Universidad ORT Uruguay

Facultad de Ingeniería

Descripción del diseño

Marco Fiorito (227548)
Matias Salles Dulan (201701)
Link al Repositorio en GitHub

26 de noviembre de 2020

Índice general

1. Descripción general y bugs conocidos **2**

2. Diagrama de descomposición de los namespaces del proyecto **3**

3. Diagrama general de paquetes **5**

4. Diagrama de clases por paquete **6**

5. Descripción de jerarquías de herencia utilizadas **25**

6. Modelo de tablas de la estructura de la base de datos **26**

7. Diagramas de secuencia **27**

8. Justificación del diseño **31**
8.1. Patrones GRASP 31
8.2. Principios SOLID 32
8.3. Patrones de diseño utilizados 32
8.4. Algoritmos/Mecanismos que impactan la mantenibilidad 33
8.5. Mecanismo de acceso a datos 33
8.6. Manejo de excepciones 33
8.7. Resumen de mejoras 34

9. Diagrama de implementación **35**

10. Métricas **37**

11. Anexo **41**
11.1. API 41
11.2. Cobertura de pruebas 41
11.3. Angular 43
11.4. Instalación 44

1. Descripción general y bugs conocidos

Nuestra solución a lo planteado en el obligatorio dispone de todas las funcionalidades requeridas, cabe destacar que en el momento de la entrega no hemos encontrado ningún bug o defecto en estas funcionalidades.

El trabajo fue realizado en GitHub siguiendo la metodología TDD y GitFlow, utilizando Pull Requests como herramienta para validar cambios y mantener al equipo sincronizado. Cada Pull Request consta con una descripción y capturas de pantalla si aplica.

En cuanto al idioma en cual desarrollamos nuestra solución, elegimos el Inglés tanto para el código como para los commits y Pull Requests.

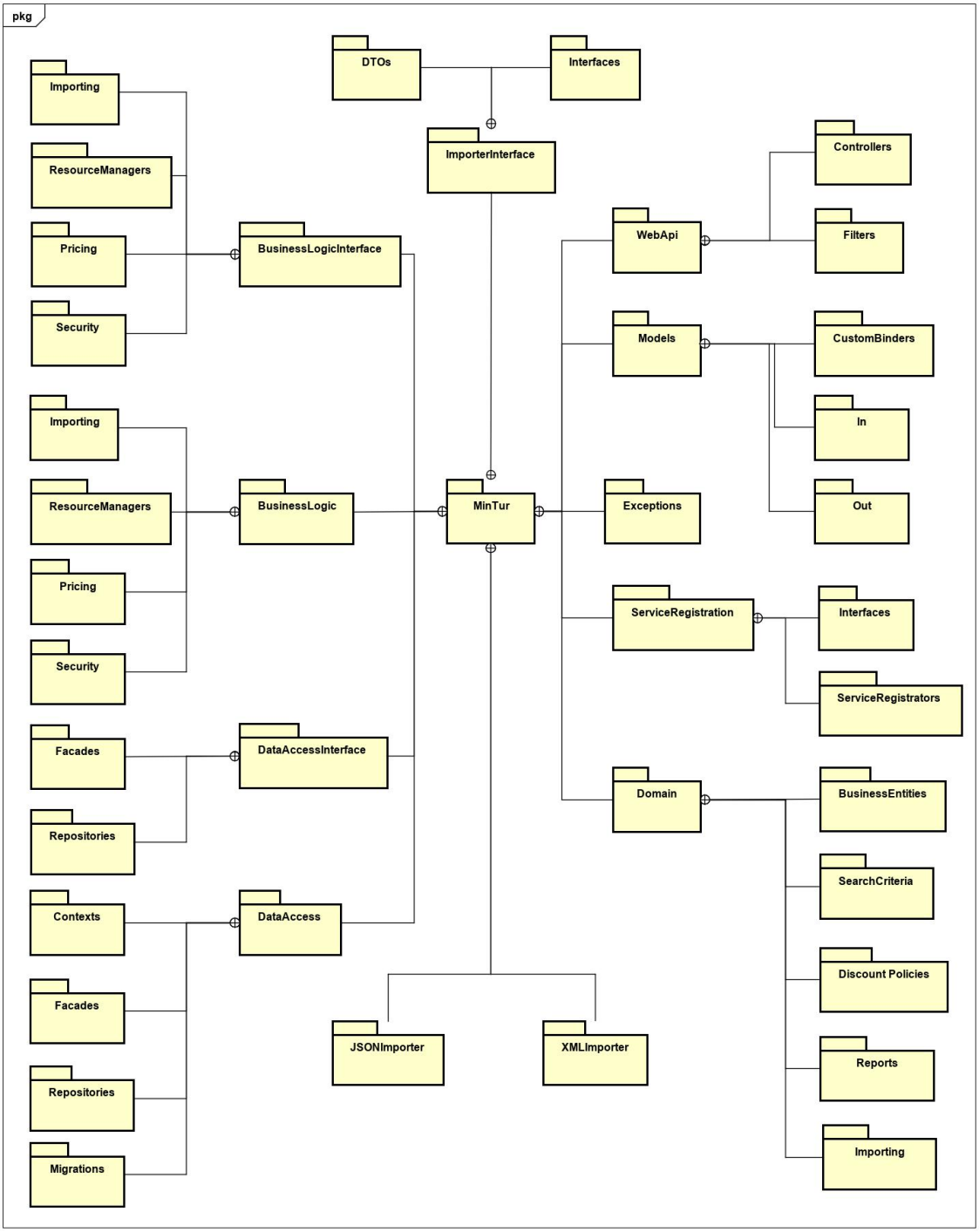
Finalmente, para realizar los diagramas de dicha documentación elegimos Astah Uml, los archivos en caso de no poder visualizar correctamente las imágenes en este documento se encuentran en una carpeta aparte llamada 'Diagramas UML'.

2. Diagrama de descomposición de los namespaces del proyecto

En la siguiente página se muestra la descomposición de los namespaces del proyecto, para esto seguimos el estándar de Microsoft <NombreEmpresa>.<NombreComponente> que se explica en el siguiente link:

Names of Assemblies and DLLs

Para esto tomamos MinTur (abreviación de Ministerio de turismo) como nombre de la empresa seguido del nombre de cada componente de la solución.

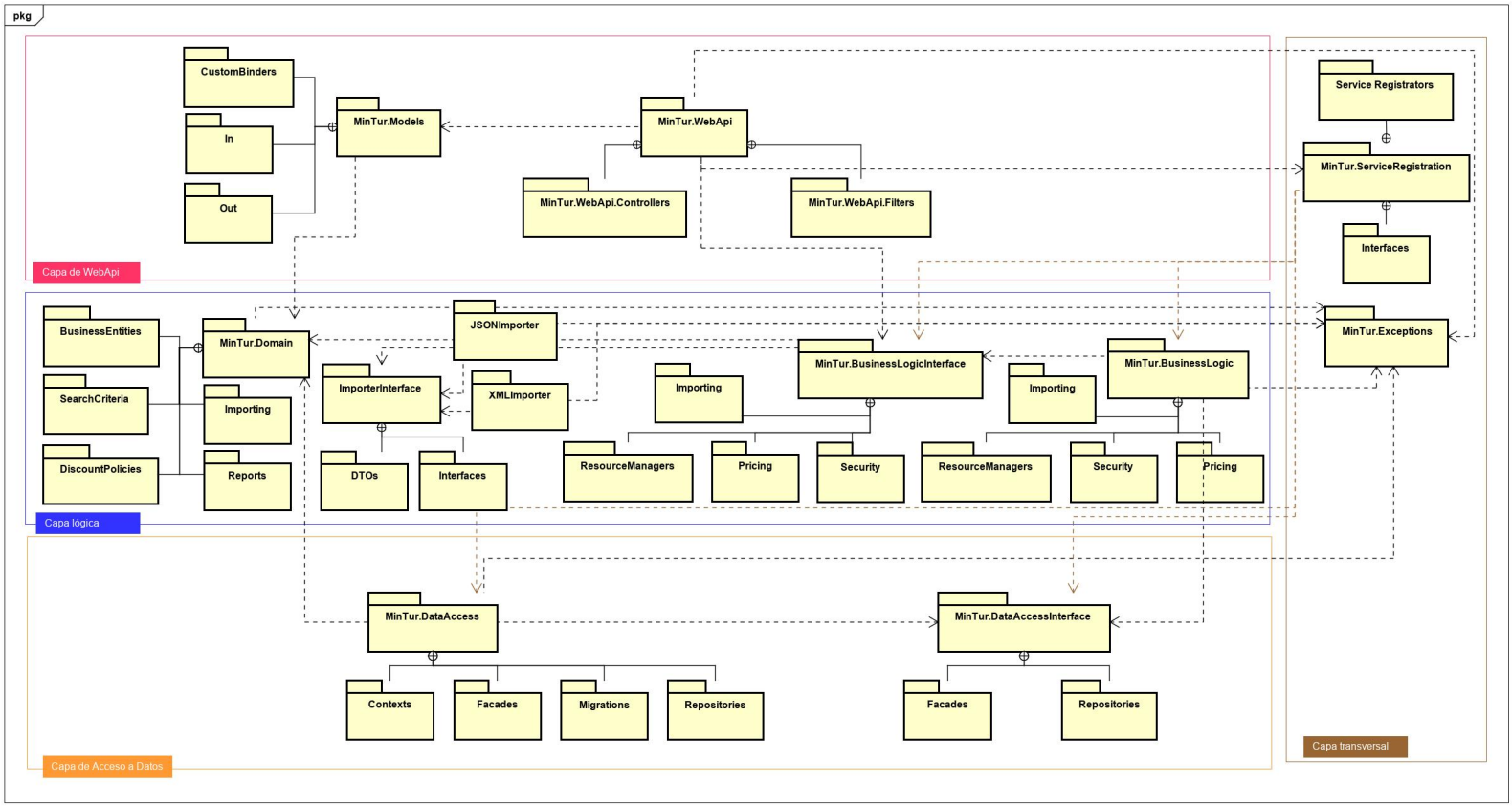

pkg
DTOs
Interfaces
Importing
ResourceManagers
Pricing
Security
ImporterInterface
Controllers
Filters
WebApi
BusinessLogicInterface
Models
CustomBinders
In
Out
Importing
ResourceManagers
Pricing
Security
BusinessLogic
MinTur
Exceptions
ServiceRegistration
Interfaces
ServiceRegistrators
Facades
Repositories
DataAccessInterface
Domain
BusinessEntities
SearchCriteria
Contexts
Facades
Repositories
Migrations
DataAccess
Discount Policies
JSONImporter
XMLImporter
Reports
Importing

3. Diagrama general de paquetes

En la siguiente página se muestra un diagrama general de paquetes dividido por capas, en donde podemos identificar tres capas principales:

- Capa de WebApi - Encargada de recibir los requests y procesarlos
- Capa Lógica - Encargada de exponer los servicios necesarios para poder llevar a cabo los requerimientos
- Capa de Acceso a datos - Encargada de acceder, retornar y realizar el mantenimiento de los datos de nuestro sistema.

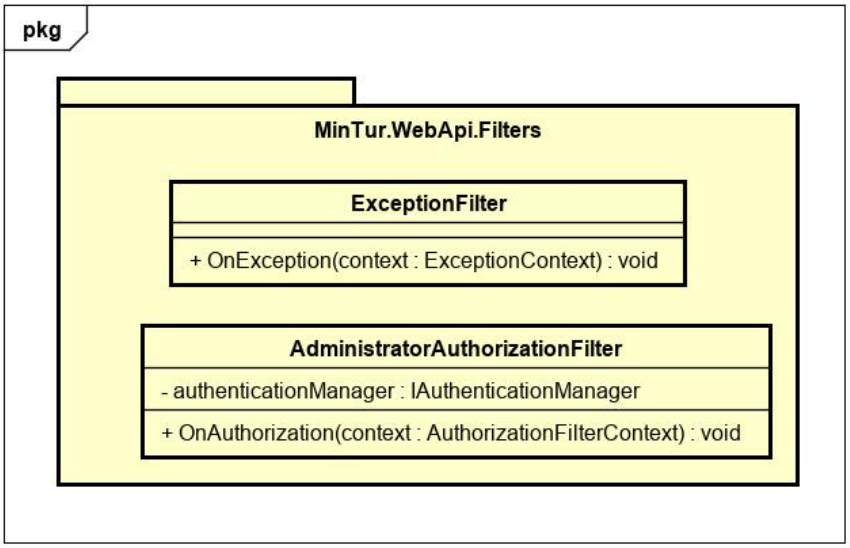
Finalmente tenemos una capa transversal la cual no cae dentro de las anteriores ya que o son de uso común o su responsabilidad es distinta, en el próximo capitulo se ira paquete por paquete explicando la responsabilidad que tiene cada uno.



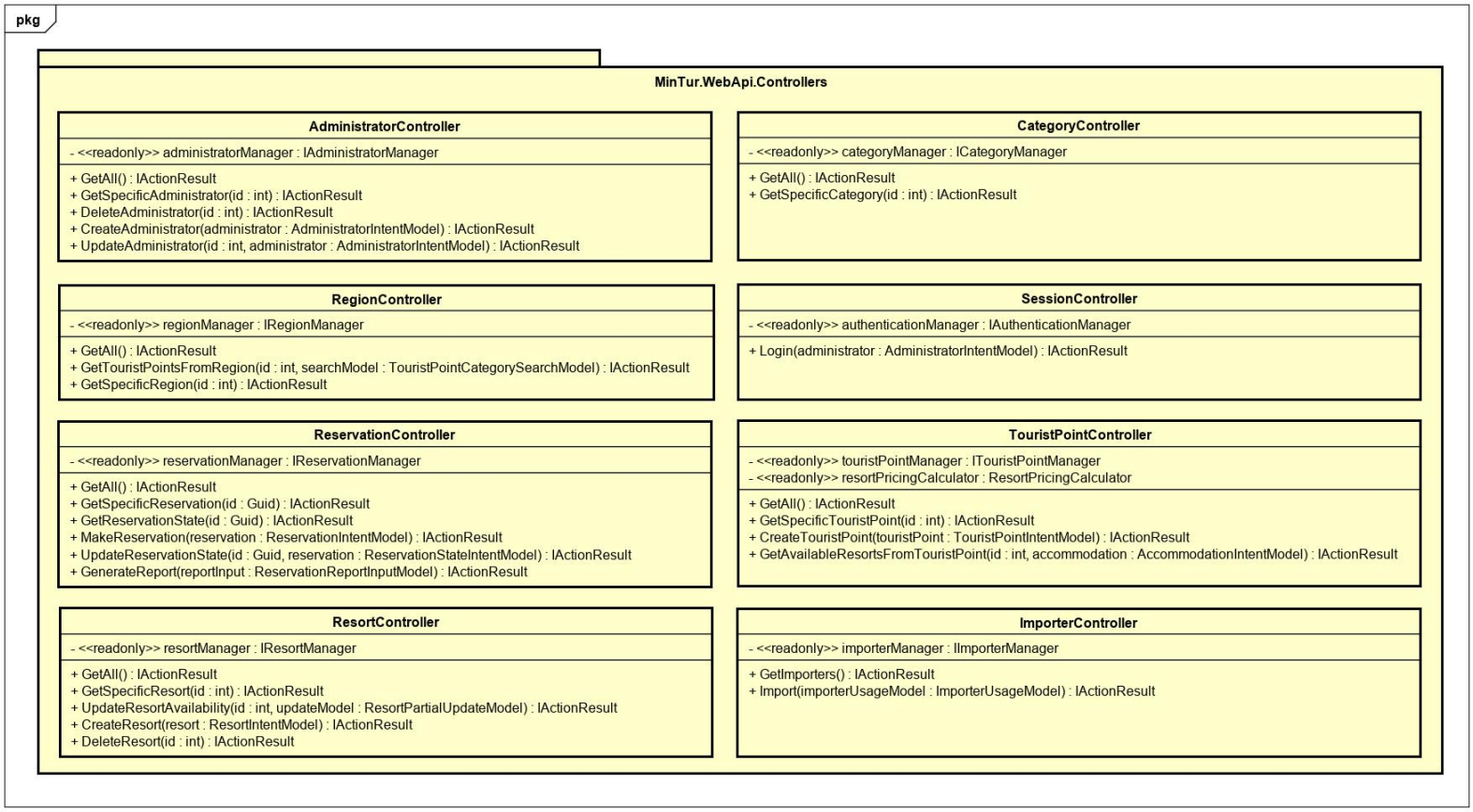
4. Diagrama de clases por paquete

A continuación iremos paquete por paquete explicando brevemente sus responsabilidades e incluyendo un diagrama de clases asociado, esto lo haremos desde la capa mas alta (WebApi) hasta la capa de Acceso a Datos y finalmente pasaremos por la capa transversal. Los paquetes de Test no llevan diagramas asociados.

- MinTur.WebApi: Este paquete es el encargado de exponer los endpoints, 'levantar' el sistema y tanto procesar, como denegar y armar las respuestas para las consultas. Para esto tiene tres paquetes dentro:
  - Controllers: Encargado de exponer los endpoints y procesar las consultas, aquí se encuentra un controller por recurso. **Se agrego un método en Reservation Controller y un controlador nuevo (Importer Controller).**
  - Filters: Encargado de filtrar las consultas antes que lleguen a los controladores y capturar las excepciones generadas por el sistema. **No cambio.**
  - Test: Encargado de testear tanto los filtros como los controladores



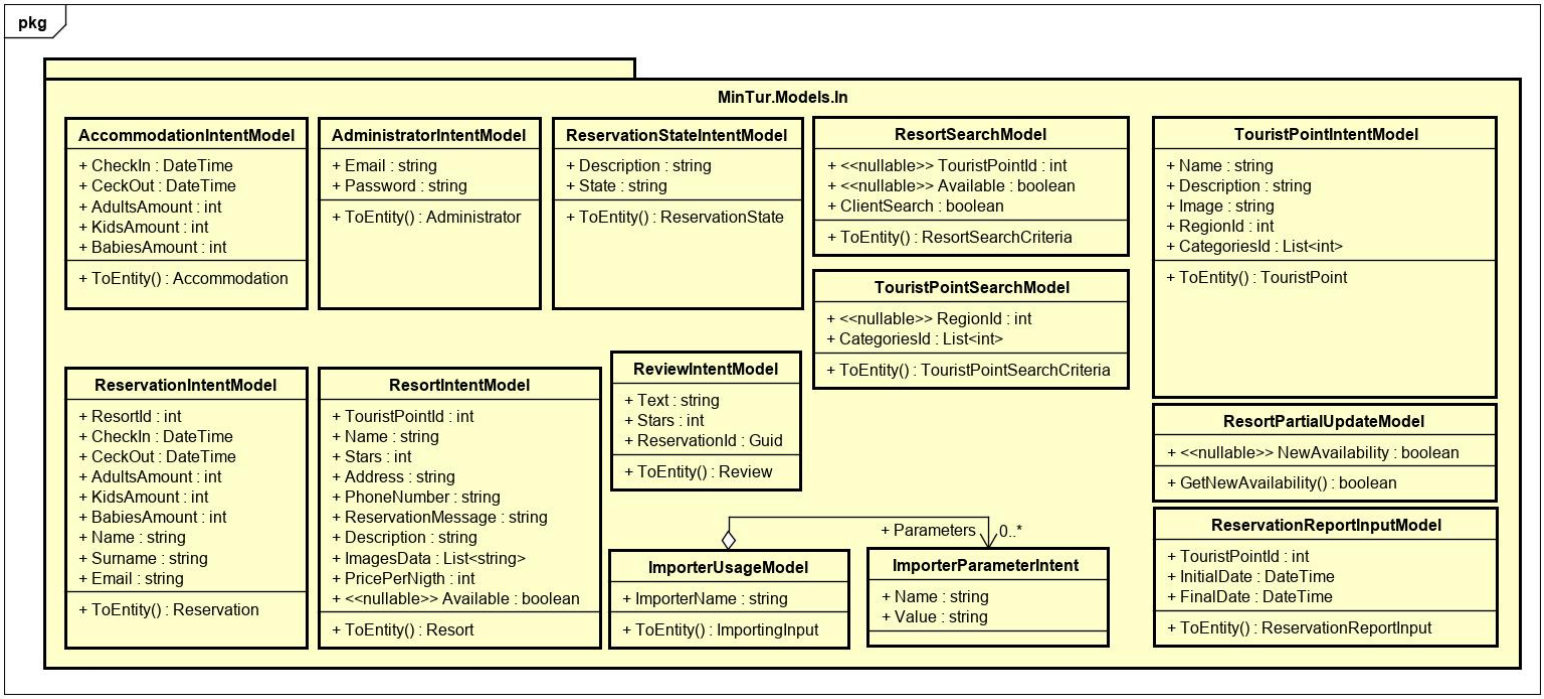
Paquete MinTur.WebApi.Filters



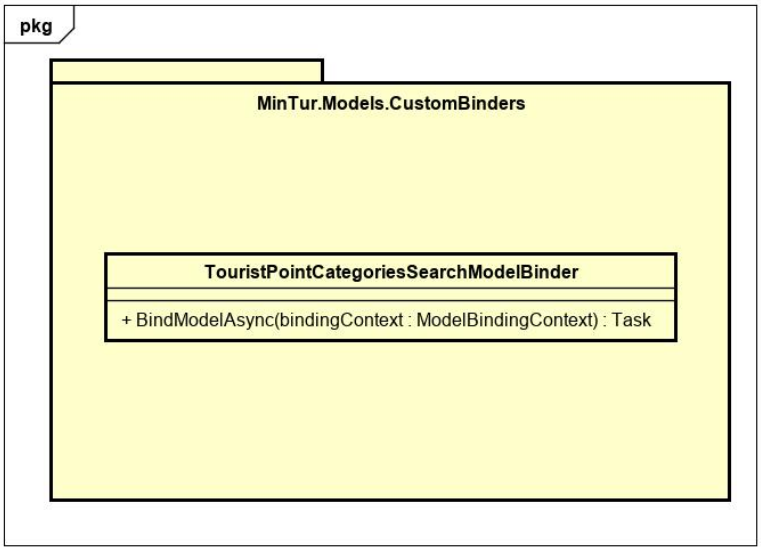
Paquete MinTur.WebApi.Controllers

- MinTur.Models: Este paquete es el encargado de exponer las clases que la WebApi va a utilizar para recibir y enviar información al cliente, escondiendo información en caso de ser necesario y brindando un mecanismo para transformar entidades de dominio en modelos y vice versa. Para esto tiene los siguientes paquetes dentro:

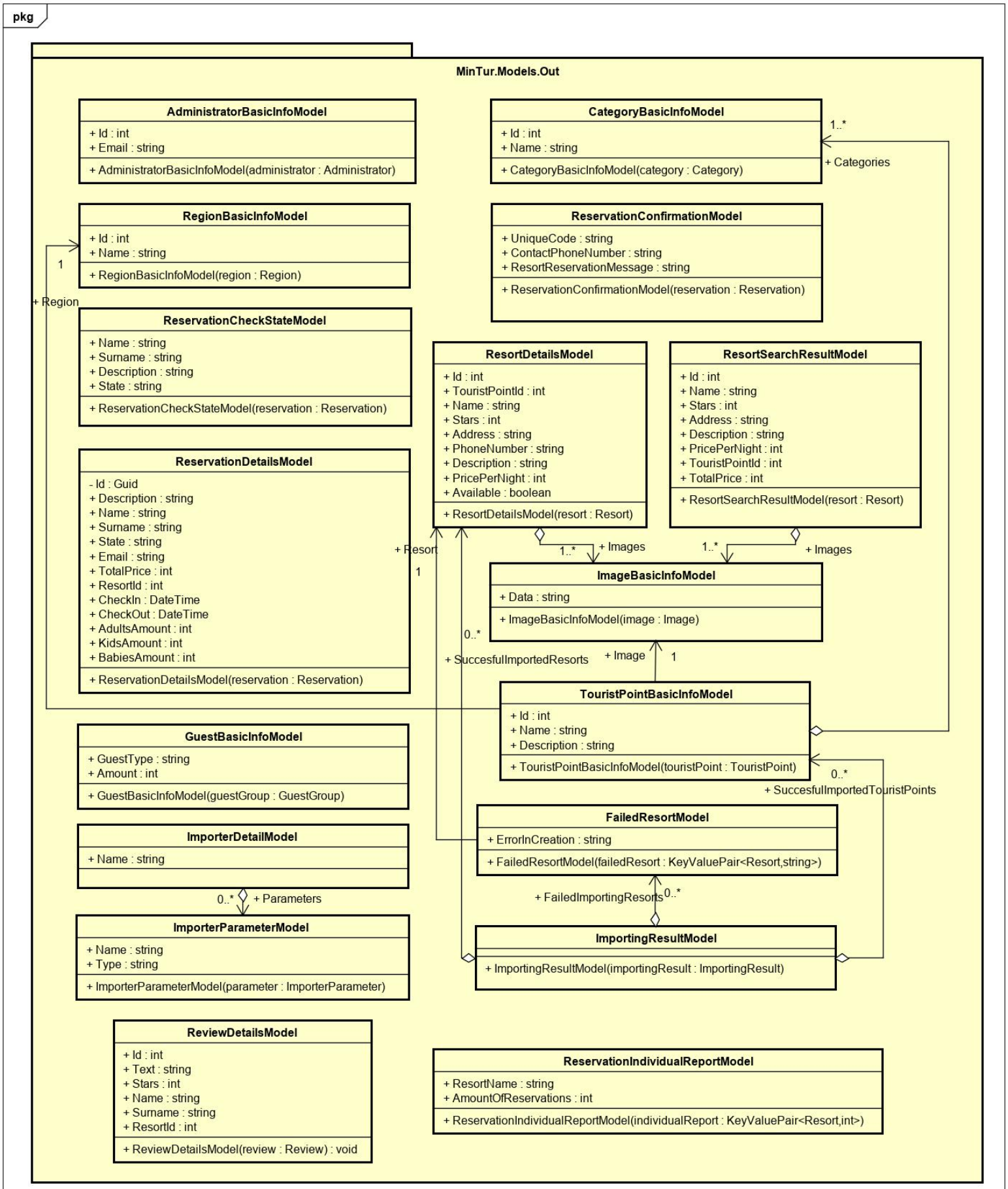
  - CustomBinders: Define Custom Model Binders utilizados en ciertos modelos para brindar servicios de binding distintos a los que trae ASP.NET por defecto. Principalmente utilizado para pasar los Id's de las categorías en un solo parámetro separados por un guión '-'. **No cambio**
  - In: Define los modelos de entrada, estos se utilizaran para recibir información enviada por el cliente y transformarlo en una entidad de nuestro dominio. **Se agregaron modelos.**
  - Out: Define los modelos de salida los cuales se crean a partir de las entidades de dominio asociadas, estos se encargan de mostrar solo la información necesaria al cliente. **Se agregaron modelos.**
  - Test: Encargado de testear tanto los modelos de salida y entrada como los custom binders (no se incluye diagrama de este paquete).



Paquete MinTur.Models.In

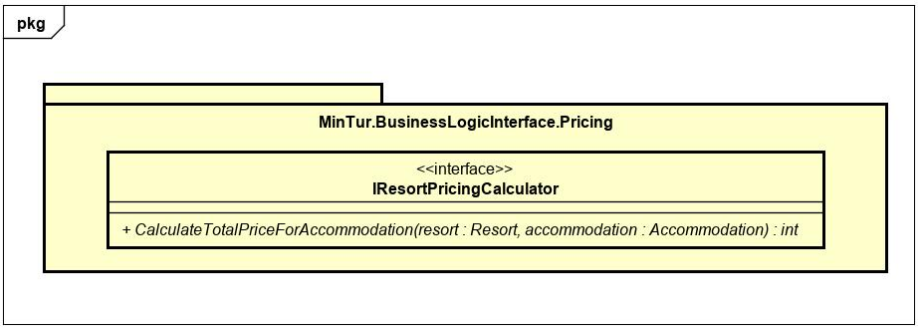


Paquete MinTur.Models.CustomBinders

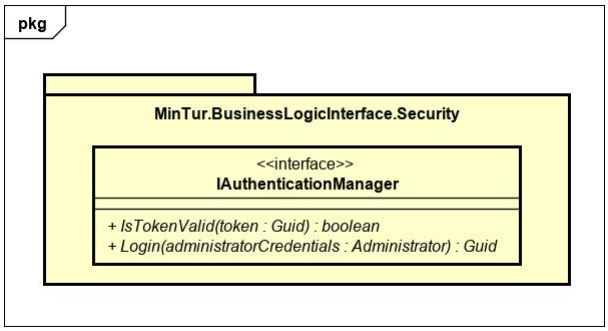


Paquete MinTur.Models.Out

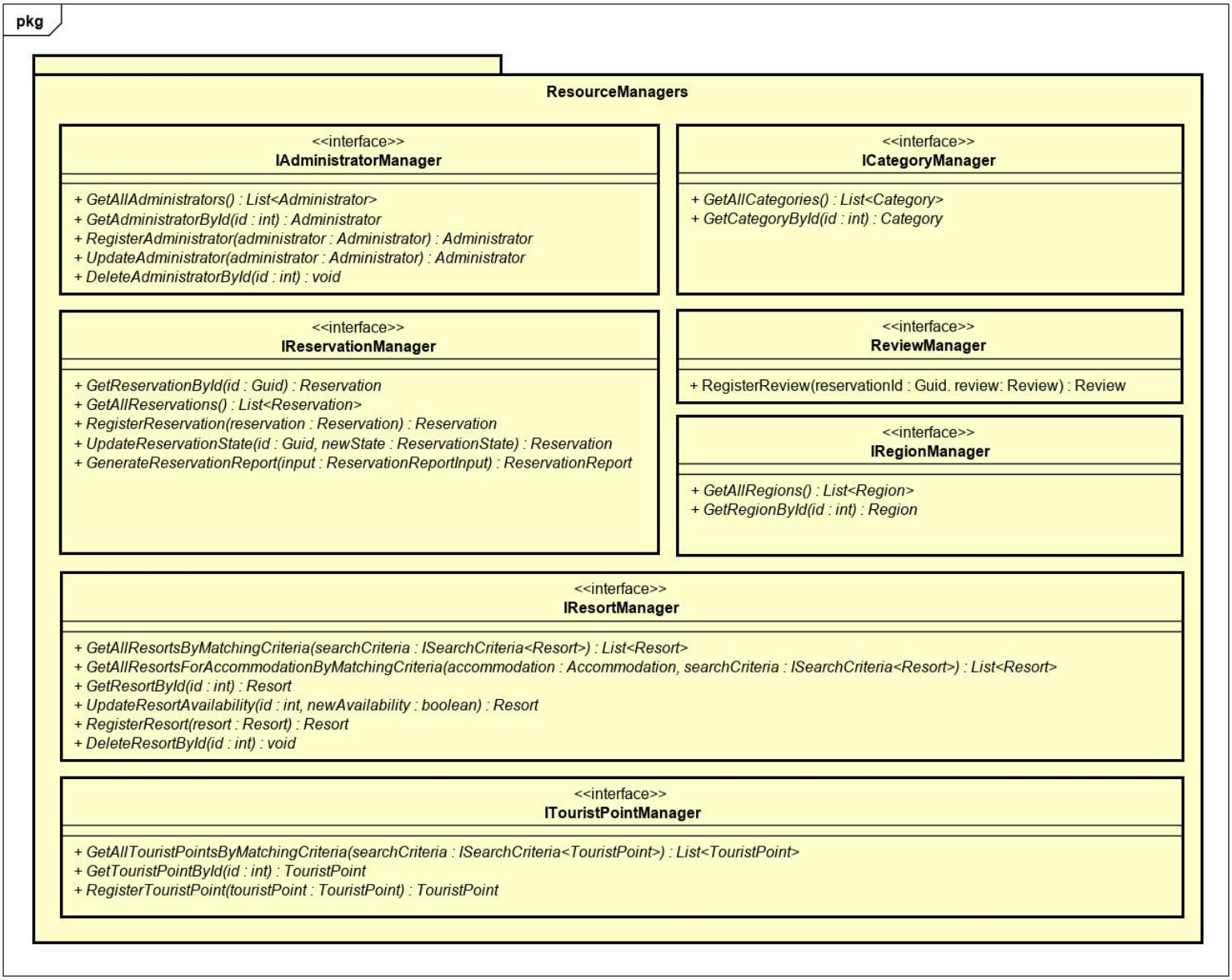
- MinTur.BusinessLogicInterface: Este paquete es el encargado de exponer las interfaces necesarias para resolver los distintos casos de uso. Para esto se separaron las interfaces en tres paquetes internos diferenciados por 'grupos de funcionalidades'.
    - ResourceManagers: Define todas las interfaces que tienen que ver con el manejo de los recursos (obtención, alta, baja y modificación). **Se agrego un Review Manager, un método a Reservation Manager y se unificaron los endpoints que no cumplían REST mediante SearchCriterias.**
    - Pricing: Define todas las interfaces que tienen que ver con el calculo de precios. **No cambio.**
    - Security: Define todas las interfaces que tienen que ver con la seguridad/sesiones en nuestro sistema. Como se puede ver en el diagrama, aquí optamos por crear una clase IAuthenticationManager al igual que se usa en Laravel para realizar login y validar si un token es válido. **No cambio.**
    - Importing: Define un manejador para visualizar y utilizar los importadores, así como un adapter para poder utilizar la interfaz de IImporter, la cual fue definida de forma tal que el tercero solo necesite conocer el paquete ImportingInterface y Exceptions. **Nuevo paquete.**



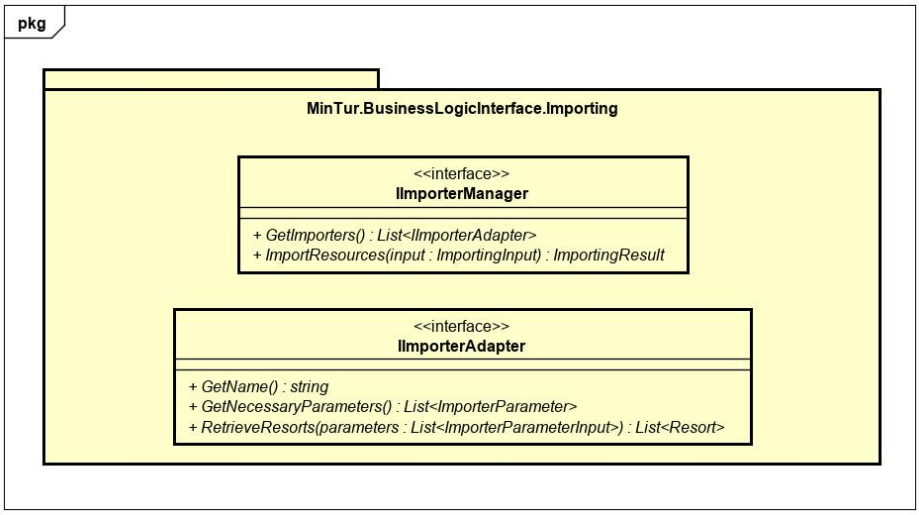
Paquete MinTur.BusinessLogicInterface.Pricing



Paquete MinTur.BusinessLogicInterface.Security

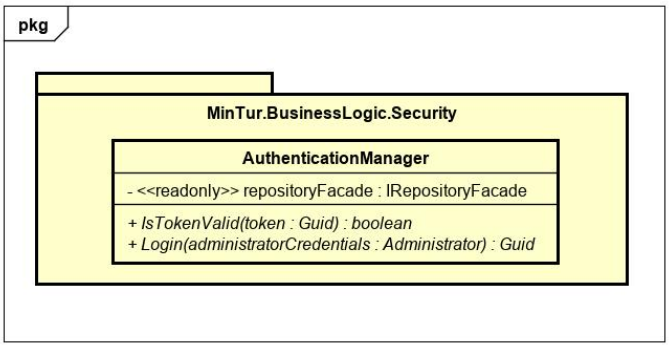


Paquete MinTur.BusinessLogicInterface.ResourceManagers

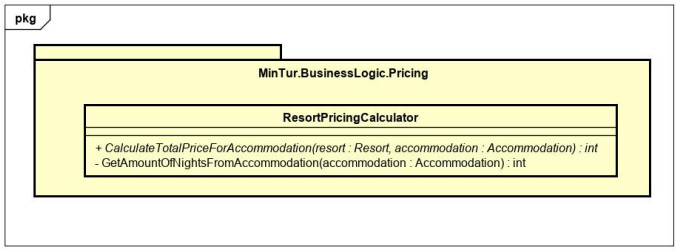


Paquete MinTur.BusinessLogicInterface.Importing

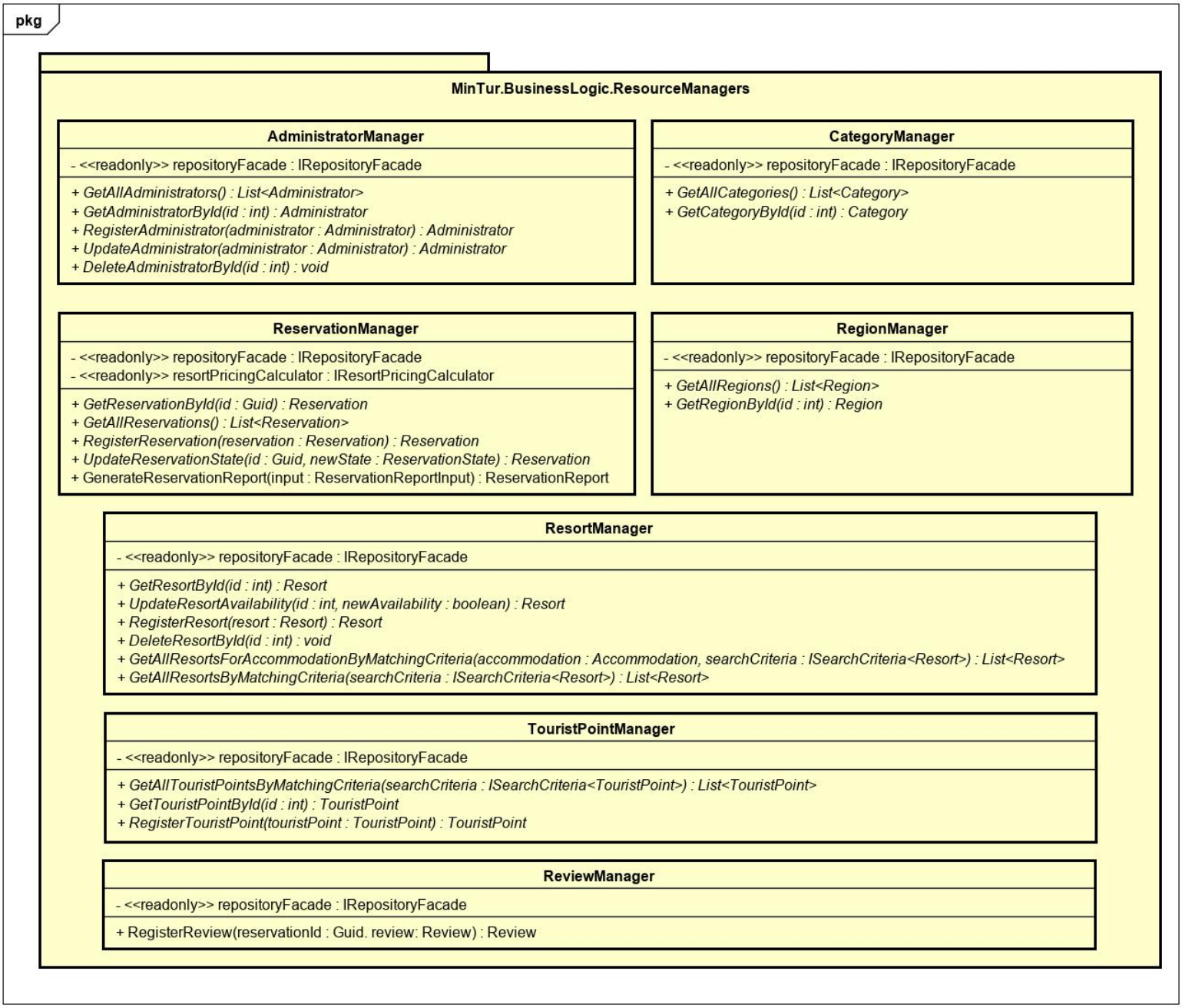
- MinTur.BusinessLogic: Este paquete es el responsable de implementar las interfaces que expone MinTur.BusinessLogicInterface, el cual luego puede ser registrado para ser utilizado mediante inyección de dependencias. Para esto tenemos los mismos paquetes internos que MinTur.BusinessLogicInterface más uno para los tests. **Implementamos lo nuevo de Business Logic Interface y cambio la forma en la cual esta implementada Resort Pricing Calculator.**



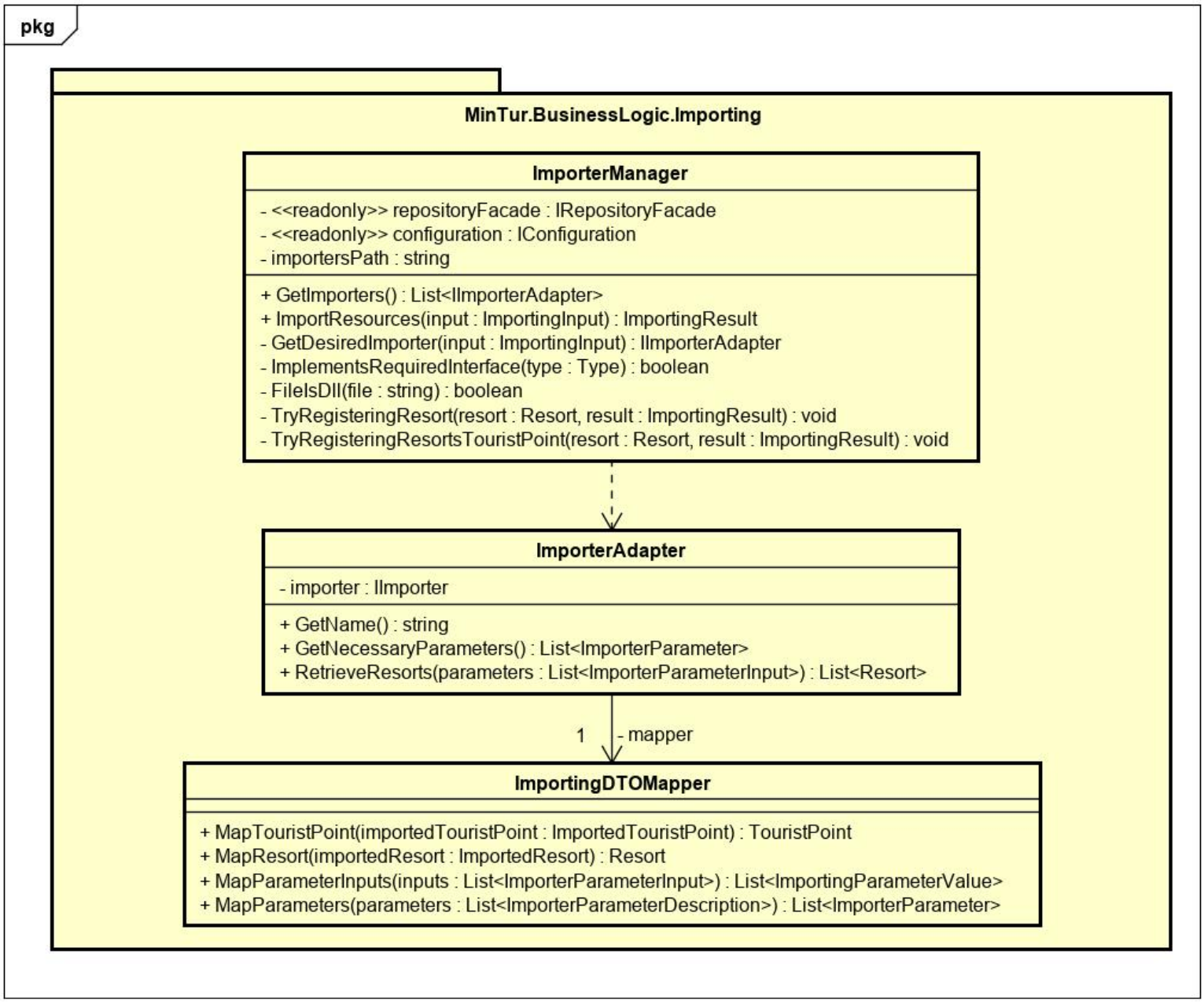
Paquete MinTur.BusinessLogic.Security



Paquete MinTur.BusinessLogic.Pricing

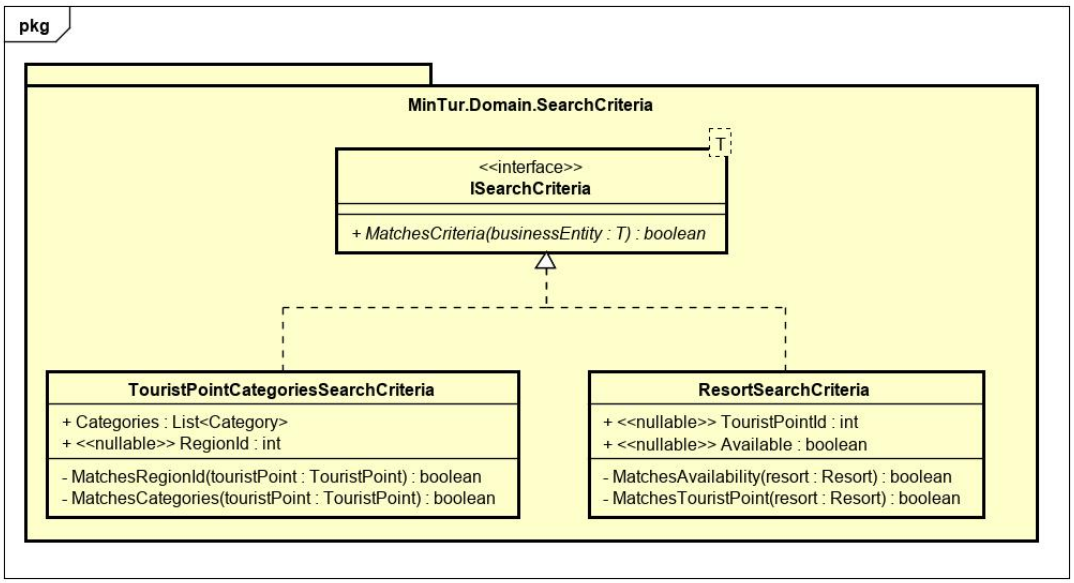


Paquete MinTur.BusinessLogic.ResourceManagers

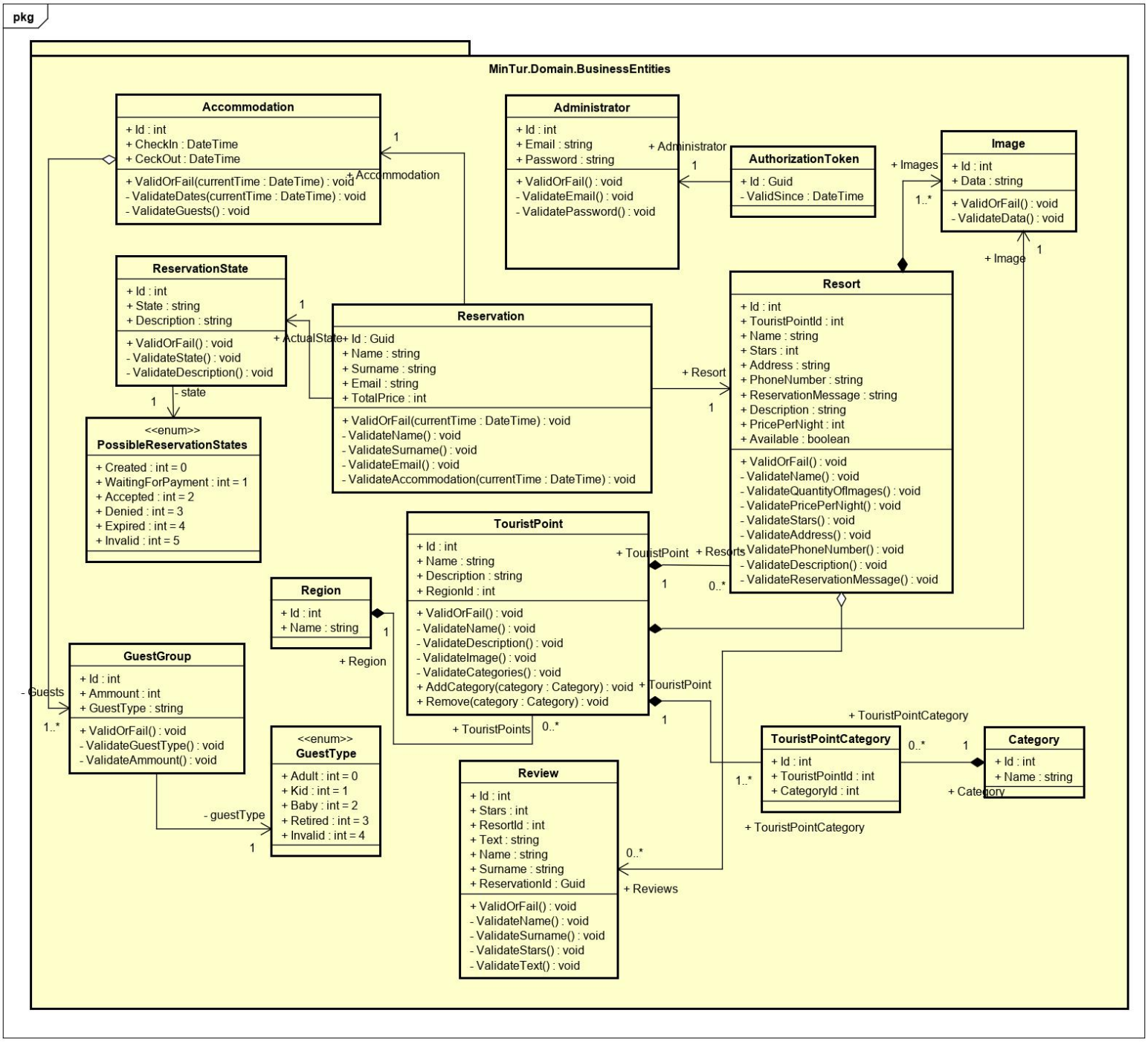


Paquete MinTur.BusinessLogic.Importing

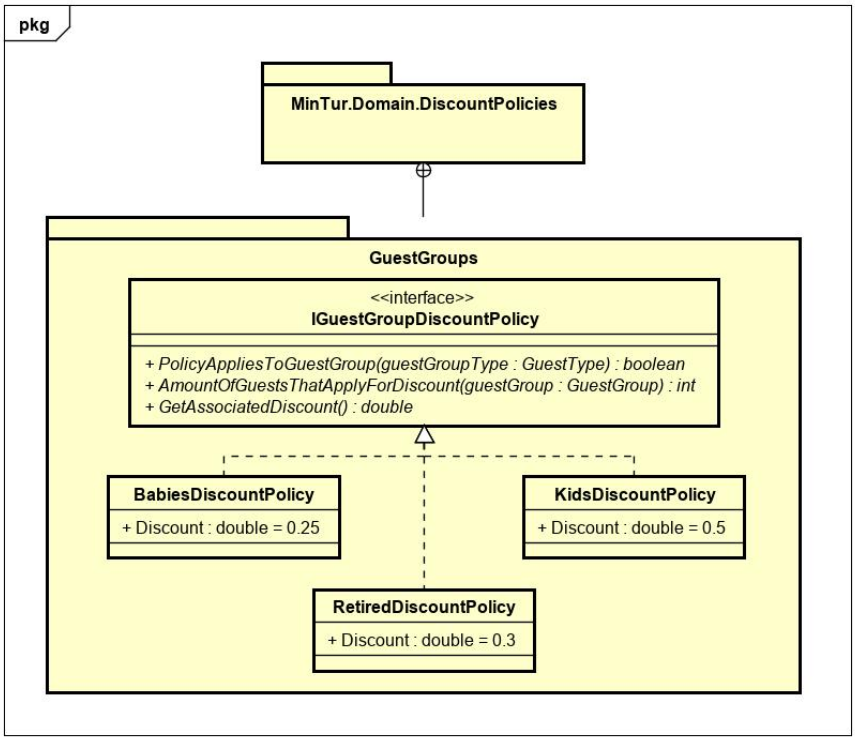
- MinTur.Domain: Este paquete es el responsable de definir las entidades con las cuales va a operar el sistema, aquí tenemos los siguientes paquetes internos:
    - BusinessEntities: Encargado de definir las entidades que representan el negocio y de su auto-validación. **Se agregaron clases para el manejo de huéspedes genéricos y reseñas.**
    - SearchCriteria: Encargado de definir los criterios de búsqueda que existen en el sistema y sus comportamientos. **Para unir los endpoints que no cumplían con las buenas practicas de REST redefinimos el criterio de búsqueda existente y agregamos otro.**
    - Reports: Encargado de definir los reportes que se pueden generar con nuestra aplicación, así como los inputs de ellos. **Nuevo paquete.**
    - Discount Policies: Encargado de definir las políticas de descuento para toda la aplicación, actualmente solo dispone de políticas de descuento para los grupos de huéspedes, pero se puede extender a politicas de descuento de los hospedajes, puntos turísticos, etc. **Nuevo paquete.**
    - Importing: Encargado de definir todas las clases necesarias para trabajar con los importadores de forma genérica. **Nuevo paquete.**
    - Test: Encargado de testear tanto BusinessEntities como SearchCriteria



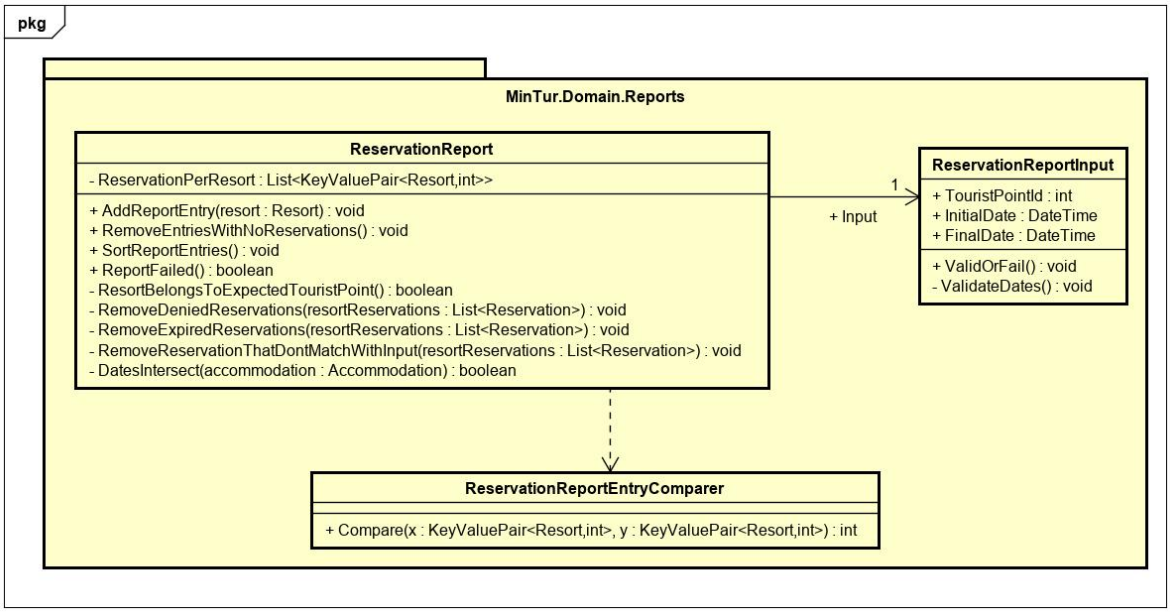
Paquete MinTur.Domain.SearchCriteria



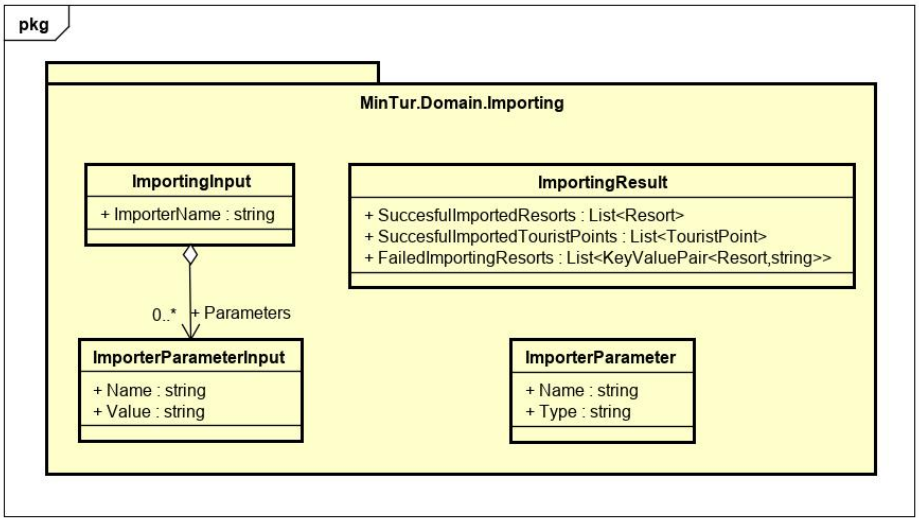
Paquete MinTur.Domain.BusinessEntities



Paquete MinTur.Domain.DiscountPolicies

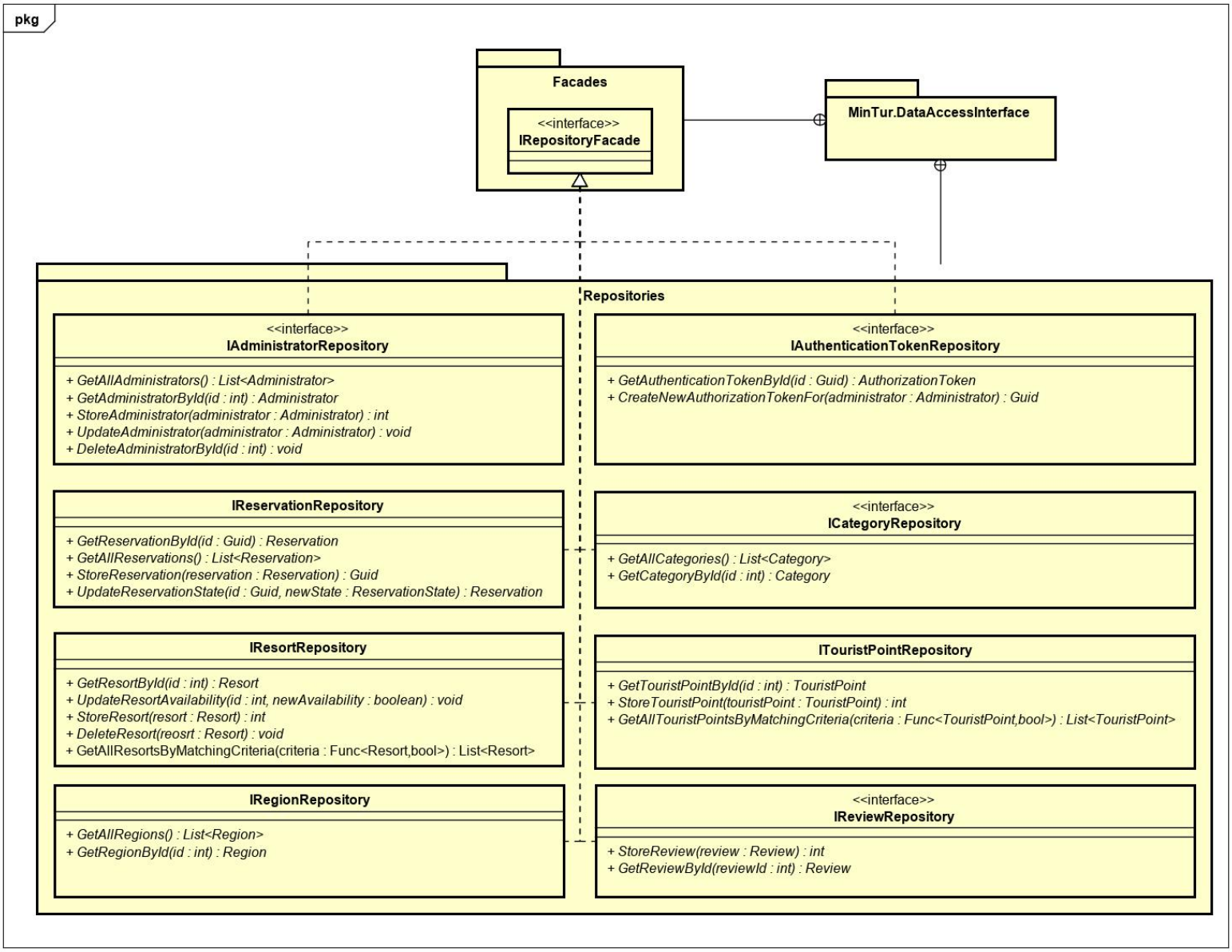


Paquete MinTur.Domain.Reports



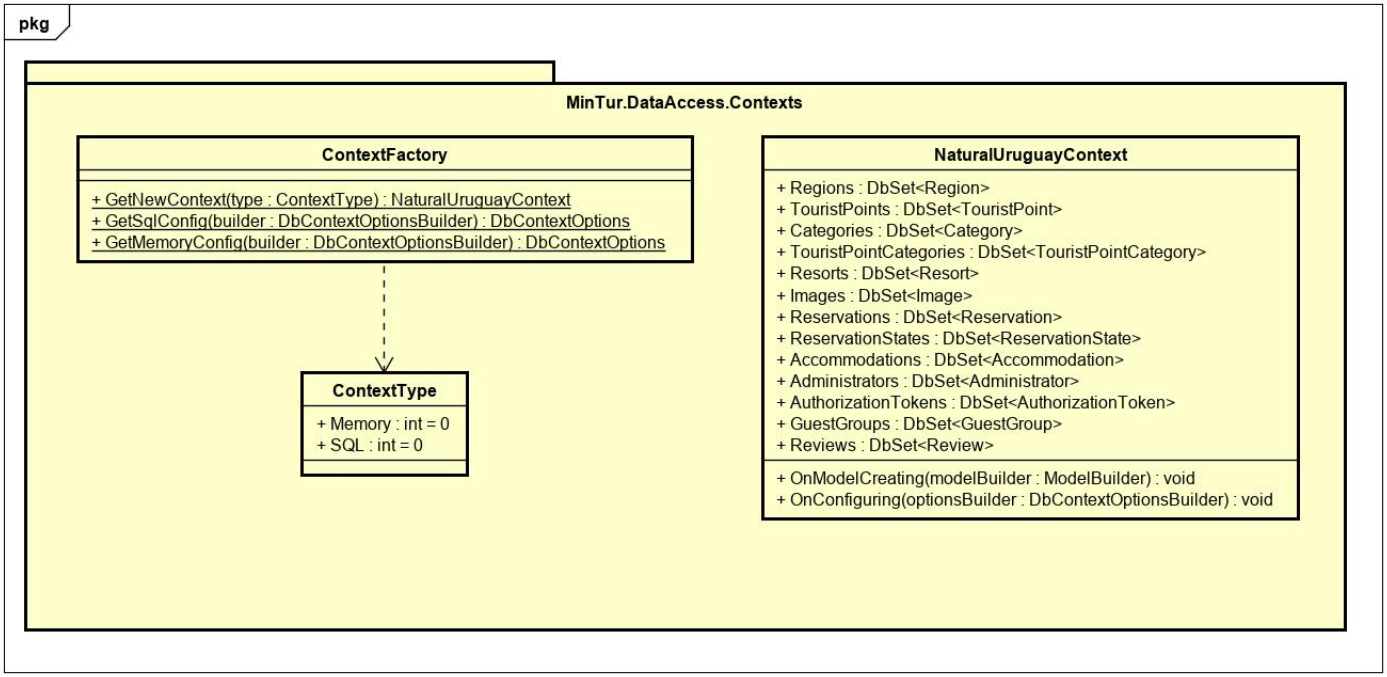
Paquete MinTur.Domain.Importing

- MinTur.DataAccessInterface: Este paquete es el encargado de exponer las interfaces necesarias para brindar acceso a los datos de nuestro sistema. Este paquete se compone de los siguientes paquetes internos
  - Facades: Define fachadas para brindar un único punto de acceso y evitar la necesidad de pasar por parámetro varios repositorios en los constructores. **Se le agrego el Review Repository.**
  - Repositories: Define todas las interfaces de los repositorios, estos brindan el acceso a los datos asociados a una entidad en particular. **Se agrego un Review Repository y se cambio en el repositorio de Region, TouristPoint y Resort los métodos Get, siguiendo el lineamiento con los nuevos Search Criteria para unir los endpoints del obligatorio anterior.**

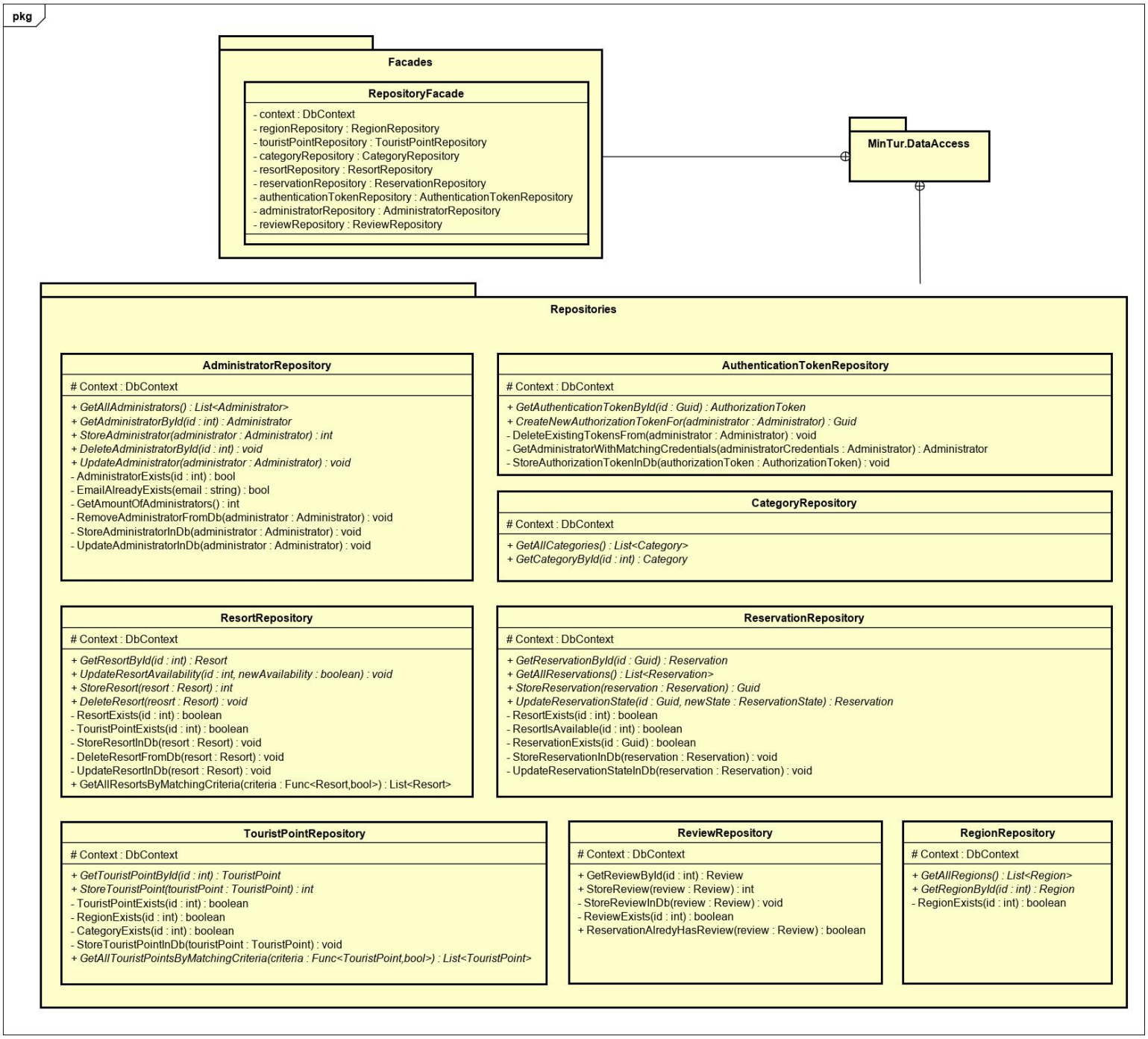


Paquete MinTur.DataAccessInterface.Facades y .Repositories

- MinTur.DataAccess: Este paquete es el responsable de implementar las interfaces que expone MinTur.DataAccessInterface, el cual luego puede ser registrado para ser utilizado mediante inyección de dependencias. Para esto tenemos los mismos paquetes internos que MinTur.DataAccessInterface más los siguientes:
  - Migrations: Donde se encuentran todas las migraciones realizadas en el transcurso de la entrega (no se incluye diagrama para este paquete). **Se agregaron nuevas migraciones.**
  - Contexts: Donde definimos el contexto que se utilizará para acceder a nuestra base de datos. **Se agregaron nuevos DbSets.**
  - Repositories: Donde implementamos el paquete equivalente en DataAccessInterface. **Se implementaron los cambios de las interfaces.**
  - Facades: Donde implementamos el paquete equivalente en DataAccessInterface. **Se implementaron los cambios de las interfaces.**



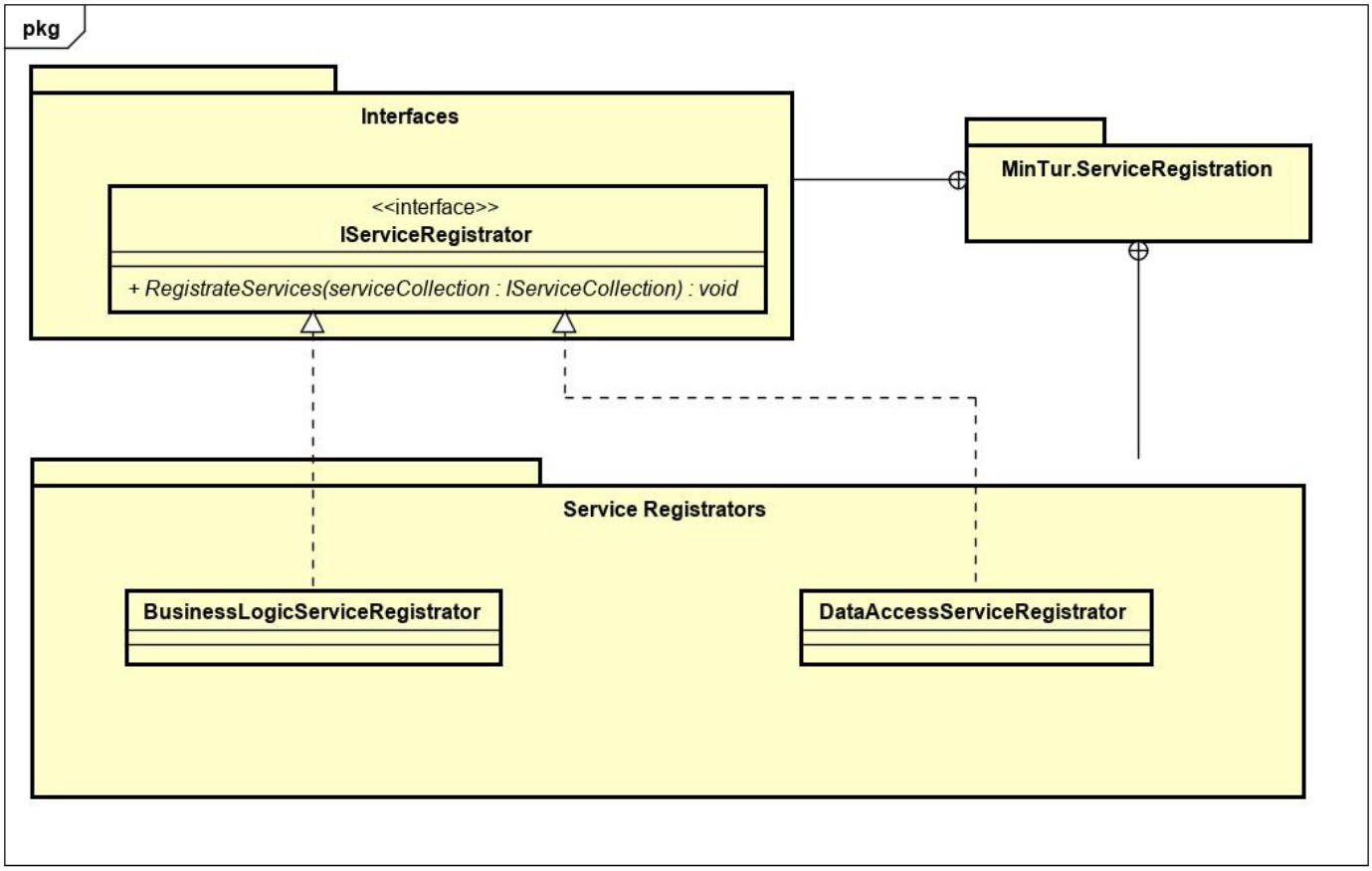
Paquete MinTur.DataAccess.Contexts



Paquete MinTur.DataAccess.Facades y Repositories

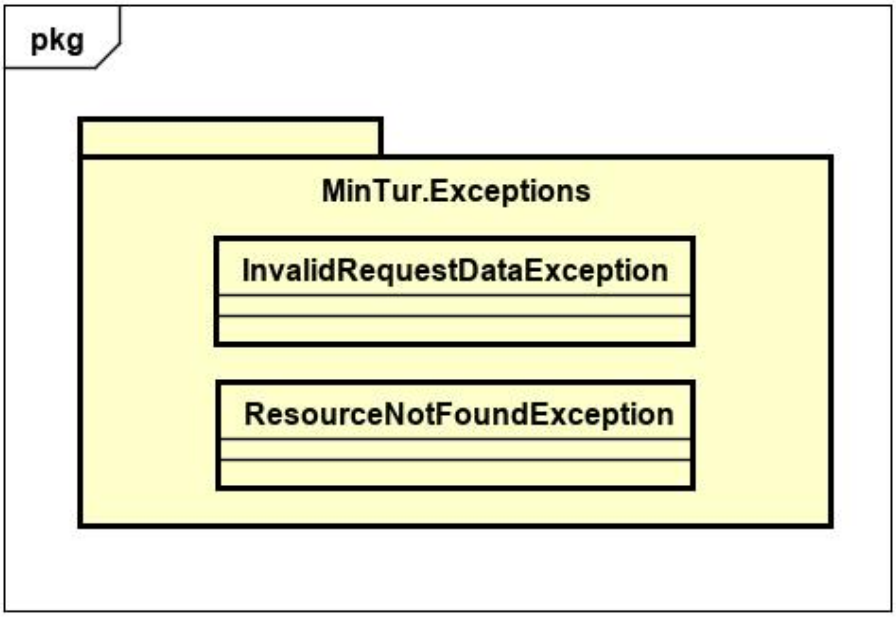
- MinTur.ServiceRegistration: Es el encargado de definir todos los registradores de servicios los cuales derivan de una interfaz especifica, para luego ser creados dinámicamente en Startup.cs mediante reflection y así registrar todos los servicios que el sistema necesita. Para esto tenemos los siguientes paquetes internos:
    - Interfaces: Donde se define la interfaz de la cual todos los registradores de servicio deberán implementar. **No cambio.**
    - ServiceRegistrators: Donde se encuentran todos los registradores de servicios (uno por componente). **No cambio.**

Cabe destacar que esta solución esta inspirada en el funcionamiento del framework Laravel, en donde uno debe crear un ServiceProvider por cada servicio que crea, este hereda de IServiceProvider e ingresa al contenedor de servicios lo necesario. Finalmente el framework automáticamente cada vez que se 'levanta' crea y corre todos los ServiceProviders.



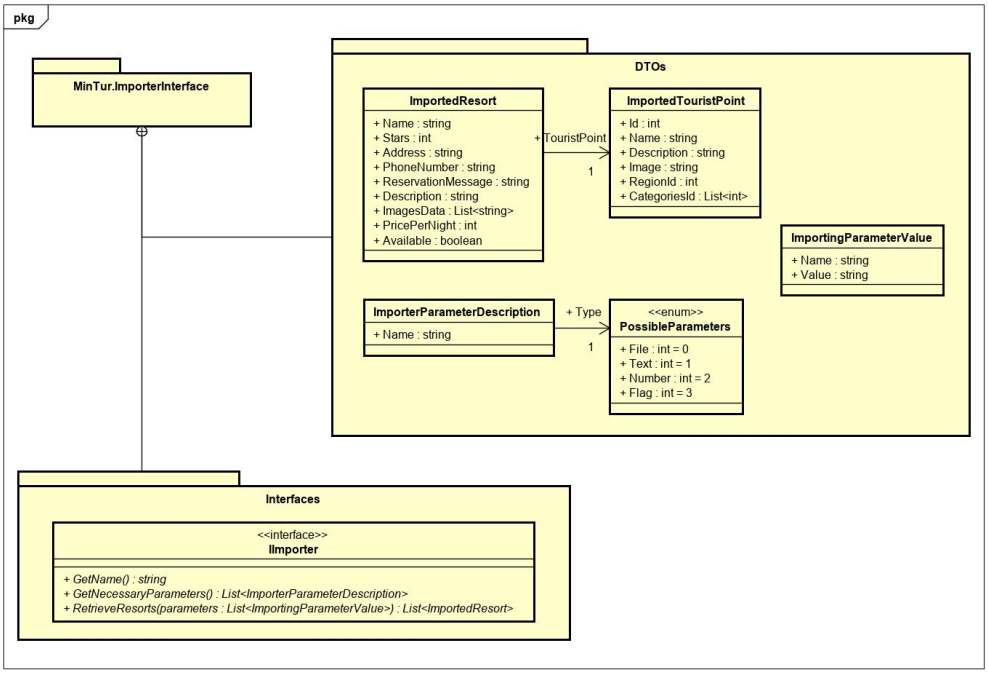
Paquete MinTur.ServiceRegistration.Interfaces y .ServiceRegistrators

- MinTur.Exceptions: Es el encargado de definir las excepciones que se van a utilizar en nuestro sistema. Estas deben ser genéricas y reusables, pero también representar claramente el dominio en el cual estamos trabajando. **No cambio.**



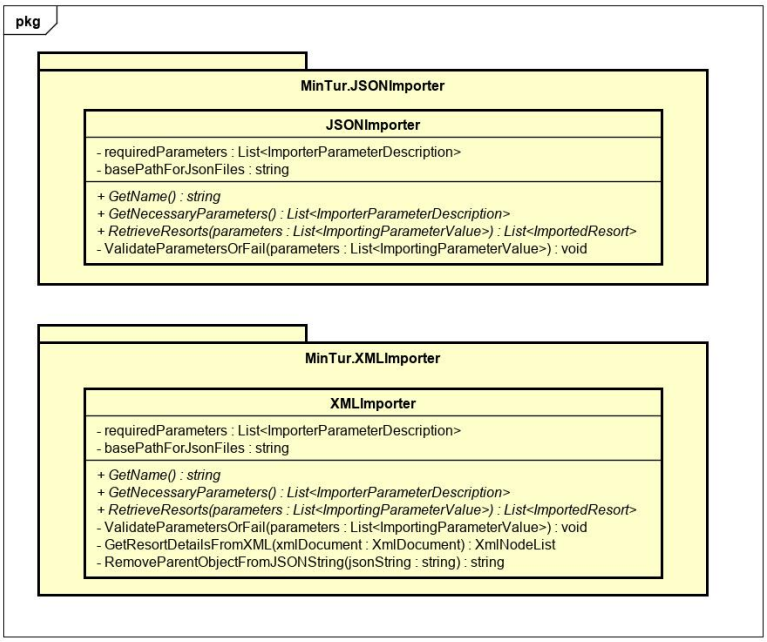
Paquete MinTur.Exceptions

- MinTur.ImporterInterface: Es el encargado de definir las interfaces y los DTOs a utilizar de forma tal de lograr un interfaz extensible para que cualquier tercero pueda agregar importadores de distinto tipo. Los importadores se identifican por nombre y los parámetros permiten la flexibilidad de estos, luego se dispone de un método en el cual el importador devuelve los hospedajes a guardar en la base de datos. **Nuevo paquete.**



Paquete MinTur.ImporterInterface

- MinTur.JSONImporter/XMLImporter: Son los importadores especificos pedidos en esta entrega. El paquete instalado Newtonsoft no solo maneja json sino que puede manejar json a partir de XML, por lo que fue utilizado en ambos importadores. **Nuevos paquetes.**



Paquete MinTur.JSONImporter y MinTur.XMLImporter

5. Descripción de jerarquías de herencia utilizadas

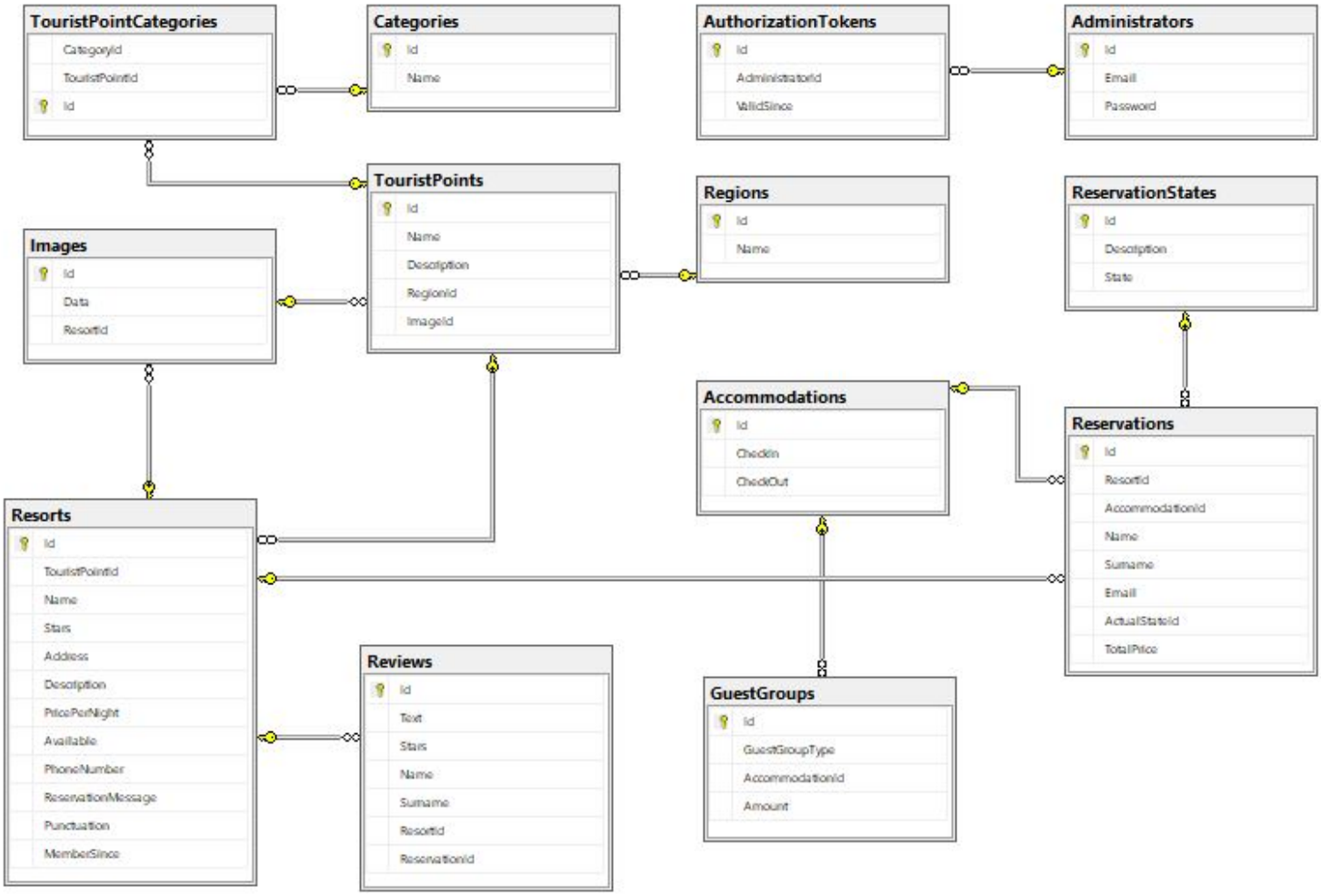
En el desarrollo de nuestra solución evaluamos aplicar herencia en los administradores utilizando una clase abstracta usuario para luego poder extender el sistema con turistas y otros tipos de usuario.
Esto fue descartado debido al tiempo acotado de la entrega y falta de claridad en las diferencias entre un administrador y otro tipo de usuario (ej: turista) ya que ambos deberían tener un email y una contraseña, por lo que nos encontraríamos con clases concretas vacías, sin atributos que la diferencien.

Fuera de lo mencionado, no se utilizo 'herencia' como tal en nuestra solución, si creamos varias interfaces que ayudan a la claridad, reutilización del código y la extensibilidad como en:

- MinTur.ServiceRegistration con IServiceRegistrator
- MinTur.Domain.SearchCriteria ISearchCriteria
- MinTur.Domain.DiscountPolicies IGuestGroupDiscountPolicy **Nuevo**
- MinTur.DataAccessInterface con IRepositoryFacade

6. Modelo de tablas de la estructura de la base de datos

A continuación se incluye el modelo de tablas de la estructura de la base de datos diseñada en el transcurso de la entrega.



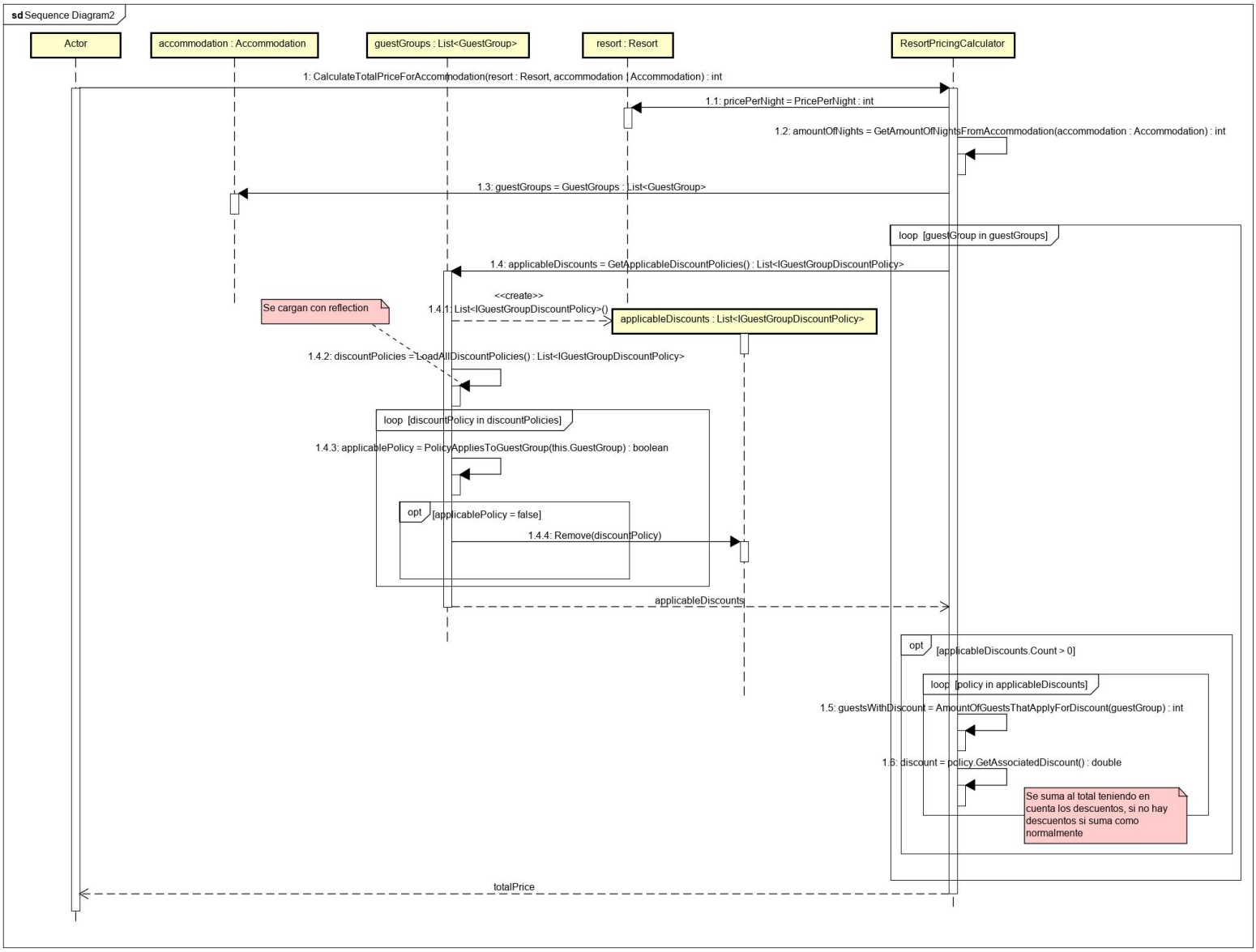
7. Diagramas de secuencia

A continuación se incluyen los diagramas de secuencia para las tres funcionalidades que consideramos tanto interesantes como variadas, estas son:

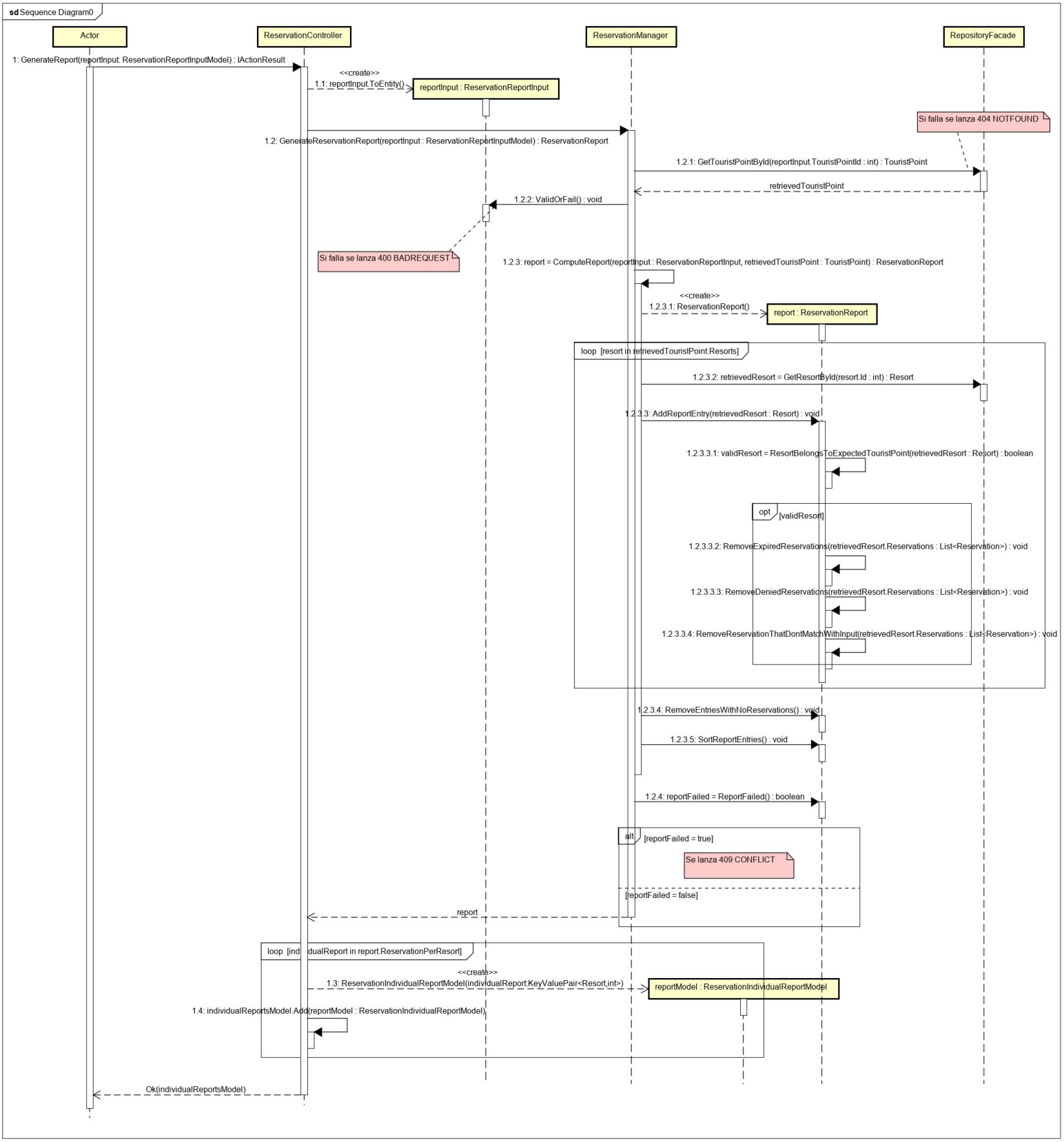
- Calcular el precio de parte del ResortPrincingCalculator **Nuevo**
- Obtener todos los importadores **Nuevo**
- Generar reporte A **Nuevo**

Cada diagrama tiene un enfoque distinto, por lo que puede ser que algunos sean mas detallados que otros en algunos aspectos.

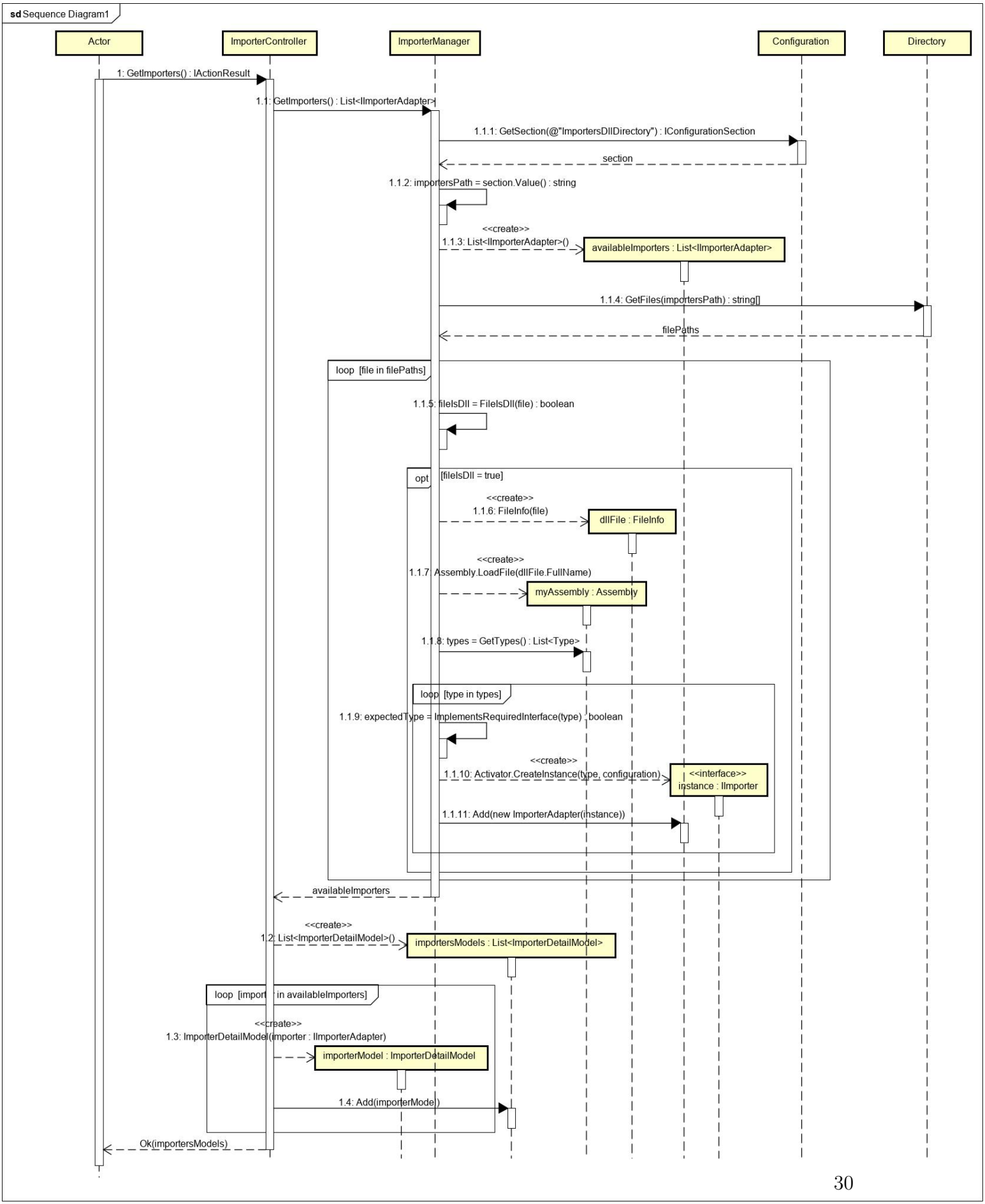
Se decidió realizar nuevos diagramas para brindar mas información de nuestra solución (sobre todo de las funcionalidades nuevas), mientras los diagramas anteriores se pueden seguir encontrando en la carpeta de diagramas UML.



Calcular el precio de parte del ResortPricingCalculator



Generar reporte A



Obtener todos los importadores

8. Justificación del diseño

A continuación explicaremos el diseño de nuestra solución basándonos en principios de diseño y apoyándonos en los diagramas mostrados anteriormente.

8.1. Patrones GRASP

- **Bajo Acoplamiento:** Nos apoyamos en esto para separar WebApi de Domain mediante el uso de Models en un proyecto distinto, también la inyección de dependencias y el componente ServiceRegistration ayudan a respetar este principio quitando el acoplamiento entre WebApi y todos los servicios a registrar/utilizar en el sistema. También se definió el componente MinTur.ImporterInterface de forma tal que este no quede acoplado a BusinessLogic y Domain, para que el tercero pueda implementar su solución solamente con este paquete y Exceptions, lo cual también aporta a la seguridad del sistema.

- **Alta Cohesión:** Esto lo podemos ver reflejado en todas las clases de nuestra solución a través de todas las capas. En WebApi existe un controlador por recurso, al igual que en BusinessLogic existe un Manager por recurso y en DataAccess un Repositorio por recurso.

- **Polimorfismo:** A pesar de no tener herencia como tal como se menciono anteriormente, utilizamos polimorfismos en nuestra solución con las interfaces (ServiceRegistration, DI, ISearchCriteria, IGroupDiscountPolicy, etc.)

- **Fabricacion pura:** Podemos ver ejemplos de esto con el paquete SearchCriteria en donde se crean clases distintas para cada criterio de búsqueda existente en nuestro dominio ayudando a disminuir el acoplamiento y potenciando la reutilización del código.

- **Indirección:** Existen intermediarios en nuestra solución, por ejemplo en la capa de acceso a datos, previniendo el uso directo del contexto, al igual que en la capa lógica con los manejadores de recursos.

- **Experto:** Nos basamos en el patrón experto para asignarle la lógica de validación a las propias entidades de negocio, también siguiendo lo que Martin Fowler comenta en su articulo (Anemic Domain Model)

8.2. Principios SOLID

- **Principio de Responsabilidad Única:** Cada clase debe tener una única responsabilidad, por ejemplo en ServiceRegistrators, cada clase es responsable de registrar solamente el componente al cual esta asociada. También podemos visualizar esto en ResortPricingProcessor, utilización de filtros, y clases para criterios de búsqueda y políticas de descuento específicas, quitándole responsabilidades a otras clases y favoreciendo tanto la alta cohesión como el bajo acoplamiento.

- **Principio de Abierto/Cerrado** Las clases deben estar abiertas a la extensión y cerradas a la modificación. Podemos ver esto por ejemplo mediante el uso de reflection en ServiceRegistrators, ImporterManager y PricingProcessor, así como la inexistencia de RTTI en nuestro sistema ayuda a cumplir este principio.

- **Principio de Liskov:** Relacionado al uso de polimorfismo, las clases hijas deben poder usarse en el lugar de la clase padre sin que el comportamiento esperable de esta se vea afectado. Como se menciono anteriormente, no utilizamos herencia como tal, pero el uso de interfaces en nuestra solución cumple con este principio.

- **Principio de Segregación de Interfaces:** Cada vez que una interfaz define métodos y es implementada por una clase, esta clase debe implementar de manera correcta todos los métodos definidos en la interfaz. Logramos esto mediante la definición explicita de una interfaz por cada repositorio y por cada manager, haciendo que algunas clases (por ej: RegionManager) no tengan que implementar funcionalidades las cuales no se piden en esta entrega, lo cual sucedería si estuviéramos definiendo un set de funcionalidades en una interfaz con generics de la cual heredaran todos.

- **Principio de Inversión de Dependencias:** Se utilizan intermedios de alto nivel, entre alto y bajo nivel, para que el alto nivel no dependa del bajo nivel. El uso de inyección de dependencias fomenta el cumplimiento de este principio.

8.3. Patrones de diseño utilizados

- **Inyección de dependencias:** Se utilizo el patrón de inyección de dependencias para favorecer la extensibilidad de nuestro sistema, logrando que un cambio en un componente especifico solo tenga impacto en el mismo.

- **Factory:** Utilizamos el patrón Factory para testear la base de datos, permitiéndonos crear tanto un contexto SQL como un contexto InMemory.

- **Fachada:** Utilizamos el patrón fachada para proveer un único punto de acceso a los repositorios, evitando tener que pasar varios repositorios específicos por constructor.

- **Adapter:** Utilizamos el patrón Adapter para desacoplar el componente de ImporterInterface de los componentes de Domain y BusinessLogic, para no brindar información extra a los terceros a la hora de crear un nuevo importador, al igual que se explico antes.
- **Strategy:** Utilizamos el patrón Strategy para definir distintos tipos de políticas de descuento, al igual que criterios de búsqueda, permitiendo extender el sistema fácilmente.

8.4. Algoritmos/Mecanismos que impactan la mantenibilidad

Aquí destacamos el uso de reflection en conjunto del componente ServiceRegistration, lo cual fue inspirado en la forma de trabajar del framework Laravel como se explico anteriormente.
También las interfaces en SearchCriteria y DiscountPolicies, las cuales facilitan agregar criterios de búsqueda y políticas de descuento fácilmente, así como extender las ya existentes, por ejemplo agregar un campo nuevo de búsqueda en algún search criteria, es simplemente extender el metodo MatchesCriteria y aceptar este nuevo campo desde la WebApi, y tanto DataAccess como BusinessLogic no se enteran del cambio.
También el uso de CustomBinders permite darle al usuario una mejor experiencia sin las limitaciones de los binder por defecto del framework,

8.5. Mecanismo de acceso a datos

Utilizamos clases repositorios en conjunto con una interfaz favoreciendo el principio de inversión de dependencias, en conjunto con un contexto y entity framework para la obtención y mantenimiento de nuestras entidades en la base de datos. Como se menciono anteriormente también se utilizo una factory para testear la capa de acceso a datos. **No cambio**

8.6. Manejo de excepciones

Utilizamos la excepcion InvalidOperationException y las otras dos definidas en MinTur.Exceptions: (InvalidDataRequestException y ResourceNotFoundException) ya que son suficientemente genericas como para describir todos los errores que se pueden generar en nuestro sistema en esta entrega. **No se tuvieron que agregar nuevas**
También utilizamos un Exception Filter para capturar y tomar decisiones a partir de la generación de estas u otras excepciones, quitando bloques try catch en cada método de la Web Api.

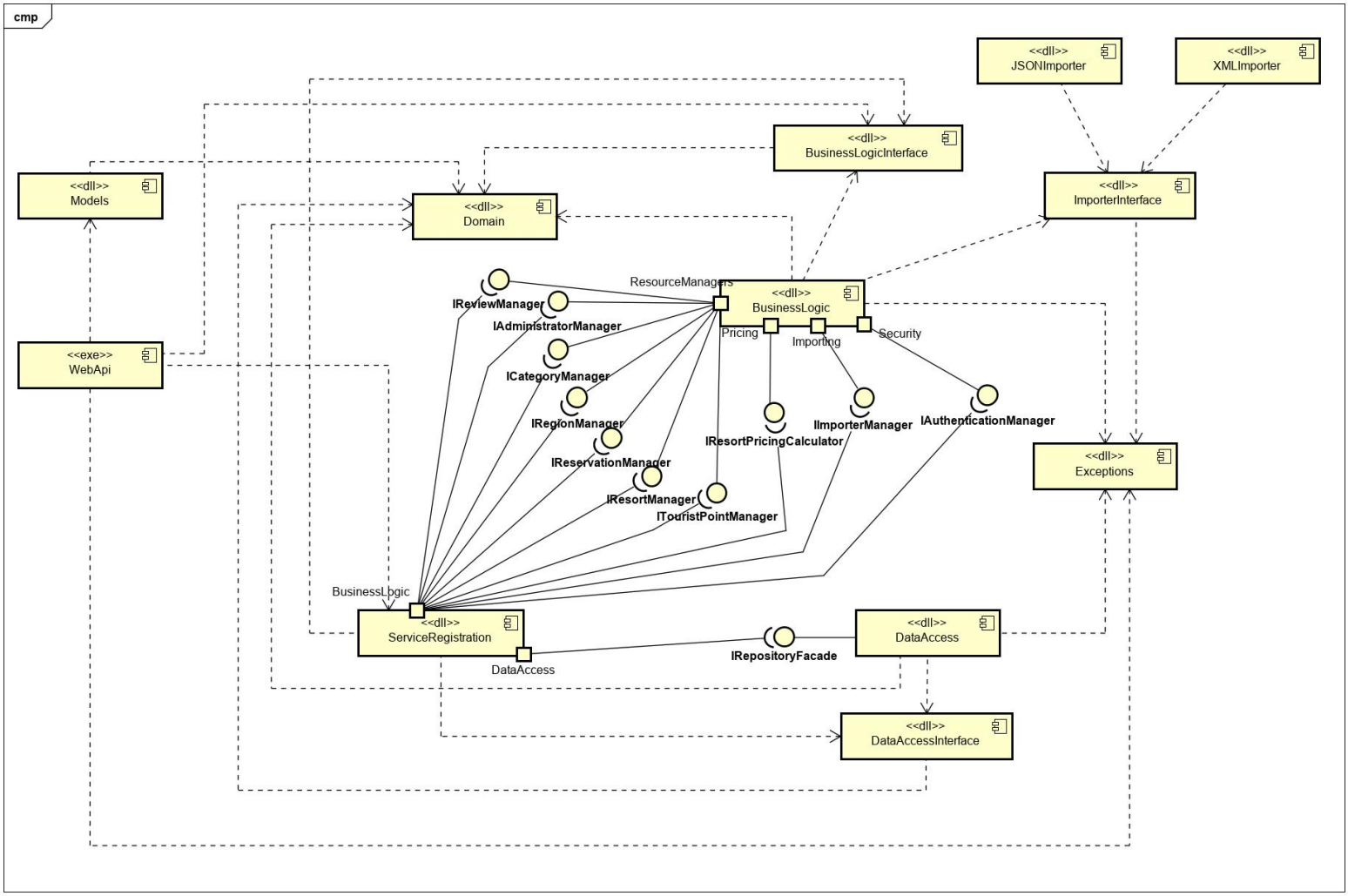
8.7. Resumen de mejoras

En esta seccion se mencionaran las 4 principales mejoras respecto al obligatorio anterior

- Se unificaron los endpoints que no cumplian las buenas practicas en el libro sobre REST (se explica mas en el documento de la API)
- Se realizo el manejo de huéspedes genérico
- Se realizo el calculo del costo de una acomodación para un hospedaje de forma genérica a los tipos de huéspedes existentes.
- Se agregaron algunas relaciones en el paquete de BusinessEntities que modelan mejor la realidad, las cuales fueron necesarias para cumplir con algunos requerimientos (ej: No poder brindar mas de una reseña por reserva)

9. Diagrama de implementación

A continuación se incluye un diagrama de componentes mostrando la dependencias entre los mismos.



La elección de la implementación fue basada principalmente en simplicidad, reuso y arquitecturas de tres capas como se explica a continuación:
Al principio de la entrega teníamos dividida la solución en dos componentes extra pertenecientes a la capa lógica: PrincingProcessor y SessionManagement. Luego de conversaciones con el profesor decidimos que por simplicidad y el dominio acotado de la entrega es mas conveniente colocar esa lógica en BusinessLogic y dividirlo en distintos paquetes.
También podemos ver que cada componente provee una funcionalidad distinta y bien definida lo cual ayuda el reuso de estos mismos. La única excepción es BusinessLogic, pero como se explico anteriormente la simplicidad termino siendo mas importante a nuestro parecer en la solución. Finalmente cabe destacar que se aisló ImporterInterface del resto de la solución a excepción de Exceptions como se explico también anteriormente, permitiendo que el tercero pueda implementar sus propios importadores sin conocer nada mas ni menos de lo que necesita.

10. Métricas

Utilizamos NDepend para calcular las métricas (H, I, A y D') sobre los assemblies de nuestra solución.

- MinTur.WebApi:
  - I = 1
  - A = 0
  - D' = 0 (esta en la secuencia principal) lo cual es la relación ideal entre inestabilidad y abstracción, ya que WebApi no deberia depender de nadie al ser el punto de entrada de nuestra aplicacion, por lo que tampoco incluye clases abstractas.

  En cuanto a H tanto en WebApi en general como en ambos paquetes internos (Filters y Controllers) es 0 ya que no existen relaciones entre los tipos presentes. Esto no es algo malo, ya que nos basamos en el principio CRP ante el CCP, ya que tener los controllers en distintos paquetes no tiene sentido en cuanto a reuso en este caso.

- MinTur.Models:
  - I = 0.86
  - A = 0
  - D' = 0.14 lo cual se encuentra cerca de la secuencia principal, a pesar de no ser mal resultado quizás se podría mejorar separando los paquetes internos en otros componentes, haciendo que algunos de ellos estén mas cerca de la secuencia principal.

  En cuanto a H:

  - Models.Out: (R + 1) / N = (12 + 1) / 17 = 0.76 - No llega a 1.5 debido a que varios modelos tienen asociaciones con otros pero la mayoría no necesita. Aquí predomino el CRP ante el CCP, ya que agrupamos en este paquete todos los modelos salientes lo cual facilita el reuso, aunque otra opción podría ser agrupar por recurso y ver si al mejorar esta métrica no perjudica a otras.
  - Models.In: (R + 1) / N = (1 + 1) / 13 = 0.15 - Al igual que en el paquete Out, predomino CRP ante el CCP por el mismo motivo, y la posible mejora seria la misma.

- MinTur.BusinessLogicInterface:
  - I = 0.35
  - A = 1 (Todas las clases son abstractas)
  - D' = 0.35 lo cual se encuentra cerca de la secuencia principal, a pesar de no ser mal resultado quizás se podría mejorar separando los paquetes internos en otros componentes, al igual que antes, lo cual es algo que se menciono anteriormente en el diagrama de implementación y se converso con el profesor antes del primer obligatorio, en donde idealmente el componente BusinessLogicInterface no debería existir y estaría dividido en varios distintos, pero por simplicidad se decidió hacerlo así.
    Aquí se cumple mas CRP que CCP, aunque para incrementar CRP se debería hacer lo mencionado.

  En cuanto a H todos los paquetes internos tienen un H de 0 excepto Importing en donde IImporterManager necesita de un IImporterAdapter para funcionar, pero fuera de esto el resultado se debe a que se agrupo al igual que en los controladores por reuso, y cada manager es independiente de otro, pero cumplen todos la misma función.

- MinTur.BusinessLogic:
  - I = 13/18 = 0.99
  - A = 0
  - D' = abs(A + I - 1) = 0.01 lo cual se encuentra casi en la secuencia principal debido a que depende solamente de Domain y no posee clases abstractas. Aquí se nos ocurre la misma mejora que la anterior separa BusinessLogic en varios paquetes de negocio, aunque la métrica no mejoraría por esto (ya que casi todos precisan a Domain) pero quizás si en los otros paquetes.

  En cuanto a H pasa lo mismo que antes, todos los paquetes internos menos Importing poseen un H de 0, y la razón aquí es la misma que en BusinessLogicInterface.

- MinTur.DataAccessInterface:
  - I = 0.3
  - A = 1
  - D' = 0.3 lo cual se encuentra cerca de la secuencia principal, aquí se podría aplicar lo mismo que se menciono en BusinessLogicInterface separando los paquetes internos por recursos.

  En cuanto a H es 0 en ambos paquetes internos, esto se podria mejorar quitando el paquete de Facades e incluyéndolo en Repositories, el cual se agrego para mantener un orden, pero no fue agregado pensado en ninguno de los principios CRP y CCP.

- MinTur.DataAccess:
  - I = 0.99
  - A = 0
  - D' = 0.01 aquí pasa lo mismo que en BusinessLogic, por lo que los argumentos son los mismos.

  En cuanto a H también es 0 en todos los casos menos en Contexts que es 1, esto se debe a que se pensó mas en el reuso que en CCP ya que a pesar de no estar relacionadas cumplen todas el mismo propósito.

- MinTur.Domain:
  - I = 0.33
  - A = 0.07
  - D' = 0.42 Se encuentra en la zona de dolor ya que cambiar algo aquí afecta a casi toda la solución, no se nos ocurre una forma de mejorar esto ya que son las definiciones de las entidades de negocio.

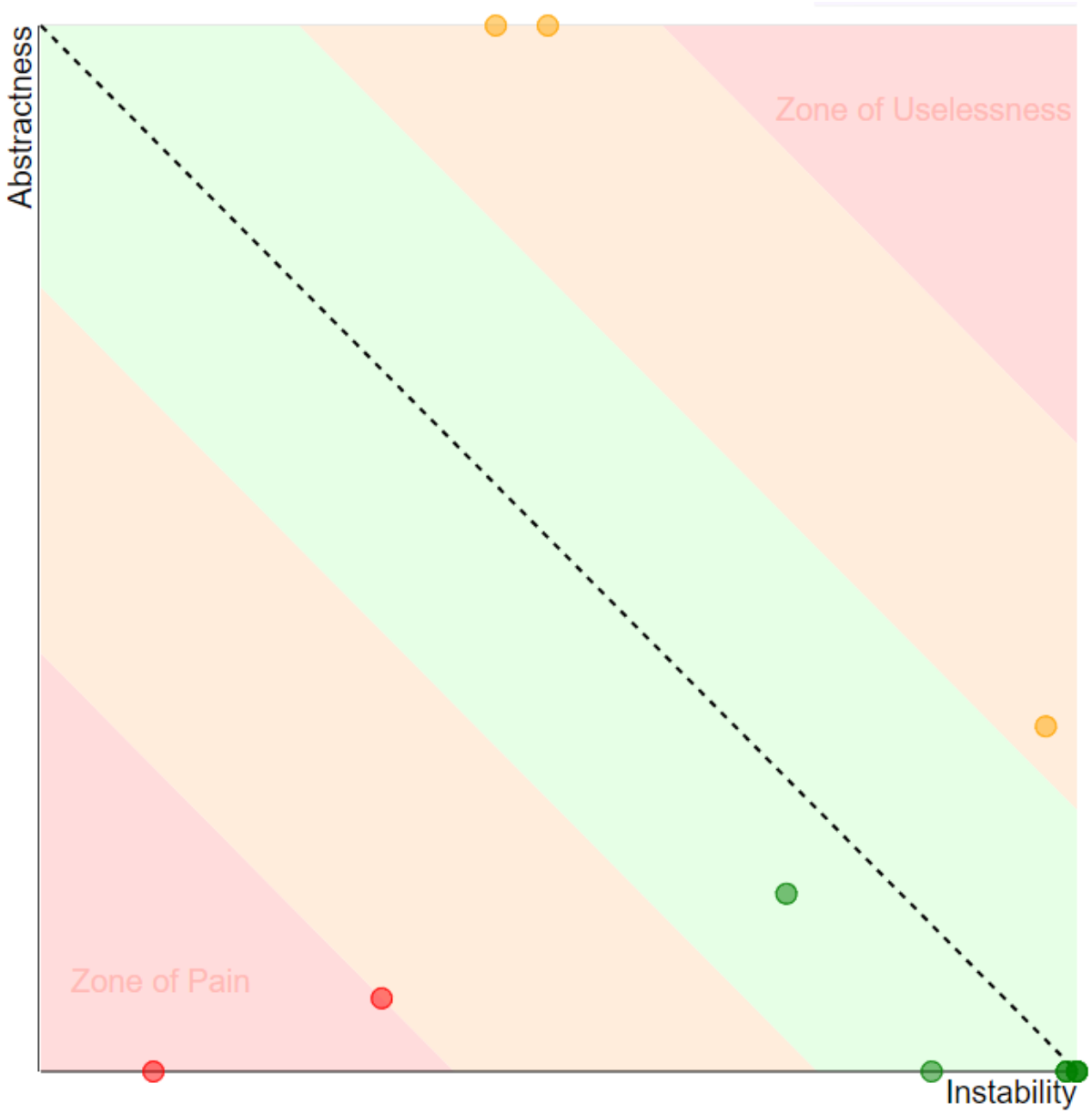
  En cuanto a H:

  - Domain.BusinessEntities: (R + 1) / N = (17 + 1) / 4 = 4.5 las clases están muy relacionadas entre si, por lo que cumplimos con CCP. A pesar que no se pueda reutilizar un conjunto de entidades por separado, esperamos que si se usa este paquete se use la mayoría de estas.
  - Domain.SearchCriteria: (R + 1) / N = (2 + 1) / 3 = 1 aquí predomina el CRP ya que se puede reutilizar los criterios de búsqueda en conjunto, aunque podriamos separarlo por recurso para brindar mas flexibilidad en CRP.
  - Domain.Reports: (R + 1) / N = (2 + 1) / 3 = 1 aquí pasa lo mismo que en SearchCriteria.
  - Domain.Importing: (R + 1) / N = (1 + 1) / 4 = 0.5 aquí definimos todo lo relacionado a importers, pensando en el reuso por lo que esta métrica tiene sentido.
  - Domain.DiscountPolicies: (R + 1) / N = (3 + 1) / 4 = 0.5 aquí al igual que en los puntos anteriores se pensó en el reuso, ya que cuando utilizamos 1 política de descuento pensamos usar todas, por lo que tiene sentido también esta métrica.

- MinTur.Exceptions:
  - I = 0.11
  - A = 0
  - D' = 0.63 Se encuentra en la zona de dolor ya que cambiar algo aquí afecta a casi toda la solución, aquí se pensó en el CRP ya que uno espera utilizar todas las excepciones y no solo 1, por lo que esta métrica tiene sentido.

En cuanto a H en este caso es 0, ya que como se menciono antes se penso en CRP antes que CCP.

A continuación se deja la captura de IvsA de NDepend como evidencia de los resultados anteriores:



11. Anexo

11.1. API

Se actualizo la documentacion de la API, así como una sección en la cual explica los cambios realizados desde la entrega anterior. La misma se encuentra en el siguiente link:

Link a la documentación en Postman

11.2. Cobertura de pruebas

A continuación se incluye el informe de cobertura de pruebas unitarias, cabe destacar que se dejo de testear los modelos ya que en clase unos días antes de entregar el primer obligatorio se menciono que no era necesario, por lo que por motivos debido de tiempo debido al trabajo, decidimos dejar de hacerlo.
Aun así obtuvimos una cobertura mayor al 90 % como se puede ver en la imagen:

Matias_DESKTOP-QEVFBJI 2020-11-25 18_36_

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Matias_DESKTOP-QEVFBJI 2020-1...	1053	8.67%	11088	91.33%
mintur.businesslogic.dll	23	5.53%	393	94.47%
mintur.businesslogic.test.dll	113	6.53%	1617	93.47%
mintur.dataaccess.dll	411	28.50%	1031	71.50%
mintur.dataaccess.test.dll	88	7.95%	1019	92.05%
mintur.domain.dll	122	11.29%	959	88.71%
mintur.domain.test.dll	39	2.67%	1419	97.33%
mintur.exceptions.dll	4	50.00%	4	50.00%
mintur.importerinterface.dll	9	5.81%	146	94.19%
mintur.jsonimporter.dll	0	0.00%	56	100.00%
mintur.jsonimporter.test.dll	3	1.90%	155	98.10%
mintur.models.dll	95	8.08%	1081	91.92%
mintur.models.test.dll	3	0.27%	1120	99.73%
mintur.serviceregistration.dll	0	0.00%	18	100.00%
mintur.serviceregistration.tes...	74	25.61%	215	74.39%
mintur.webapi.dll	2	0.66%	302	99.34%
mintur.webapi.test.dll	60	4.34%	1323	95.66%
mintur.xmlimporter.dll	4	5.26%	72	94.74%
mintur.xmlimporter.test.dll	3	1.86%	158	98.14%

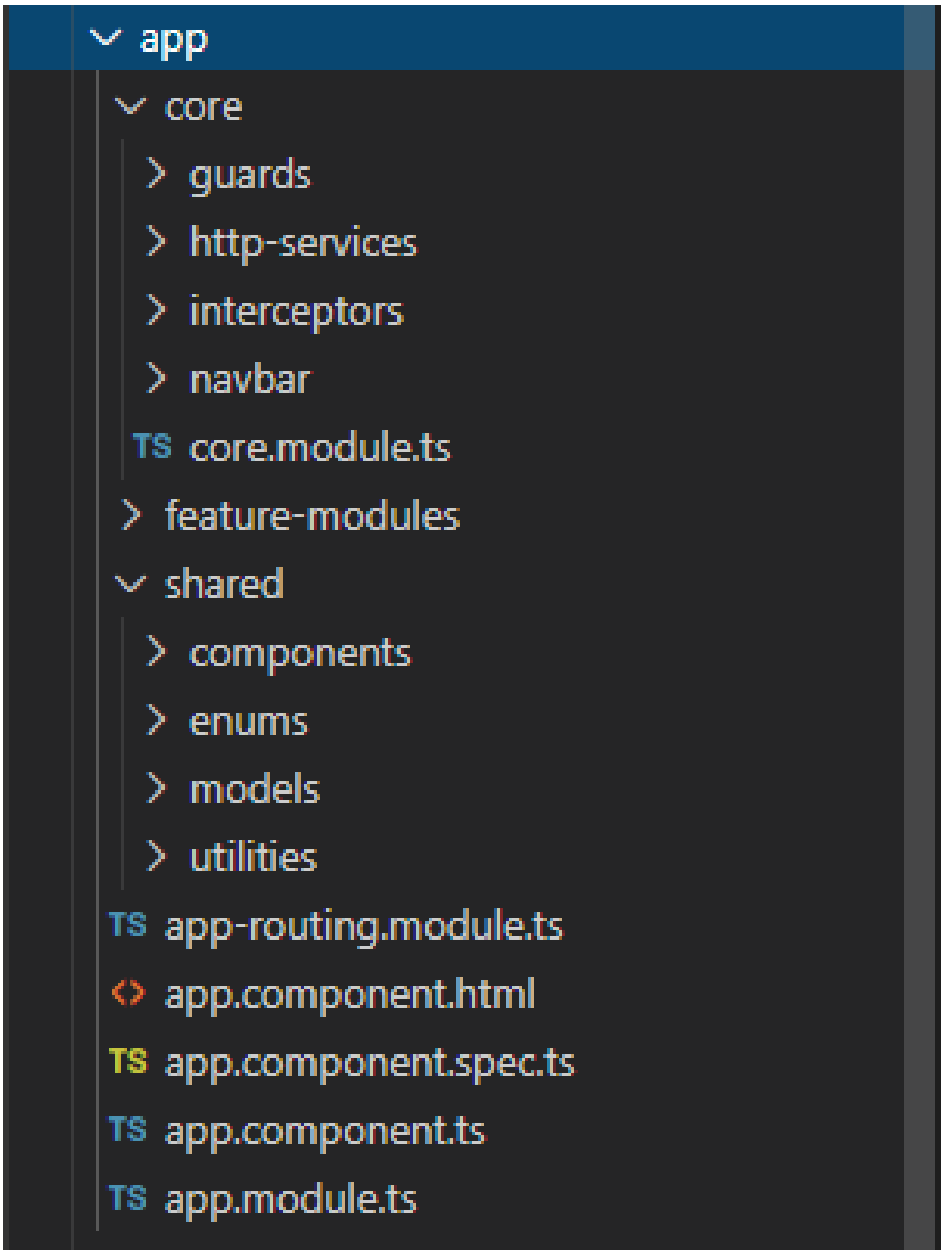
Se excluyo del informe de cobertura solamente las fachadas y las migraciones. Cabe destacar que figuran algunas partes de las migraciones como con cobertura 0 % las

cuales no pudimos evitar, lo que hizo que la cobertura global de DataAccess sea menor a la real.

11.3. Angular

En angular usamos una estructura de carpetas lo mas 'FLAT' posible, pensando en mantenibilidad y escalabilidad ante todo, nos inspiramos en el siguiente artículo el cual explica el propósito de cada carpeta:

Link al artículo



En cuanto al código en Angular, no nos dio el tiempo para realizar algunos refactors que teníamos pensados, como el uso de los forms de angular entre otros.
En cuanto a la interfaz se podrían realizar pequeñas mejoras para mejorar la experiencia de usuario las cuales no fueron realizadas por falta de tiempo.

Finalmente, utilizamos TailwindCSS como framework de CSS ya que nos permite maquetar rápidamente y fácilmente las paginas. Seguimos criterios en cuanto al orden en el cual se aplican las clases de CSS para aumentar la mantenibilidad, aunque para escalar se debería remover algunas clases de CSS a componentes como se explica de la pagina de Tailwind, pero esto no se hizo por falta de tiempo.

11.4. Instalación

A continuación explicaremos el proceso de instalación a partir de las carpetas MinTurBackend y MinTurFrontEnd dentro de Aplicación y los archivos en la carpeta Base de Datos.

- En SQL Server Management Studio recuperamos la base de datos con el .bak con datos incluidos brindado.
  Quizas sea necesario copiar el .bak en la carpeta que SQL Server Management Studio tiene permisos para leer/escribir.
- Abrir IIS (tecla inicio + r) y escribir 'inetmgr'.
  En caso de no tenerlo se debe activar mediante el panel de control, en la sección de programas.
- Verificar que tenemos el modulo de .Net Core, de lo contrario lo bajamos de la pagina oficial.
- Copiamos la carpeta MinTurBackEnd en 'C:\inetpub\wwwroot\'
- Creamos un nuevo sitio en IIS con cualquier nombre (ej: MinTurBackend) y seleccionamos como ruta de acceso física la carpeta que copiamos en el directorio anteriormente mencionado. El puerto tambien es opcional (ej: 81)
- Luego en nuestro grupo de aplicaciones seleccionamos la recien creada y en las opciones le desactivamos la version de .NET CLR
- En SQL Server Management Studio creamos un nuevo login con el nombre IIS APPPOOL\[nombre que le dimos a nuestro sitio] con el rol de sysadmin
- Ya podemos testear cualquier endpoint de la API
- Ahora pasamos a copiar la carpeta MinTurFrontEnd a la ruta 'C:\inetpub\wwwroot\'
- Creamos un nuevo sitio en IIS con un nombre a elección (ej: MinTurFrontEnd) y seleccionamos como ruta de acceso física la carpeta que copiamos en el directorio anteriormente mencionado. El puerto también es opcional pero no debe coincidir con el del backend (ej: 82).
- En la carpeta que copiamos dentro del directorio de IIS, abrimos el archivo que comienza con la palabra 'main' con algun editor de texto como VS Code u otro.

- Usando ctrl + f buscamos 'localhost' y cambiamos el puerto al que le dimos anteriormente al backend, al igual que modificamos https por http.
- En caso de que de error al intentar al acceder mediante el navegador debemos bajar la dll 'rewrite.dll' del sitio oficial de IIS.
- Ahora ya podemos utilizar nuestra aplicación sin problema alguno.