones y GitHub para el repositorio remoto. La base de datos fue implementada usando SQL Server y el ORM Entity Framework modalidad Code First.

Índice general

1. Descripción del sistema

1.1. Funcionalidades del sistema

El sistema es una Web API y una aplicación para navegadores web que permite ofrecer un servicio a dos tipos de usuarios diferentes. Por un lado, permite que turistas puedan realizar reservas de hospedajes en Uruguay, habilitándolos a hacer búsqueda filtrables tanto de puntos turísticos en una región como de hospedajes en un punto turístico.

También permite a administradores gestionar el sistema, permitiéndoles crear, eliminar y actualizar la capacidad de los hospedajes, crear puntos turísticos, cambiar el estado de las reservas creadas por turistas y realizar el mantenimiento de los demás administradores.

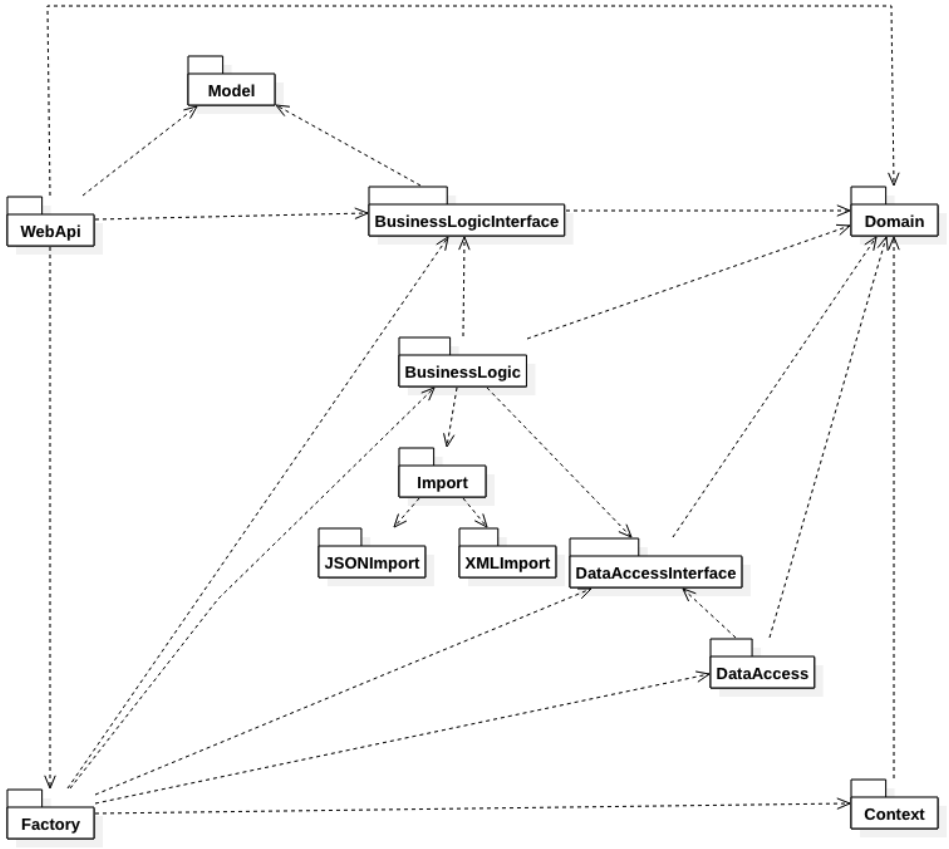
1.2. Aclaraciones

- Decidimos guardar las imágenes en base de datos utilizando un string base 64. Sabemos que no es la decisión mas correcta, pero como confirmaron en el foro para el alcance del obligatorio es correcto.
- Decidimos guardar el historial de estados completo de las reservas. Entonces cuando se pide ver el estado de una reserva se muestran todos, para obtener el actual va a ser necesario desde frontend obtener el estado con mayor id.

2. Diseño

2.1. Estructura de Paquetes

2.1.1. Dependencia de paquetes



Nuestra principal intención fue separar a los paquetes en dos niveles cuando sea necesario, un nivel con los objetos concretos y otro con abstracciones y objetos estables, para de esta manera cumplir con el Dependency Inversion Principle, así desacoplando nuestros módulos de software.

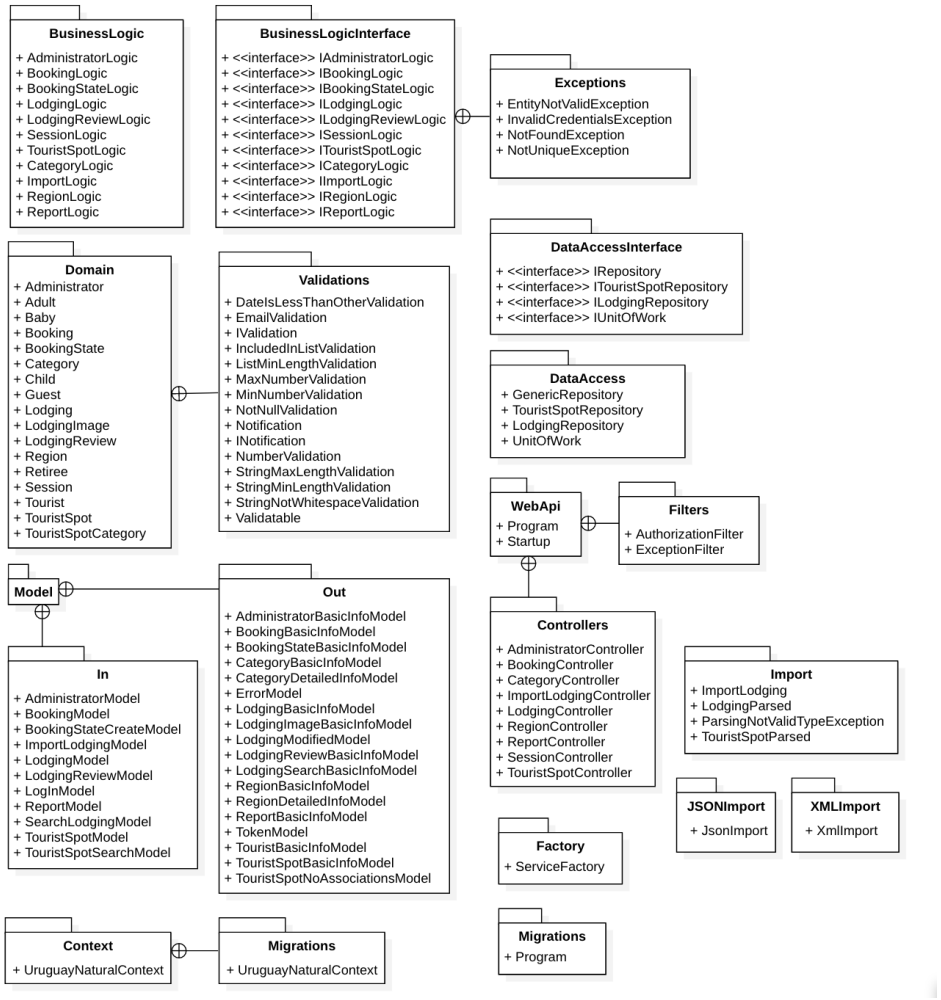
Además, nuestros paquetes fueron separado en capas donde de un lado se acopla a la idea de ser una Rest API que consume JSON, del otro se acopla al acceso de datos/ORM Entity Framework, para luego tener paquetes que describan la lógica de negocio con el objetivo de independizarnos del ORM tanto como de que es una Rest API.

Descripcion de cada paquete:

- **WebApi:** Es el paquete ejecutable del proyecto, donde se encuentran los distintos controladores que definen los endpoints y los filters
- **Model:** Paquete que define todos los modelos tanto de entrada como de salida.
- **BusinessLogicInterface:** Establece las interfaces necesarias para las clases de lógica. "Conecta" al paquete WebApi con el paquete de lógica. Incluye las excepciones, decidimos ponerlo aquí dado que estas son clases muy estables, no consideramos que haga falta ponerlas en su propio proyecto, dado que no vemos ninguna ventaja en separarlas de la lógica que las usaría.
- **BusinessLogic:** Contiene la lógica de negocio para cumplir con los requerimientos funcionales de la aplicación.
- **DataAccessInterface:** Establece las interfaces necesarias para las clases que deseeen acceder a la base de datos.
- **DataAccess:** Es el paquete que tiene la responsabilidad de comunicarse con la base de datos.
- **Domain:** Es el paquete que contiene las entidades de la solución junto con sus validaciones.
- **Context:** Paquete el cual su única responsabilidad es la de mantener las sesión con la base de datos, por lo tanto estamos respetando SRP.
- **Factory:** Encargado de registrar los servicios en IServiceCollection por lo que conoce tanto los paquetes de interfaces como su implementación
- **Import:** Contiene lo necesario para que un paquete que dependa de él pueda implementar la lógica para hacer importaciones en masa.
- **JSONImport:** Implementación ejemplo de interfaz definida en Import
- **XMLImport:** Implementación ejemplo de interfaz definida en Import

2.1.2. Diagrama de paquetes con nesting

A continuación se muestra el diagrama de paquetes, con el listado de las clases que componen a cada uno de ellos.

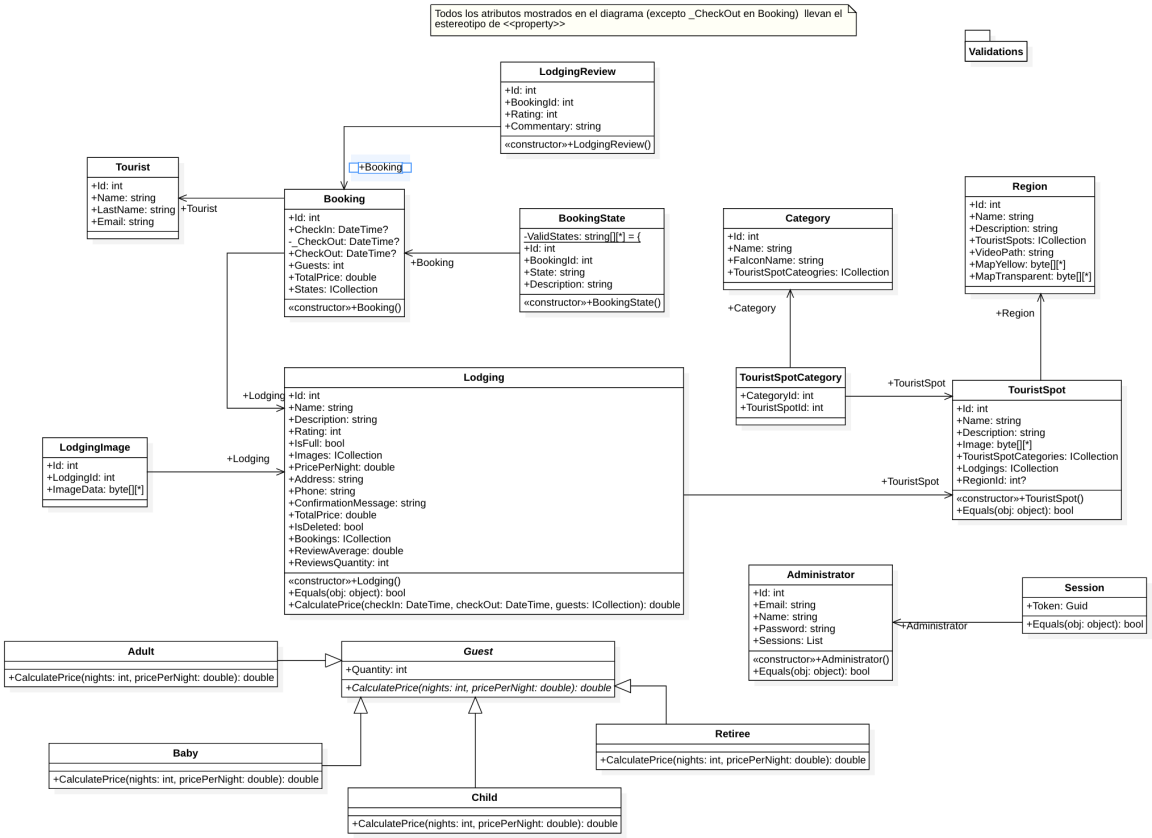


2.2. Paquetes del sistema y sus diagramas de clase

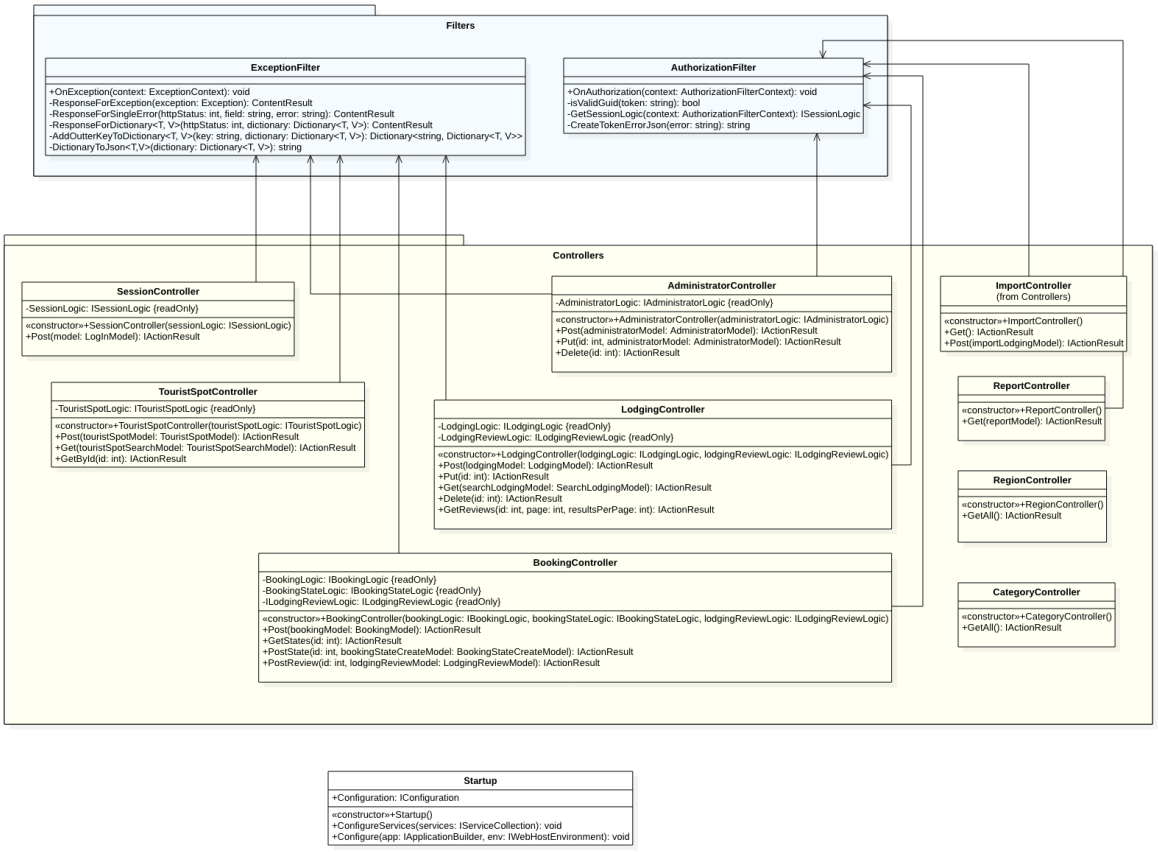
2.2.1. Domain

El proyecto domain, incluye todas las clases de entidades de negocio, pero también incluye como puede verse en el diagrama de nesting 2.1.2 a las validaciones. Por otra parte, también incluye las clases que utilizamos implementando polimorfismo que son Guest y sus implementaciones.

En el diagrama se utiliza una coloración, que no es más que para que pueda visualizarse con mayor facilidad las asociaciones que existen entre las validaciones y las entidades.

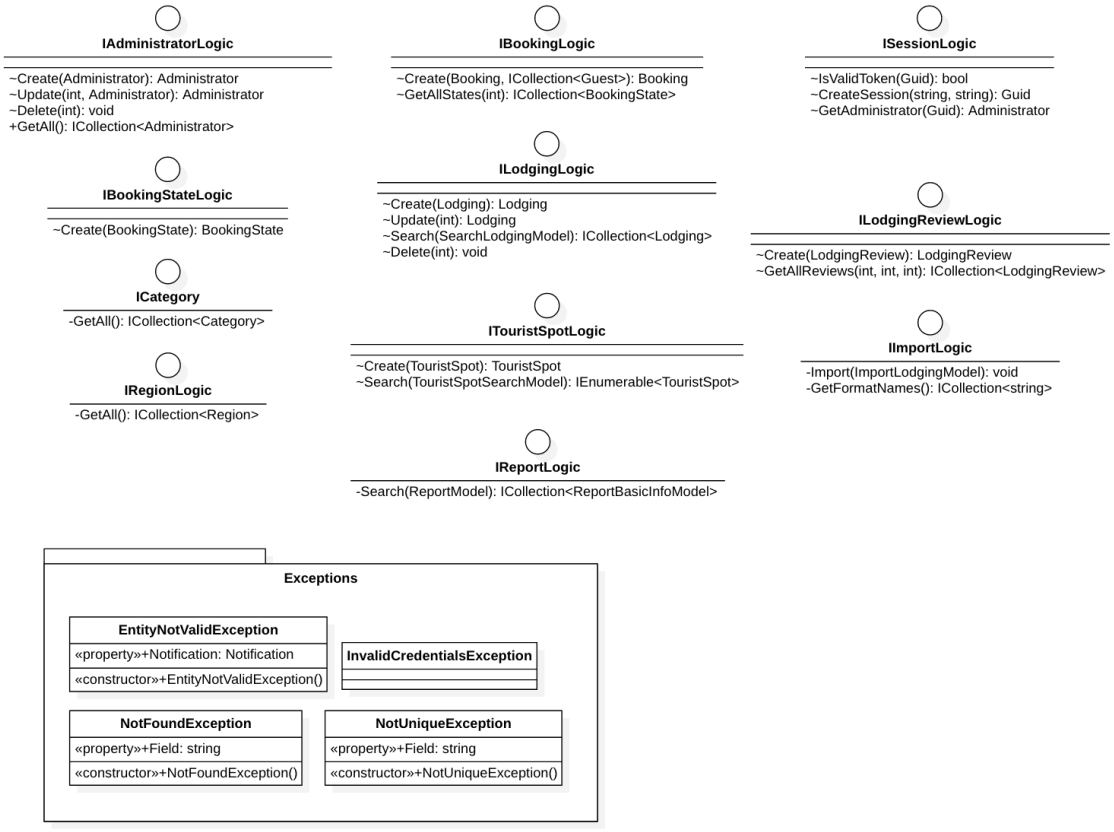

Todos los atributos mostrados en el diagrama (excepto _CheckOut en Booking) llevan el estereotipo de <<property>>
Validations
LodgingReview
+Id: int
+BookingId: int
+Rating: int
+Commentary: string
«constructor»+LodgingReview()
+Booking
Tourist
+Id: int
+Name: string
+LastName: string
+Email: string
+Tourist
Booking
+Id: int
+CheckIn: DateTime?
-_CheckOut: DateTime?
+CheckOut: DateTime?
+Guests: int
+TotalPrice: double
+States: ICollection
«constructor»+Booking()
+Booking
BookingState
-ValidStates: string[][*] = {
+Id: int
+BookingId: int
+State: string
+Description: string
«constructor»+BookingState()
Category
+Id: int
+Name: string
+FalconName: string
+TouristSpotCateogries: ICollection
+Category
Region
+Id: int
+Name: string
+Description: string
+TouristSpots: ICollection
+VideoPath: string
+MapYellow: byte[][*]
+MapTransparent: byte[][*]
+Region
+Lodging
Lodging
+Id: int
+Name: string
+Description: string
+Rating: int
+IsFull: bool
+Images: ICollection
+PricePerNight: double
+Address: string
+Phone: string
+ConfirmationMessage: string
+TotalPrice: double
+IsDeleted: bool
+Bookings: ICollection
+ReviewAverage: double
+ReviewsQuantity: int
«constructor»+Lodging()
+Equals(obj: object): bool
+CalculatePrice(checkIn: DateTime, checkOut: DateTime, guests: ICollection): double
TouristSpotCategory
+CategoryId: int
+TouristSpotId: int
+TouristSpot
TouristSpot
+Id: int
+Name: string
+Description: string
+Image: byte[][*]
+TouristSpotCategories: ICollection
+Lodgings: ICollection
+RegionId: int?
«constructor»+TouristSpot()
+Equals(obj: object): bool
+TouristSpot
LodgingImage
+Id: int
+LodgingId: int
+ImageData: byte[][*]
+Lodging
Administrator
+Id: int
+Email: string
+Name: string
+Password: string
+Sessions: List
«constructor»+Administrator()
+Equals(obj: object): bool
+Administrator
Session
+Token: Guid
+Equals(obj: object): bool
Adult
+CalculatePrice(nights: int, pricePerNight: double): double
Guest
+Quantity: int
+CalculatePrice(nights: int, pricePerNight: double): double
Retiree
+CalculatePrice(nights: int, pricePerNight: double): double
Baby
+CalculatePrice(nights: int, pricePerNight: double): double
Child
+CalculatePrice(nights: int, pricePerNight: double): double

2.2.2. WebApi

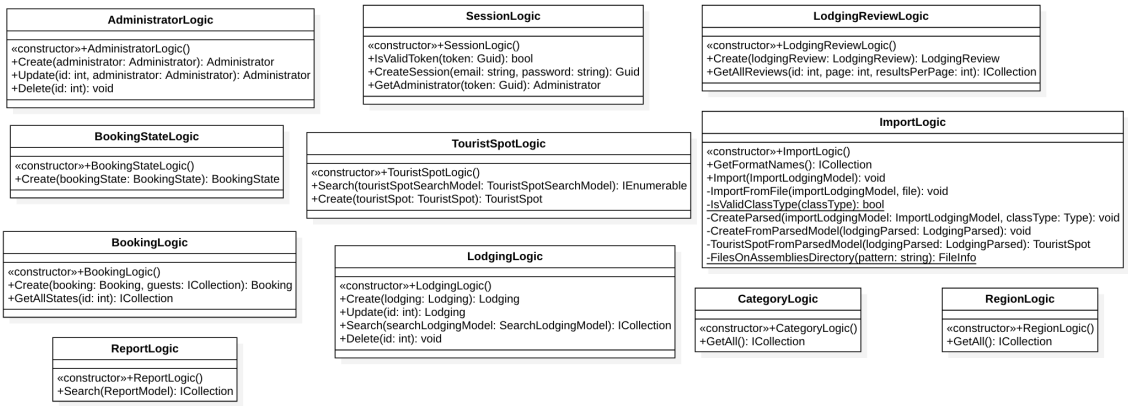


Como era de esperarse en WebApi se encuentran los Controladores, los mimsos estan separados por endpoints. También en el diagrama se puede ver el paquete filters, dentro del mimso se encuentran el filtro para excepciones y el filtro de autorización. Mediante el diagrama podemos ver que filtro utiliza cada clase o controller.

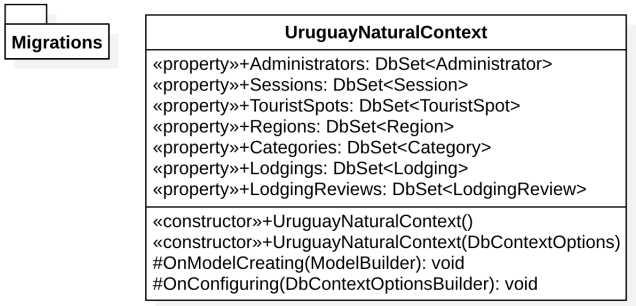
2.2.3. BusinessLogicInterface



2.2.4. BusinessLogic

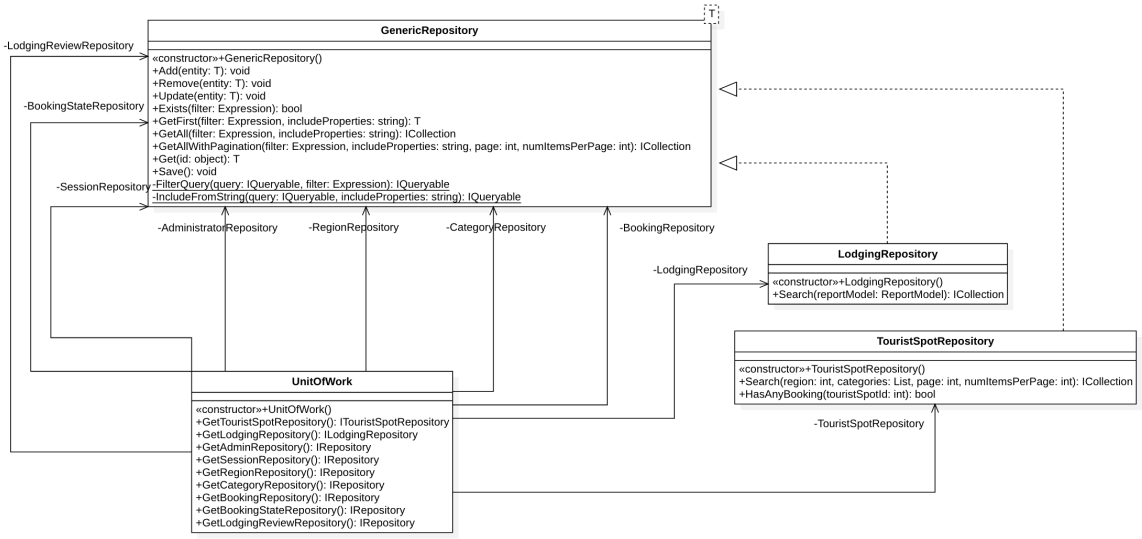


2.2.5. Context

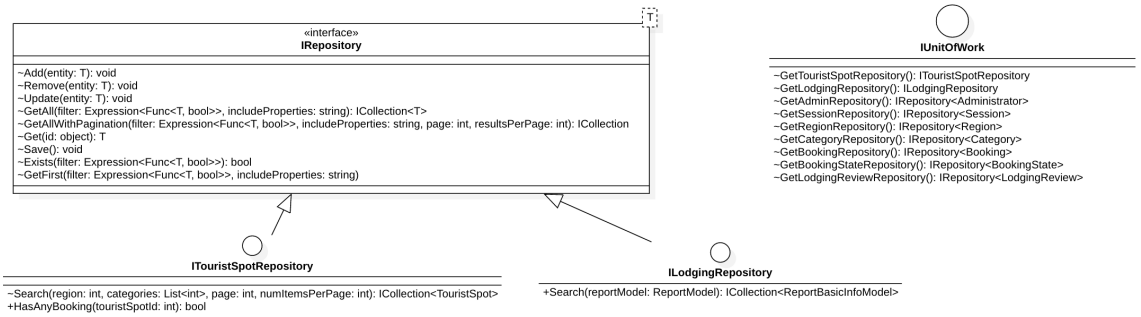


2.2.6. DataAccess

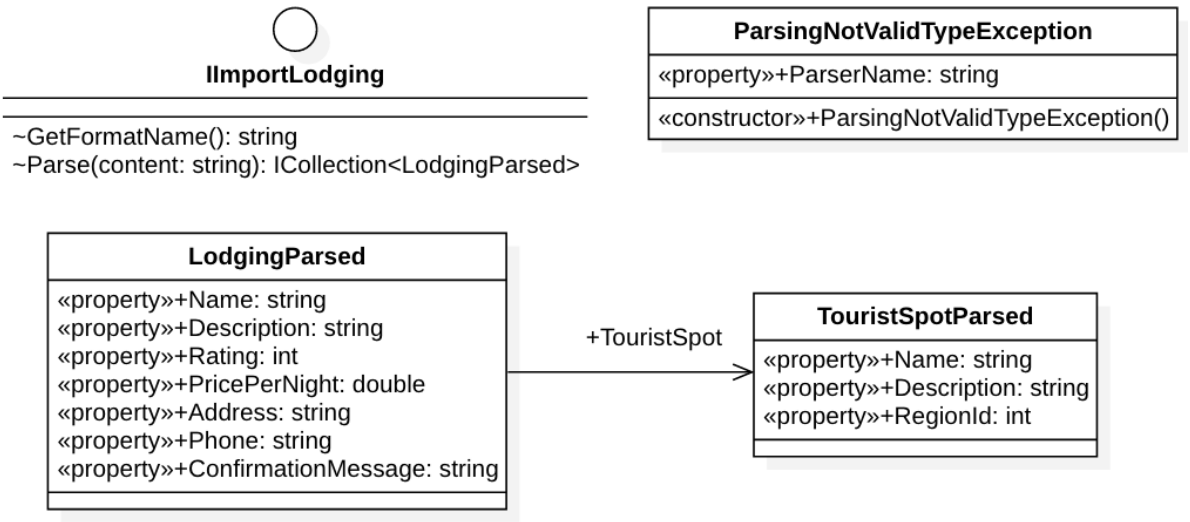
En el diagrama de clases de DataAcces podemos ver la implementación del patrón Unit of Work, que más adelante se explica con detalle en la sección 2.7.1. Además de la creación de dos repositorios concretos, como lo son el de lodgings y touristspots, implementados para cumplir con SRP, y utilizados para realizar consultas específicas de dichas entidades en la base de datos sin necesidad de traer todos los datos a memoria.



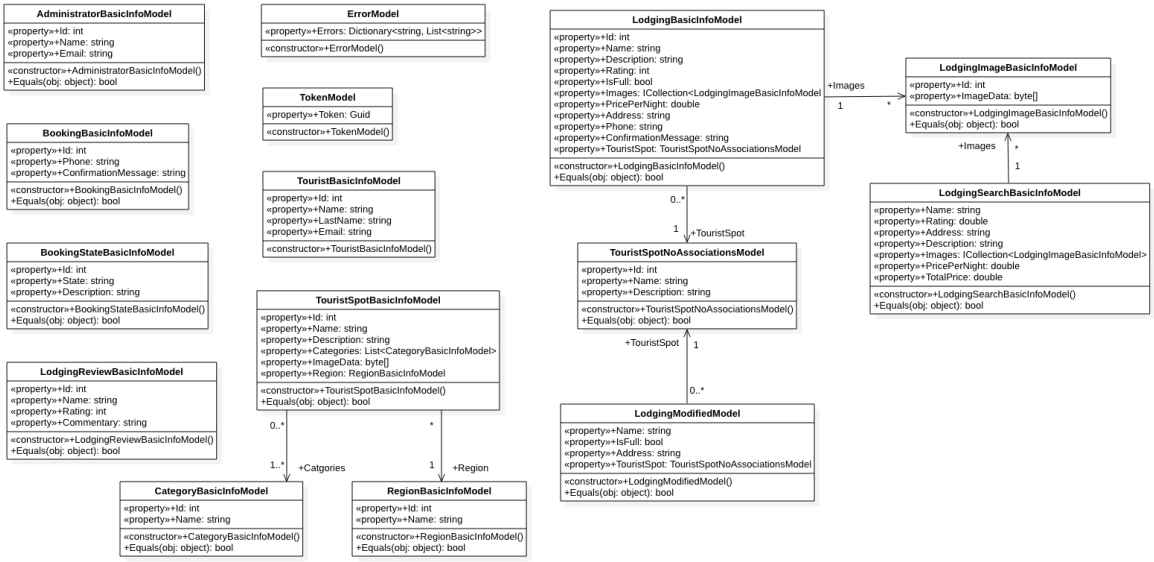
2.2.7. DataAccessInterface

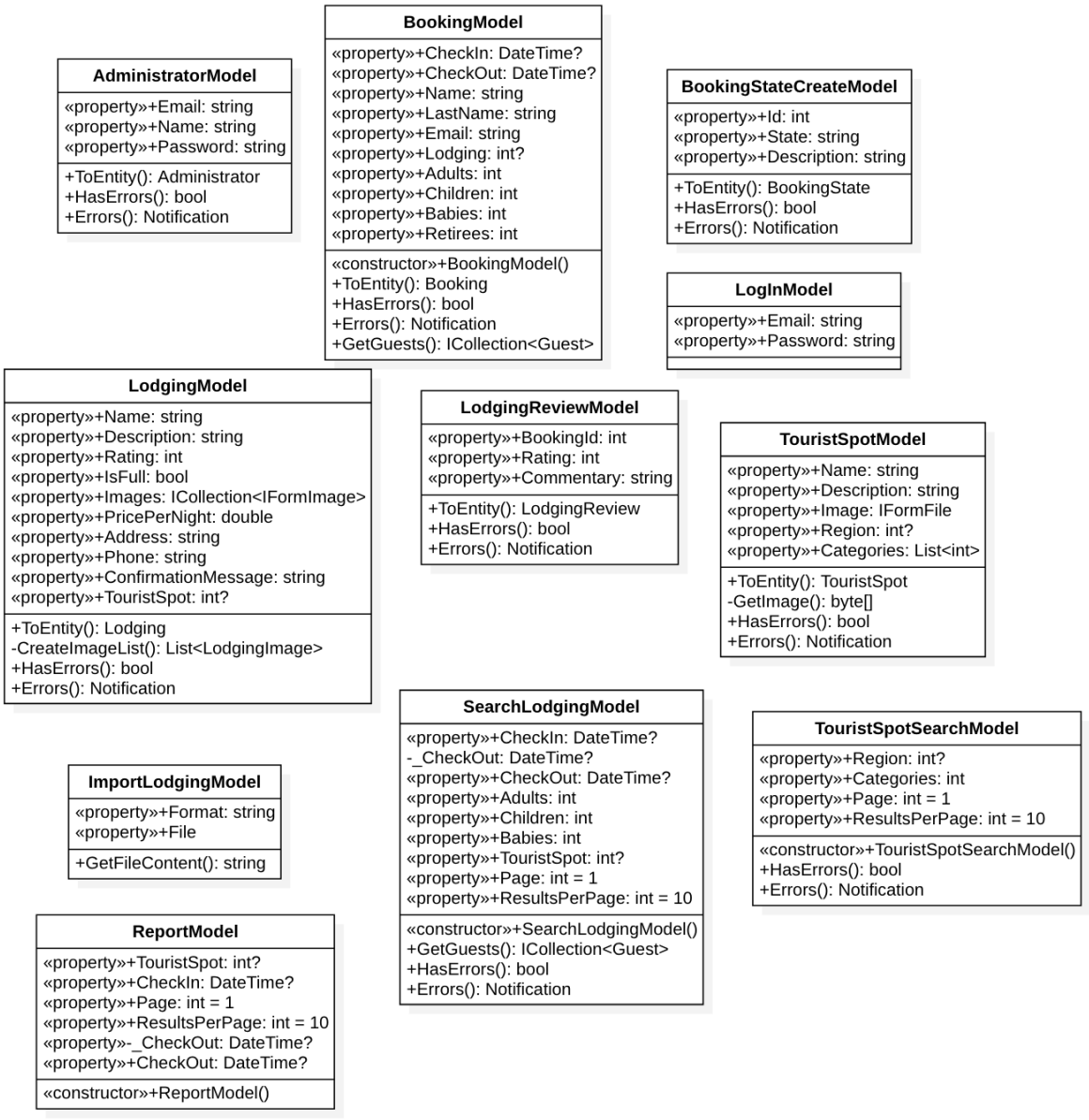


2.2.8. Import



2.2.9. Model


AdministratorBasicInfoModel
«property»+Id: int
«property»+Name: string
«property»+Email: string
«constructor»+AdministratorBasicInfoModel()
+Equals(obj: object): bool
ErrorModel
«property»+Errors: Dictionary<string, List<string>>
«constructor»+ErrorModel()
TokenModel
«property»+Token: Guid
«constructor»+TokenModel()
BookingBasicInfoModel
«property»+Id: int
«property»+Phone: string
«property»+ConfirmationMessage: string
«constructor»+BookingBasicInfoModel()
+Equals(obj: object): bool
TouristBasicInfoModel
«property»+Id: int
«property»+Name: string
«property»+LastName: string
«property»+Email: string
«constructor»+TouristBasicInfoModel()
BookingStateBasicInfoModel
«property»+Id: int
«property»+State: string
«property»+Description: string
«constructor»+BookingStateBasicInfoModel()
+Equals(obj: object): bool
TouristSpotBasicInfoModel
«property»+Id: int
«property»+Name: string
«property»+Description: string
«property»+Categories: List<CategoryBasicInfoModel>
«property»+ImageData: byte[]
«property»+Region: RegionBasicInfoModel
«constructor»+TouristSpotBasicInfoModel()
+Equals(obj: object): bool
LodgingReviewBasicInfoModel
«property»+Id: int
«property»+Name: string
«property»+Rating: int
«property»+Commentary: string
«constructor»+LodgingReviewBasicInfoModel()
+Equals(obj: object): bool
0..*
1..*
+Catgories
*
1
+Region
CategoryBasicInfoModel
«property»+Id: int
«property»+Name: string
«constructor»+CategoryBasicInfoModel()
+Equals(obj: object): bool
RegionBasicInfoModel
«property»+Id: int
«property»+Name: string
«constructor»+RegionBasicInfoModel()
+Equals(obj: object): bool
LodgingBasicInfoModel
«property»+Id: int
«property»+Name: string
«property»+Description: string
«property»+Rating: int
«property»+IsFull: bool
«property»+Images: ICollection<LodgingImageBasicInfoModel
«property»+PricePerNight: double
«property»+Address: string
«property»+Phone: string
«property»+ConfirmationMessage: string
«property»+TouristSpot: TouristSpotNoAssociationsModel
«constructor»+LodgingBasicInfoModel()
+Equals(obj: object): bool
+Images
1
*
LodgingImageBasicInfoModel
«property»+Id: int
«property»+ImageData: byte[]
«constructor»+LodgingImageBasicInfoModel()
+Equals(obj: object): bool
+Images
*
1
LodgingSearchBasicInfoModel
«property»+Name: string
«property»+Rating: double
«property»+Address: string
«property»+Description: string
«property»+Images: ICollection<LodgingImageBasicInfoModel>
«property»+PricePerNight: double
«property»+TotalPrice: double
«constructor»+LodgingSearchBasicInfoModel()
+Equals(obj: object): bool
0..*
1
+TouristSpot
TouristSpotNoAssociationsModel
«property»+Id: int
«property»+Name: string
«property»+Description: string
«constructor»+TouristSpotNoAssociationsModel()
+Equals(obj: object): bool
+TouristSpot
1
0..*
LodgingModifiedModel
«property»+Name: string
«property»+IsFull: bool
«property»+Address: string
«property»+TouristSpot: TouristSpotNoAssociationsModel
«constructor»+LodgingModifiedModel()
+Equals(obj: object): bool



2.3. Modelo de tablas

El modelo de tablas nos muestra como quedó representado nuestro dominio en la base de datos. En el diagrama 2.1 podemos apreciar las diferentes claves foráneas que existen entre las distintas tablas. También nos permite observar cuales son los atributos que finalmente son persistidos en la base de datos.

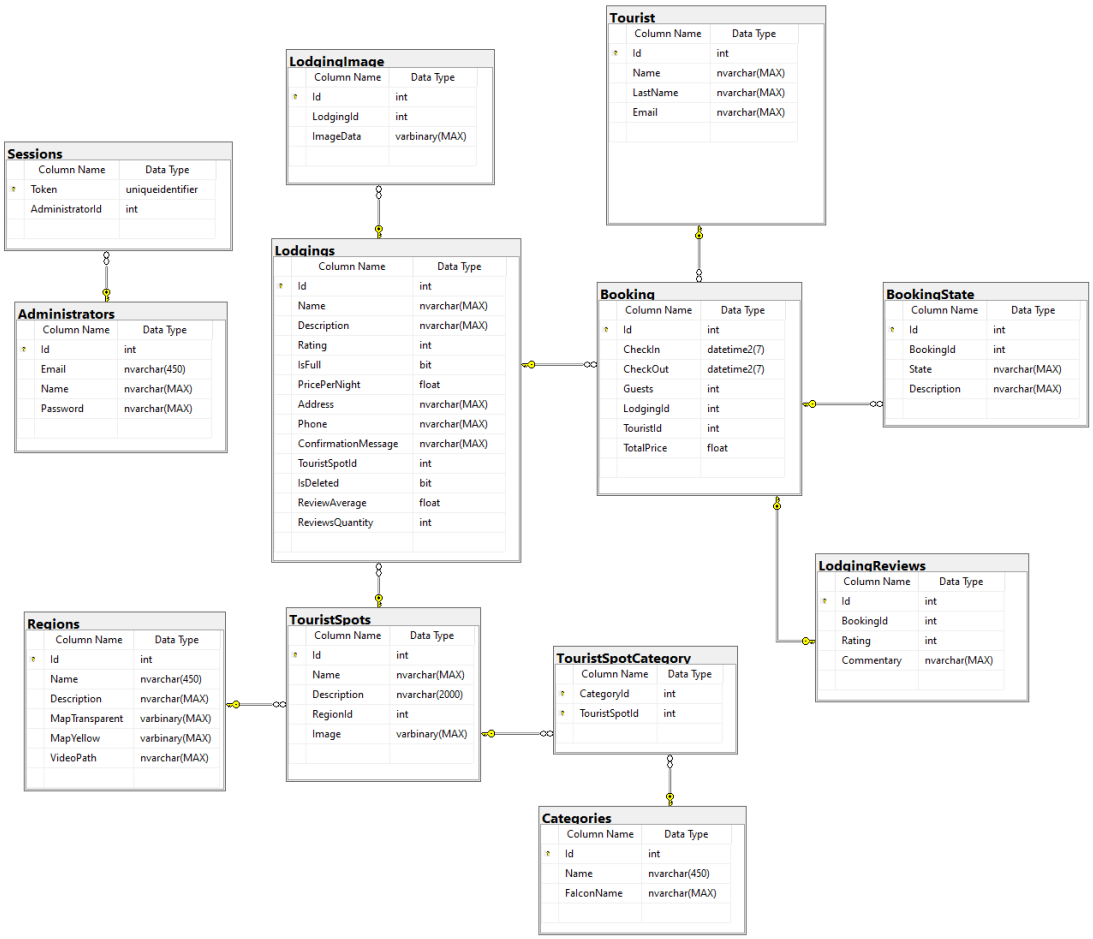


Figura 2.1: Modelo de tablas

En el modelo anterior, podemos observar que la tabla TouristSpotCategory, es una "Join table" como suele llamarse. La misma se encarga de persistir la relación many to many (o N a N) que existe entre TouristSpots y Categories.

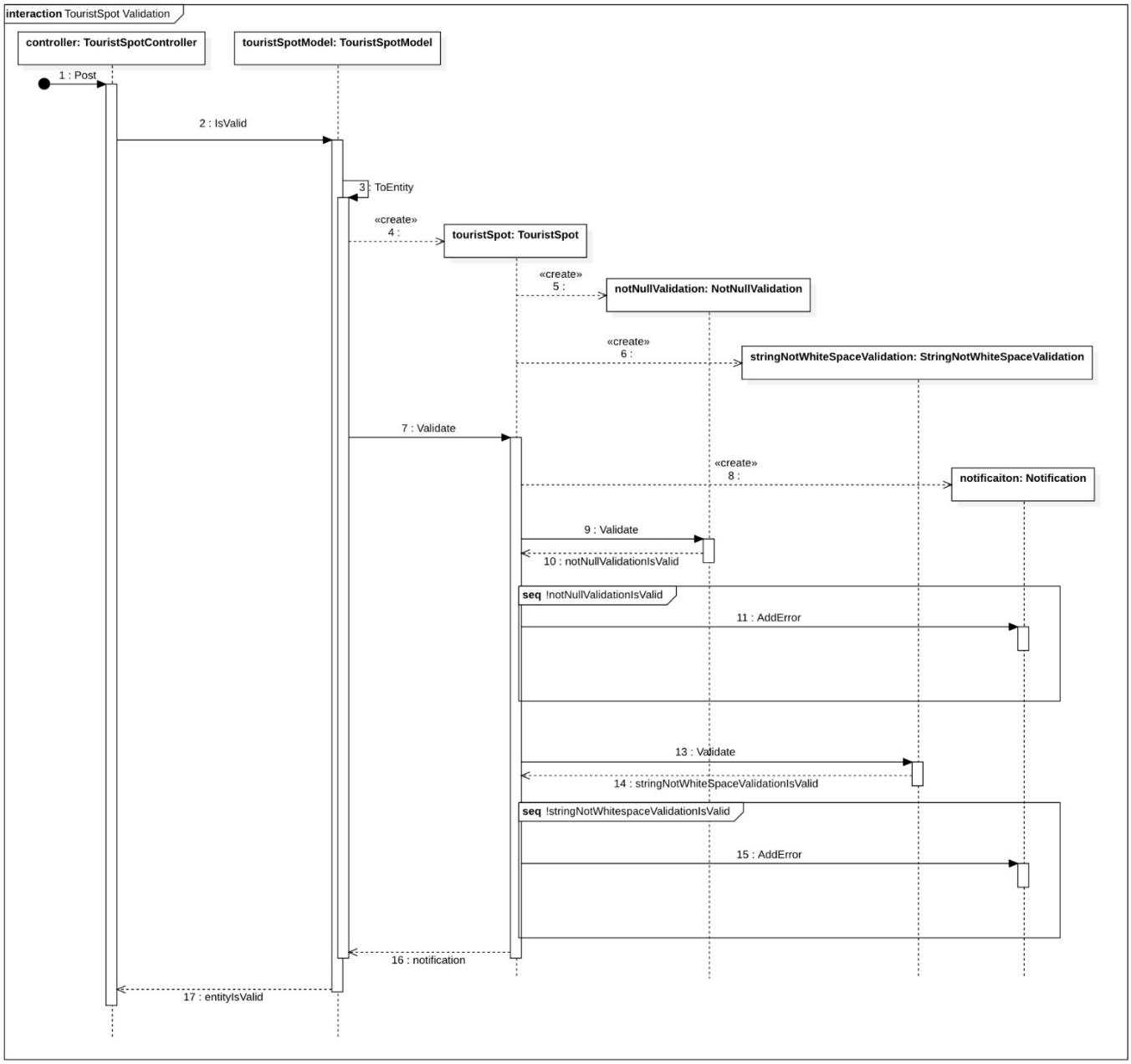
2.4. Funcionalidades clave

Detallaremos algunas de las funcionalidades clave para con ellas entrar mas a detalle de nuestro diseño, donde podremos comentar algunos puntos que consideramos importantes sobre nuestras decisiones.

2.4.1. Validaciones

Nuestra solución para las validaciones se explica en detalle en la parte 2.6.2, pero ahora vamos a entrar a detalle de como interactúan nuestras clases para conseguir el primer nivel de validaciones, o sea la validación sobre los valores proporcionados por el cliente sin importar la integridad de los mismos con el estado de la base de datos.

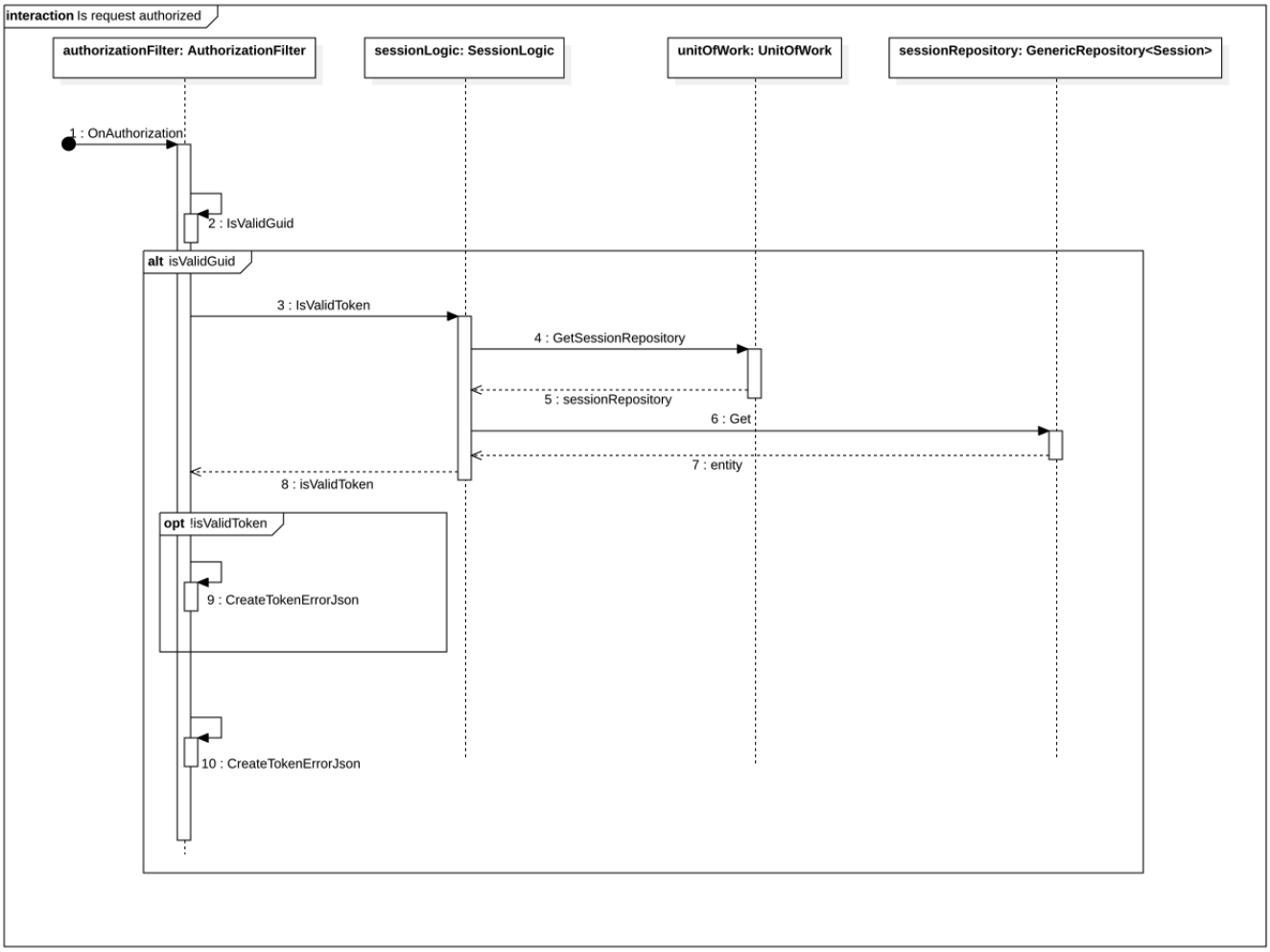
Ejemplificaremos con la creación de un nuevo TouristSpot mostrando como asignamos la responsabilidades a nuestras clases, y que principios seguimos para llegar a nuestro diseño final con la ayuda de diagramas de secuencia y colaboración.



Como se puede ver, cuando llega un POST a TouristSpotController, creamos una entidad con los valores proporcionados en el modelo y esta entidad se valida utilizando nuestras clases de validación y agregando a la notificación un nuevo error en el caso de que el valor que contiene la propiedad (esto lo conseguimos usando reflection) agregándolo a la notificación, y esta es la que se devuelve.

2.4.2. Autenticación

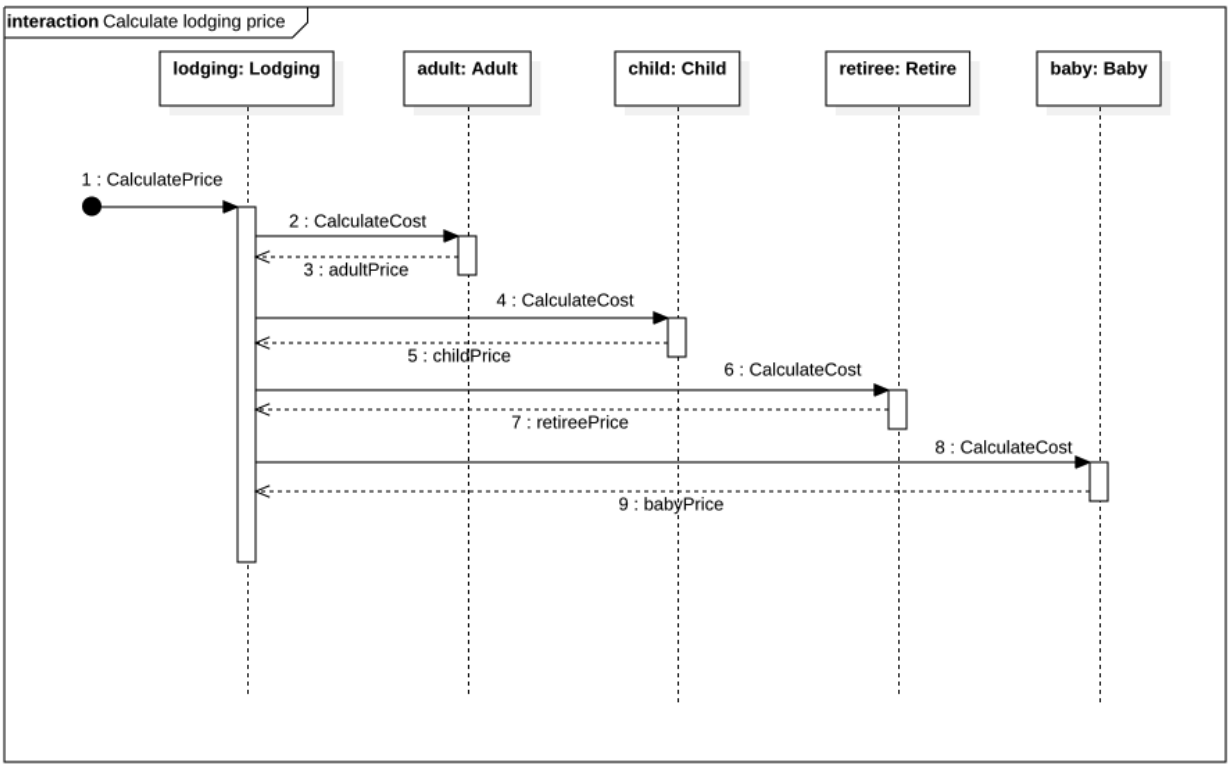
Otra funcionalidad importante es validar que el administrador esta autenticado. Dado que HTTP es un protocolo stateless, no podemos guardar el estado de un cliente en el servidor, entonces pasamos un token haciendo referencia a la sesión del adminsitrador en el header de cada request que debe ser autorizada. En caso de no estar autorizada, se devuelve una response con código de error 401 o 403 dependiendo del caso. El mecanismo se muestra con mas detalle en el siguiente diagrama de secuencia.



Podemos ver cuando entra a este filtro (en el documento de especificación de la api se habla con mas detalle de como funcionan los filtros en ASP.NET) delega el trabajo a SessionLogic el cual luego consulta en base de datos si existe este token, en caso de existir continua y en caso contrario devuelve el código de error descrito anteriormente.

2.4.3. Mecanismo de calculo de precio

Siguiendo el Open Closed Principle, y los GRASP polimorfismo y experto de información, optamos por una solucion en la que todos los tipos de hospedajes hederan de una clase abstracta ‘Guest’ que tiene una función abstracta ’CalculatePrice’ la cual sus hijos (Adult, Baby, Child, Retiree) deben implementar, teniendo en cuenta la cantidad de cada uno de estos.



Como se ve en el diagrama de secuencia anterior, cuando se calcula el precio de un hospedaje, este va a recorrer sus huéspedes y calculando el costo para los que sean necesarios y luego sumándolos, dejando la lógica de descuento a responsabilidad de clases hijas.

2.5. Mecanismo Importaciones

Uno de los requerimientos de nuestro trabajo era "soportar agregar nuevos hospedajes y sus puntos turísticos desde cualquier tipo de fuente pudiendo ser tan diversa como se requiera, por ejemplo desde un .xml, desde un .csv, desde un .json, leyéndolo desde una base de datos, consumiendo un servicio, etc. Como no se conocen todos los formatos posibles y se quiere permitir extensibilidad a futuro, es que se desea que un tercero pueda extender nuestro sistema agregando nuevas formas de importación propias".

Optamos por usar Reflection para resolver este problema, dado que permite definir una interfaz mediante la cual un desarrollador pueda poner opciones dentro de un menú de importación, las cuales ejecutarán código desarrollado por terceros, sea esta la implementación que sea, mientras siga la interfaz que definimos todo debería funcionar correctamente.

Además, nos damos cuenta que reflection es una muy buena opción dado que utilizando los DLLs compilados podemos cumplir con el punto que dice "Estas opciones deben poder ser agregadas al sistema sin necesidad de recompilar la aplicación".

2.5.1. Descripción de nuestra solución

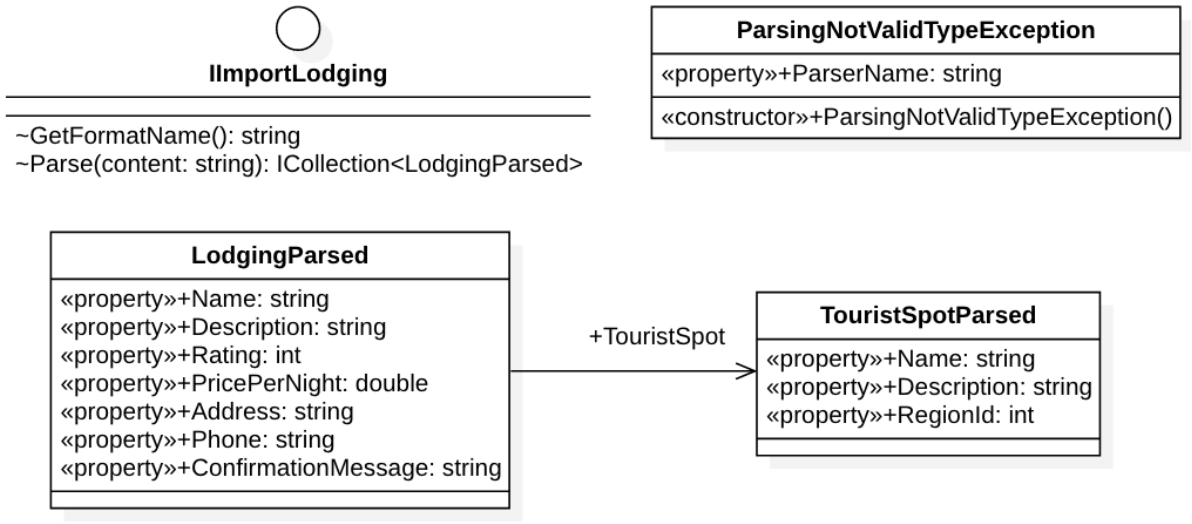
Contamos con una clase "ImportLogic" el cual se encarga de mostrar todos los nombres de formatos de importaciones posibles y de al recibir el nombre de un formato de importación y un archivo crear las entidades que representa el mismo. En el ultimo punto es que usamos Reflection, para utilizar el string que nos pasan como parámetro y buscar la implementación con este nombre, para luego pasarle a esta implementación de nuestra interfaz IImportLodging el contenido al archivo que debe parsear.

Mas detalles de como manejamos las importaciones en la REST API se dan en nuestro documento de la especificación del API, anexado al final de este documento.

Otra decisión que tomamos, es que de manera de no depender del dominio, manejamos las clases LodgingParsed y TouristSpotParsed de manera separada del mismo. Esto es debido a que cuando a un desarrollador tercero le pasemos lo que el necesita para implementar nuestra solución de importaciones bastara con proveerlo con el paquete Import, el cual solo define la interfaz IImportLodging, los anteriores dos modelos y la excepción ParsingNotValidTypeException.

2.5.2. Documentación para desarrolladores que implementen un nuevo mecanismo de importación

Diagrama de clases de Import:



Al crear una nuevo formato, se deberá crear una clase que defina el contrato de la interfaz IImportLodging. Para esto es necesario popular los modelos LodgingParsed y TouristSpotParsed y devolverlos en una lista de LodgingParsed, como se puede ver en la firma de IImportLodging::Parse .

De manera de poder devolver una excepción descriptiva, definimos la excepción "ParsingNotValidTypeException" la cual debe ser llamada en la ocasión de recibir una cadena de caracteres que no este formada correctamente como el formato que esta siendo definido exige. Una vez implementada la solución bastara con compilarla y proveernos con el DLL asociado a la recién compilada assembly.

2.6. Justificación del diseño

2.6.1. Inyección de dependencias

Con el objetivo de romper el acoplamiento entre diferentes componentes de nuestra API utilizamos el patrón inyección de dependencias, el cual mediante interfaces que definen el contrato de un componente de nuestro sistema nos permite 'inyectar' cual seria la clase concreta que vamos a usar con ese contrato. Para conseguir esto, utilizamos el patrón Factory para hacer esta inyección, obteniendo un objeto completo y consecuentemente escondiendo el tipo concreto del cliente. De esta manera, obtenemos código mas limpio, software mas fácil de testear mediante mockeos, no violamos SRP dado que nuestra lógica de creación de objetos no esta relacionada a la lógica de cada modulo y no cumplimos con el OCP dado que el acoplamiento entre módulos es a nivel de interfaz.

2.6.2. Validaciones / manejo de errores

Al comenzar las validaciones consideramos importantes los siguientes puntos:

1. El cliente de nuestra API debe entender con completitud porque su request no fue aceptada, solo cambiando los errores a medida que nos acercamos a capas de bajo

nivel (mas cercanas a la base de datos). Devolviendo los mensajes de errores de manera consistente para facilitar el parseo del mismo.

2. Si se considera pertinente las validaciones redundantes entre capas no son un problema, dado que nos permitirá cambiar las mismas en el caso que sea necesario con mayor facilidad. Es por esto que contamos restricciones a nivel de base de datos definidas con FluentApi, nuestras propias validaciones y validaciones a nivel de lógica que consultan con la base de datos antes de conseguir el error que generarían las restricciones de base de datos en algunos casos y permitiéndonos generar un mensaje mas preciso para el cliente.

3. Validar en el dominio buscando no seguir el anti-patrón Anemic Domain Model, separar las validaciones particulares de las entidades para seguir SRP y permitirnos agregar nuevas validaciones sin modificar las previamente ingresadas siguiendo OCP. Nuestra solución de validaciones debe hacer agregar validaciones un proceso fácil y eficiente, que deje clara la intención del programador y no contamine el dominio con repetitivas validaciones, sino que delegue esto a otro componente.

Para cumplir con el punto uno decidimos basarnos en el objeto Notificación propuesto por Martin Fowler, juntando todos los errores de la validación en un diccionario con clave string que representa el campo con el error y valor una lista de strings que representan los errores en ese campo. De esta manera, podemos ejecutar todas las validaciones y en el caso de no cumplir varias tanto en distintos campos como en el mismo campo mostrarlas sin problema. A la vez, decidimos utilizar excepciones para validaciones de la lógica con el objetivo de separar el flujo de negocio del de error, lo cual lo obtenemos con un ExceptionFilter. Este ultimo cuenta con la única responsabilidad de manejar las excepciones permitiéndonos responder con el código HTTP adecuado y darle el formato adecuado al mensaje en JSON.

En cuanto al segundo punto tuvimos que tomar la decisión entre usar Data Annotations o Fluent Api. La ultima de estas opciones nos pareció la mejor decisión para separar la configuración de persistencia del dominio, definiendo una linea clara entre las validaciones del dominio y de la base de datos. Para conseguir mostrar los errores de manera adecuada al cliente optamos por validar de manera redundante si un error en la base de datos fuera a ocurrir (foreign key no existe en la base de datos, campo utilizado como índice ya es representado con otra entidad, entre otros) , para manejar sus errores de manera individual y permitirnos no depender de que en la base de datos existan estas validaciones.

Para el último punto que queríamos conseguir con las validaciones optamos por el patrón Specification definido por Martin Fowler[1], utilizando su variación mas sencilla con las consecuencias que el mismo describe como “fácill” y “expresivo”, sin “invertir en un framework complejo” con el problema de que ’se debe crear una subclase para cada criterio de selección’. El ultimo punto no nos parece problemático, dado que no contamos con suficientes validaciones como para preocuparnos por el numero de clases de validaciones y en el caso que lo sea sabemos que haríamos un refactor a la variación “Composite” de este patrón. Como Fowler describe, esta variación del patrón Specification es esencialmente una aplicación del patrón Strategy, como podemos ver en el siguiente diagrama:

[1] https://www.martinfowler.com/apsupp/spec.pdf

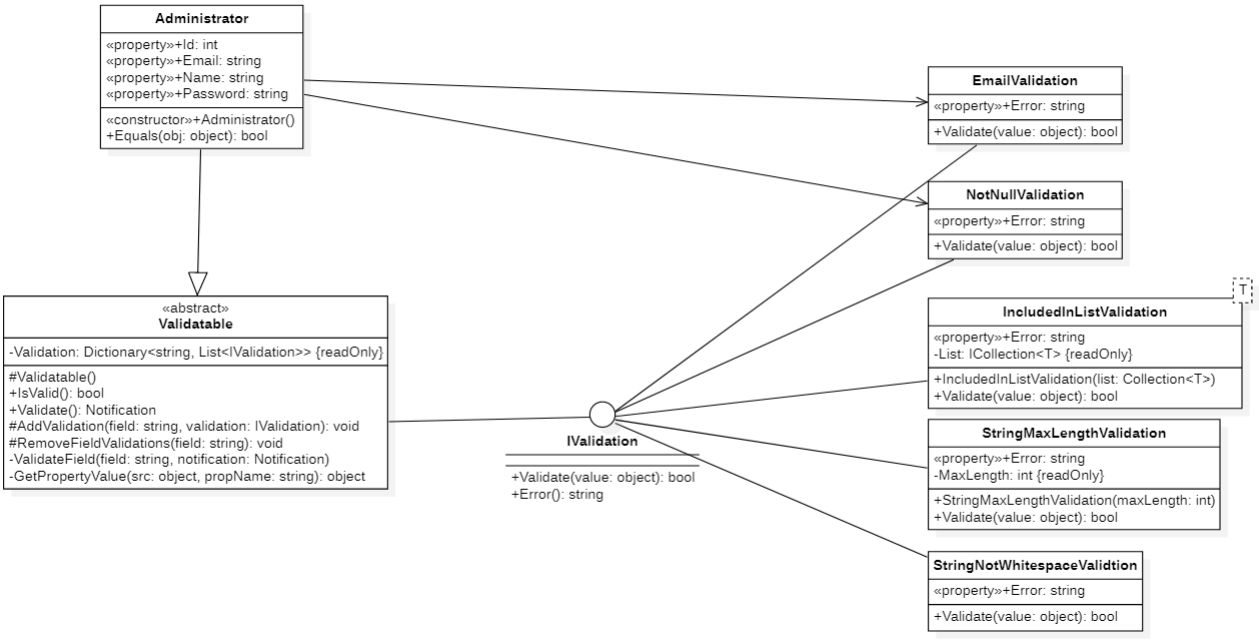


Figura 2.2: El patron que Fowler describe como “esencialmente un Strategy”. Podemos ver que esta en lo cierto dado que es un caso del strategy donde el contexto esta formado por una clase padre y las estrategias las crea la clase hija.

Ademas, para conseguir definir las validaciones en el constructor de la clase del dominio utilizamos Reflection, pasando el nombre del campo a validar y una instancia de la clase que representa la validación, para cuando se llame el método ’Validate()’ obtener el ultimo valor de la propiedad.

2.6.3. Aplicación de principios de diseño

Nuestras decisiones están basadas por los principios GRASP y SOLID, pasaremos a mencionar algunos ejemplos de donde estos últimos nos ayudaron a conseguir el diseño con el que contamos:

- Information Expert: Cuando calculamos el precio de un hospedaje lo hicimos en la entidad Lodging, ya que esta era la que contaba con gran parte de la información para hacerlo.
- Alta cohesión: Al hacer validaciones que tienen una única responsabilidad, obtenemos tanto alta cohesión como SRP.
- Creador: En los modelos creamos la entidad de dominio porque tiene todos los datos para hacerlo, teniendo un método ToEntity()
- Polimorfismo y OCP: Al manejar las entidades Guest, Adult, Child, Baby y Retiree, al algoritmo de calcular precio si se agrega un nuevo tipo (ej. Teenager) basta con crear su objeto y poner su multiplicador. Como se nota en el ejemplo, esto nos permite estar abierto a la extensión y cerrado al cambio.

- ISP: Al decidir hacer dos interfaces distintas IRepository e ITouristSpotRepository podríamos haber agregado el nuevo método que define TouristSpotRepository en GenericRepository y hacer que devuelva nulo por defecto. En cambio, nos apoyamos en el ISP, e hicimos dos interfaces de manera de no obligar a ningún otro repositorio a usar el método que solo define TouristSpotRepository.

2.6.4. Acceso a datos

Nuestros objetivos con el acceso a datos (el cual hicimos usando el ORM EntityFramework Core) eran no filtrar datos en memoria, sino que utilizar a la base de datos para ello y hacer uso de paginado cuando sea necesario. Para ello utilizamos un repositorio genérico, el cual define todos las operaciones básicas que implementan los repositorios. Luego, si lo vemos necesario como por ejemplo lo fue para TouristSpotRepository y LodgingRepository extendemos el repositorio genérico, permitiéndonos hacer consultas complejas filtrando en la base de datos sin copiar en memoria.

Otro aspecto del acceso a datos en el que hicimos hincapié fue en no realizar consultas innecesarias, sino en realizar todo en una consulta solo y dejando que el motor de base de datos maneje esa complejidad.

Como describimos mas adelante, también utilizamos el patrón Unit of Work para asegurarnos que todos los repositorios usen el mismo contexto, evitando problemas que podrían surgir al actualizar diferentes tipos en una misma transacción.

2.6.5. Excepciones

En cuanto al manejo de excepciones, lo hicimos utilizando un filtro (funcionalidad provista por ASP.NET Core explicada mas en detalle en el documento especificación de la API anexado). El cual dependiendo de que excepcion debe manejar crea una request siguiendo los formatos de error definidos de manera de ser consistente y con el codigo de error HTTP que sea necesario. Para esto, definimos nuestras propias excepciones para tener mas control sobre las mismas.

Excepciones en BusinessLogicInterface (al principio del documento se describe porque decidimos ponerlas en este paquete):

- EntityNotValidException: Esta excepción la utilizamos cuando debemos hacer una validación del dominio del lado de la logica, acoplandonos a la interfaz INotification para devolver todos los errores.
- InvalidCredentialsException: Usada cuando se proveen credenciales para log in de administrador incorrectos.
- NotFoundException: Utilizada en los casos que se referencia a una entidad mediante un campo y no existe tal relación en la base de datos.
- NotUniqueException: Cuando ya existe un elemento con un campo definido en la base de datos.

Excepciones en Import:

- ParsingNotValidTypeException: Esta excepción debe ser llamada en la ocasión de recibir una cadena de caracteres que no este formada correctamente como el formato que esta siendo definido exige.

2.7. Mejoras con respecto a la primer entrega

Describiremos las principales cuatro mejores sobre nuestra entrega previa.

2.7.1. Patrón Unit of Work

Este patrón tiene muchos beneficios, como asegurarnos que todos los repositorios usen el mismo contexto, evitando problemas que podrían surgir al actualizar diferentes tipos en una misma transacción, o ayudarnos a cumplir con Clean Code gracias a que no debemos mover a todos los repositorios por parámetros sino que con utilizar este patrón basta con pasar un único parámetro.

2.7.2. Mejora en consulta de búsqueda de puntos turistico

Con lo aprendido en la ultima parte del curso notamos la importancia de realizar la mínima cantidad de consultas a base de datos, por esto revisamos nuestras consultas para la primer entrega y notamos que en TouristSpotRepository actualizabamos un IQueryable filtrando elementos con un where dentro de un for each. Esto lo arreglamos utilizando LINQ.

2.7.3. Mejora al mecanismo de calcular precio

Dados los nuevos requerimientos, nos dimos cuenta que la implementación originalmente pensada para el calculo de precio no nos iba a permitir resolver este problema. Consecuentemente, hicimos un refactor siguiendo el Open Closed Principle, y los GRASP polimorfismo y experto de información, optando por una solución en la que todos los tipos de hospedajes hederan de una clase abstracta 'Guest' que tiene una funcion abstracta 'CalculatePrice' la cual sus hijos (Adult, Baby, Child, Retiree) deben implementar, teniendo en cuenta la cantidad de cada uno de estos.

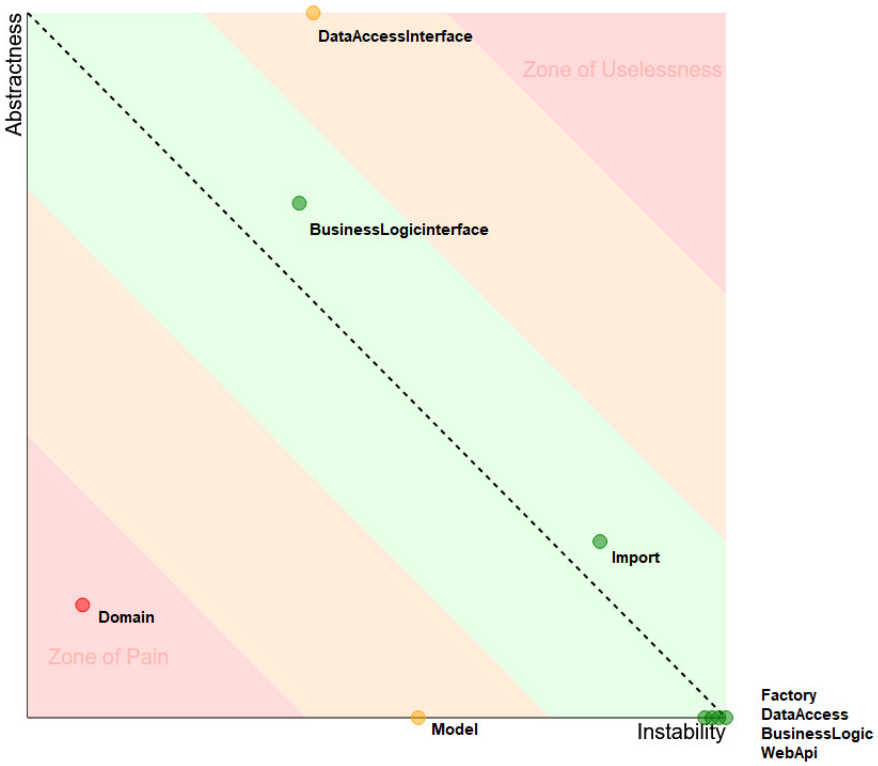
2.7.4. DIP con Interfaz INotification

Como describimos anteriormente, decidimos utilizar el patron Notification de Martin Fowler para mejorar las validaciones. Para la entrega anterior utilizamos la clase Notification la cual ya tiene una implementación de como se va a manejar este mecanismo, el cual se mantiene virtualmente sin cambios pero mejoramos al entender que nos estábamos acoplando a esta implementación de la clase Notification, y partiendo del Dependecy Inversion Principle nos acoplamos a un interfaz INotification el cual define el contrato que todas sus implementaciones deben de seguir, de esta manera haciendo que las clases de alto nivel no dependen de las clases de bajo nivel, sino que haciendo que ambas dependan de abstracciones.

2.8. Análisis de métricas

Utilizamos la herramienta NDepend para obtener métricas sobre los paquetes de nuestro trabajo. Pasaremos a analizar las los resultados del mismo:

2.8.1. Gráfica de Abstracción en función de Inestabilidad



Como podemos ver en nuestro obligatorio solo contamos con tres paquetes que se encuentran fuera de la zona deseable.

- Domain se encuentra en la zona de dolor, lo que significa que es un paquete con muchas responsabilidades y al mismo tiempo muy estable, decir que esta en la zona de dolor implica que un cambio en el mismo impactaría directamente en los paquetes que dependen de él. Aunque consideramos natural que el paquete dominio cumpla con estas propiedades, notamos que en nuestro caso hay oportunidad para la mejora dado que en este paquete también tenemos clases que se encargan de validar mayoritariamente las entidades del dominio pero también algunos modelos, de esta manera resultando en un paquete mas estable de lo necesario. La raíz de este problema se encuentra en que al comienzo del desarrollo con el objetivo de tener mensajes de error consistentes decidimos implementar nuestra propia solución para las validaciones, y luego de analizar donde deberíamos ubicar a las clases con esta responsabilidad, decidimos que de manera de cumplir con el principio de Reuso Común el mejor lugar para estas era en Domain. A su vez Domain es un paquete que consiste de clases poco volátiles, por lo que no consideramos necesariamente un error que se encuentre en la zona de dolor, pero si que amerita especial atención dadas sus responsabilidades.

- Model: Por razones análogas a la de Domain, Model es un paquete estable y concreto. Comparándola con este ultimo, podemos decir que es mas inestable dado que tiene

mas relaciones eferentes y menos aferentes, de esta manera al realizar la ecuación I = Ce / (Ce + Ca) obtiene un valor mas cercano a 1.

- DataAccessInterface: Este paquete esta mas cercano a la zona de poca utilidad dado que aunque se trata de un paquete totalmente abstracto y sin tantas relaciones aferentes. No notamos esto como un error, pero si como un indicador que puede llegar a ser importante en el futuro si decidimos hacer algún refactor.

2.8.2. Cohesión relacional

Assemblies	# lines of code	# IL instruction	# Types	# Abstract Types	# lines of comment	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
Import v1.0.0.0	25	97	4	1	0	2	9	0.75	0.82	0.25	0.05
Domain v1.0.0.0	325	1847	32	5	0	61	5	2.47	0.08	0.16	0.54
Model v1.0.0.0	542	2919	30	0	0	19	24	0.23	0.56	0	0.31
DataAccessInterface v1.0.0.0	0	0	4	4	-	16	11	1.5	0.41	1	0.29
DataAccess v1.0.0.0	54	1186	7	0	0	1	41	1.29	0.98	0	0.02
BusinessLogicInterface v1.0.0.0	9	37	15	11	0	23	15	0.07	0.39	0.73	0.09
BusinessLogic v1.0.0.0	197	2200	11	0	0	1	82	0.09	0.99	0	0.01
Factory v1.0.0.0	15	67	1	0	0	1	34	1	0.97	0	0.02
WebApi v1.0.0.0	155	1085	13	0	2	0	123	0.54	1	0	0

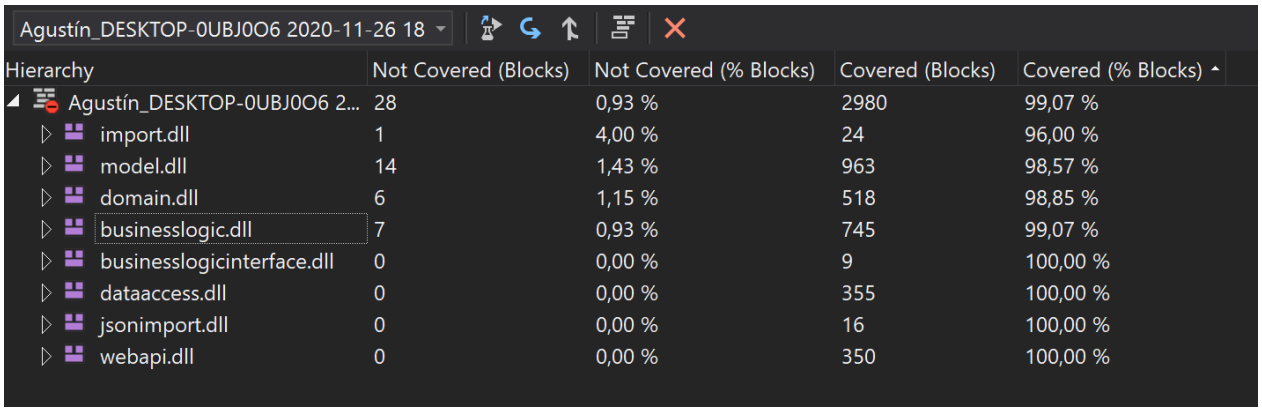
Teniendo en cuenta que el valor aceptable de esta métrica se dice que es entre 1.5 y 4 podemos ver que solo Domain y DataAccessInterface tienen valores aceptables. Luego podemos que ver que DataAccess y Factory tiene valores no tan alejados de los aceptable.

En el resto de los paquetes notamos que tendríamos que realizar algún tipo de refactor para mejorar la calidad de nuestros paquetes partiendo de este indicador, que muestra que las clases dentro de estos paquetes no se relacionan demasiado entre si.

Para mejorar los valores, las ideas principales que tenemos es, comenzando por Model, podriamos ponerlo como un namespace dentro de WebApi, o separar el paquete Model en dos paquetes ModelIn y ModelOut, dado que entre ellas son las que mas se relacionan entonces podríamos mejorar nuestros paquetes de esta manera. Análogamente podríamos realizar mejoras de este estilo a los demás paquetes, pero al tomar estas decisiones debemos también tener en cuenta otros principios de paquete y valores que también son importantes.

2.9. Informe de cobertura

Para la cobertura de pruebas, utilizamos la herramienta de code coverage provista por el Visual Studio Enterprise. Para evitar que el número final de coverage se viera afectado por clases que no debíamos testear, como la de context, o las migrations, excluimos estas clases utilizando la DataAnnotation [ExcludeFromCodeCoverage]. Los resultados fueron los siguientes:



Como se puede ver, la cobertura final es de 99,07 %, valor sumamente cercano al 100 % que obtenemos al realizar TDD. Si bien la cobertura de código no implica código correcto, nos da la pauta de que al menos no estamos dejando ninguna porción de código sin testear.

Bibliografía

[1] K. H. Russ Miles, *Learning UML 2.0*. O'Reilly Media.

[2] R. C. Martin, *clean code: a handbook of agile software craftsmanship*.

[3] J. Humble and D. Farley, *Continuous delivery*, ser. Addison-Wesley signature series. Upper Saddle River: Addison-Wesley, (c)2011, incluye Indice analítico.

Universidad ORT Uruguay

Facultad de Ingeniería

ANEXO I
Obligatorio 2
Diseño de Aplicaciones 2

Especificación de la API

Agustín Introini (211064)
Juan Ignacio Balian (211150)

Docentes:
Ignacio Valle, Daniel Acevedo

Repositorio en Github:
https://github.com/ORT-DA2/211150-211064-OBL

26 de noviembre de 2020

Índice general

1. Criterios / estándares seguidos para el cumplimiento de REST

Para cumplir con el estilo REST y sus principios, nos aseguramos de cumplir con los estándares, criterios o buenas prácticas que se describen en el artículo de Apigee [1] y en el tutorial de REST API [2]. Si bien por cuestiones de alcance del obligatorio, no se cumplen con todos los lineamientos que los documentos previos mencionan (e.g. versionado y los últimos puntos en [1]), hicimos énfasis en el cumplimiento de los siguientes puntos:

1.1. Endpoints

- Utilizamos sustantivos ante que verbos.
- Estos sustantivos, en plural antes que en singular, y sin mezclar jamás ambos.
- En minúscula.
- No utilizamos guiones bajos (_)
- No utilizamos verbos CRUD en las URIs, ya que para eso están los verbos HTTP.
- Utilizamos nombres concretos, ante que abstractos.
- No utilizamos más de dos URLs por recurso.
- Ocultamos complejidad en las URIs utilizando ?.

1.2. Manejo de errores

Nuestra forma de manejar errores se asemeja a la de Twilio [1]. Alineando los códigos HTTP con los errores, y detallando el error con un mensaje.

Como buenas prácticas, seguimos las siguientes:

- Utilizamos códigos de estado HTTP.
- No utilizamos todos los códigos disponibles, si no que nos hicimos con un grupo reducido de ellos. Empezamos con los 3 fundamentales 200 (OK), 400 (Bad Request) y 500 (Internal Server Error), y fuimos agregando a medida que necesitabamos (se detallan más abajo en este documento).
- Seguimos un formato en el que se indica en un mensaje, el origen o el campo donde se produjo el problema sumado a una lista de errores que éste podría tener.

1.3. Verbos HTTP

Siguiendo el alcance de este obligatorio, utilizamos sólo los 4 verbos HTTP básicos, los cuales nos permiten realizar todas las operaciones CRUD, estos son POST, GET, PUT y DELETE, que se mapean respectivamente con las palabras de la sigla CRUD.

1.4. Criterios REST a destacar

Existen otros criterios importantes que se desprenden de utilizar REST, puntuaremos algunos:

- Stateless: Que sea un protocolo stateless significa que cada request HTTP pasa en completo aislamiento. Es decir, cuando el cliente hace una request HTTP esta debe incluir toda la información que sea necesaria para el servidor.
- Cliente-Servidor: Este modelo es una estructura de las aplicaciones distribuidas para particionar tareas o workloads entre dos partes, el cliente que es quien desea consumir un servicio, y servidores que son quienes lo proveen.

1.5. No implementación del versionado

Optamos por no implementar versionado dado que no era importante para nuestro obligatorio, esto es porque dado el alcance del mismo podemos entender que no es necesario.

2. Mecanismo de autenticación

En este obligatorio existen endpoints que pueden ser utilizados sin autenticación, siendo estos los que realiza el "turista". Pero también existen otros que requieren mecanismos de autenticación, como los que utiliza un administrador, estos mecanismos se detallan a continuación:

2.1. Login

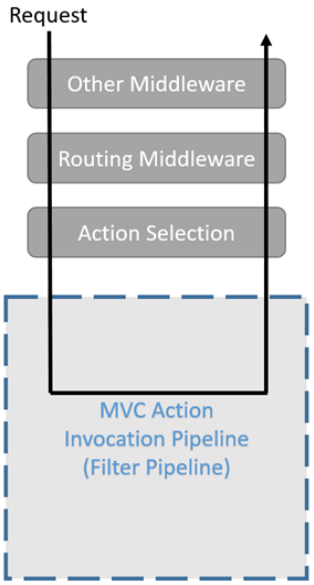
El login es el primer mecanismo para la autenticación. Lo primero que debe hacer un administrador antes de hacer uso de su autorización, es loguearse como tal. Para esto hace uso de sus credenciales, email y password. Tras ingresar esos datos el administrador obtiene o abre una nueva sesión. Mediante esto obtiene un token que, posteriormente, será lo que identifique que un administrador está logueado y puede realizar acciones mediante los endpoints.

2.2. Authorization Filter

Antes de adentrarnos en como funciona nos parece pertinente decir que es interesante sobre los filtro en ASP.NET Core:

Los filtros permiten código correr antes o después de ciertas etapas de la request en la processing pipeline.

Mas especificamente, como se peude ver en al siguiente figura, los filtros se ejecutan en la ASP.NET Core action invocation pipeline. Esta pipeline corre luego que de ASP.NET Core selecciona que accion ejecutar.



Entonces, el token obtenido en la parte previa se debe pasar en el header de la request HTTP a los endpoints que lo necesitan, donde con un filtro se realiza el siguiente procedimiento:

Verificamos que en el header se incluye un token y si en esta cadena de texto se esta representando un Guid, para luego buscar si esta asociado a un administrador en la tabla de sesiones.

En el caso de encontrar un administrador asociado se continua con la ejecución del controlador, en caso contrario se genera un HTTP response con el código de error apropiado y un mensaje descriptivo.

3. Descripción de los códigos de estado HTTP

200 - OK
Es el código estándar para indicar que una request fue exitosa.

201 - Created
Indica que un ítem fue creado.

204 - No Content
Lo utilizamos para mostrar que la request fue exitosa y que no hay informacion que mostrar en el body de la response.

400 - Bad request
Usado cuando se considera que el cliente cometió un error formando la request.

401 - Unauthorized
Lo utilizamos cuando un endpoint que requiere autenticación no es proveído con un token valido en el header de la request.

403 - Forbidden
Indica que aunque el cliente incluyó un token valido en el header de la request este no esta asociado a ningun administrador.

404 - Not Found
Significa que el recurso solicitado no fue encontrado. El ejemplo más simple es cuando se hace un GET a un recurso que no existe.

415 - Unsupported Media Type
Utilizado por el framework ASP.NET Core al detectar un error de cliente donde el mismo rechaza la request porque el formato del payload no es un formato soportado.

500 - Internal Server Error
Indica que el servidor encontró una condición inesperada que la impidió de completar la request. La utilizamos para generar un error amigable al usuario al encontrarnos ante una Exception no esperada.

4. Descripción de los resources de la API

Como se sugería en la letra, utilizamos la herramienta Swagger, si bien la misma puede integrarse al código de nuestra aplicación mediante el paquete de la misma para .Net Core, preferimos utilizarla en su modalidad en la nube. A esta documentación se puede acceder mediante el siguiente link: https://app.swaggerhub.com/apis-docs/juanBalian35/UruguayNatural/1.0.0

Bibliografía

[1] B. Mulloy, *Web API Design, Crafting Interfaces that Developers Love*. APIGEE. [Online]. Available: https://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf

[2] Tutorial REST API. [Online]. Available: www.restfulapi.net/

[3] List of HTTP status codes. [Online]. Available: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes