

Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 2

Obligatorio 1

Felipe Najson (232863)
Santiago Topolansky (228360)

Entregado como requisito de la materia Diseño de
Aplicaciones 2

15 de octubre de 2020

Declaraciones de autoría

Nosotros, Felipe Najson y Santiago Topolansky, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos Diseño de Aplicaciones 1;
- 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.

Índice general

1. Descripción del Diseño	4
1.1. Introducción	4
1.2. Descripción general del trabajo y del sistema	5
1.3. Organización de la Solución	7
1.3.1. Paquete: Domain	10
1.3.2. Paquetes: Logic y LogicInterface	12
1.3.3. Paquetes: Persistence y PersistenceInterface	13
1.3.4. Paquete: WebApplication	17
1.4. Diagrama de Componentes	18
1.5. Patrones y Decisiones de Diseño	19
1.5.1. Inyección de Dependencias	19
1.5.2. Patrón Repository	19
1.5.3. Patrón Strategy	21
1.5.4. GRASP	22
1.5.5. SOLID	24
1.6. Funcionalidades Claves	26
1.7. Manejo de Excepciones	28
2. Evidencia del Diseño y especificación de la API	29
2.1. Cumplimiento de REST	29
2.1.1. Recursos	30
2.1.2. Endpoints	31
2.1.3. Mecanismos de autenticación	35
2.1.4. Códigos de Error HTTP	37
3. Evidencia de la aplicación de TDD y Clean Code	38
3.1. Clean Code	38
3.2. TDD	40
3.2.1. Adaptación de TDD y evidencia de uso	40
3.2.2. Enfoque	40
3.3. Repositorio GitHub	41
3.4. Base de datos secundaria	42
3.5. Cobertura de Pruebas Unitarias	43

4. Evidencia de la ejecución de las pruebas de la API con Postman	45
4.1. Casos de Prueba	45
4.2. Evidencia de ejecución de pruebas	50
5. Conclusiones	51
5.1. Deuda Técnica	51
5.2. Reflexión Final	53
Bibliografía	54

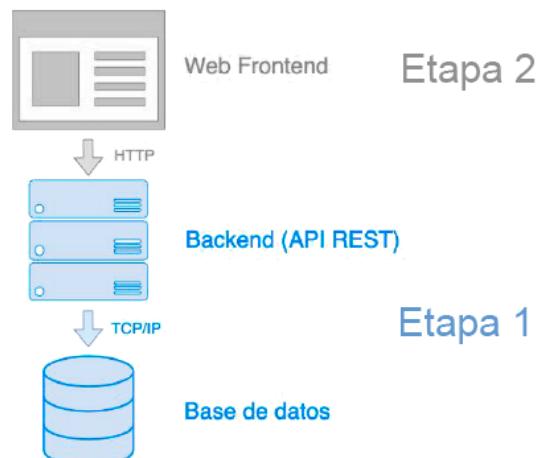
1. Descripción del Diseño

1.1. Introducción

Uruguay Natural es un portal de oferta turística nacional. En el mismo los usuarios pueden encontrar puntos turísticos de interés, y una amplia variedad de ofertas de hospedaje asociadas a estas. El sistema permite a los clientes encontrar hospedajes según las categorías deseadas, y consultar de estos: precio, datos de contacto, imágenes, entre otros. Además, el sistema permite cotizar las estadías así como la posterior reserva, diferenciando por edad de los huéspedes, y la cantidad de días de la estadía.

En esta primera etapa, el proyecto se orientó a diseñar e implementar el back-end del sistema, considerando los requerimientos y la arquitectura solicitada. También se resolvió en esta etapa la persistencia de los datos de la aplicación utilizando una base de datos.

Para la ejecución de este proyecto se tuvo en cuenta permanentemente la necesidad de un diseño que permitiera la posterior integración con un front end desarrollado en un futuro. Por esta razón, se procuró seguir patrones de diseño que facilitaran este objetivo. El producto final es una Web API con una serie de endpoints disponibles para ser consumidos por cualquier cliente.



1.2. Descripción general del trabajo y del sistema

El sistema desarrollado cumple con la siguiente especificación funcional:

1. **Regiones:** El sistema dispone de una serie de regiones de Uruguay disponibles por defecto. Estas son: Región metropolitana, Región Centro Sur, Región Este, Región Litoral Norte y Región “Corredor Pájaros Pintados”.

En esta versión no se soporta el mantenimiento de las regiones, por lo que cualquier otra se deberá agregar directamente a la base de datos. Sí se permite la consulta de todas las regiones disponibles.

2. **Categorías:** El sistema dispone de una serie de categorías turísticas disponibles por defecto. Estas son: Ciudades, Pueblos, Áreas protegidas, Playas, Etc.

En esta versión no se soporta el mantenimiento de las categorías, por lo que cualquier otra se deberá agregar directamente a la base de datos. Sí se permite la consulta de todas las categorías disponibles.

3. **Puntos Turísticos:** El sistema permite la gestión de puntos turísticos. Estos tienen una región y pueden tener categorías asociadas, que faciliten su búsqueda. El sistema permite las siguientes operaciones sobre los puntos turísticos:

- a) Consultar todos los puntos turísticos disponibles
- b) Consultar puntos turísticos por región y categorías (opcionalmente)
- c) Agregar un nuevo punto turístico *

4. **Hospedajes:** El sistema permite la gestión de hospedajes. Estos están asociados a un punto turístico en particular, y tienen información disponible, como una descripción, un contacto, imágenes, cantidad de estrellas y precio. El sistema permite las siguientes operaciones sobre los hospedajes:

- a) Consultar hospedajes por punto turístico
- b) Agregar un nuevo hospedaje *
- c) Borrar un hospedaje *
- d) Modificar un hospedaje existente *

Nota: En esta versión solo se soporta la modificación de la capacidad, que es binaria; o hay capacidad disponible, o no la hay.

5. **Reservas:** La funcionalidad central del sistema radica en la creación de reservas por parte de los usuarios. Aunque se dejó fuera del flujo de la aplicación el procesamiento del pago, la reserva tiene un precio, que es calculado a partir de la cantidad y tipo de huéspedes, la cantidad de noches, y el hospedaje

seleccionado.

Según las reglas de negocio definidas para el sistema, los adultos pagan la taza completa por cada noche, mientras que los niños (de 2 a 12 años) pagan el 50 %, y los bebés el 25 %.

Para poder realizar la reserva, el usuario debe proveer su nombre y apellido, y un correo electrónico. El sistema consulta si existe registro de dicho cliente previamente y en caso negativo lo registra, antes de asignarle la reserva.

En cuanto a la identificación de la reserva, al momento de su ingreso, se autogenera un código identificador único que se le provee al cliente. Con este código, es posible consultar el estado de la reserva en cualquier momento. El sistema dispone de una serie de estados para las reservas disponibles por defecto. Estas son: Creada, Pendiente Pago, Aceptada, Rechazada, y Expirada. Además, las reservas tienen una descripción del estado en todo momento.

El sistema permite las siguientes operaciones sobre las reservas:

- a) Consultar todas las reservas *
 - b) Consultar una reserva a partir del código de reserva
 - c) Consultar todos los estados de reserva *
 - d) Crear una nueva reserva
 - e) Modificar el estado de una reserva y su descripción *
6. **Administradores:** El sistema persiste la información básica de sus clientes en el caso de que realicen una reserva, pero no tiene un control de acceso general al sistema para los usuarios clientes. Sin embargo, en el caso de los administradores el sistema sí los gestiona, de tal manera que pueden acceder mediante un usuario y contraseña, e iniciar una "sesión". Al hacerlo, reciben un token que identifica la sesión. Este token les otorga privilegios de administrador cada vez que realizan una acción restringida a los administradores.

Las acciones que pueden realizar los administradores una vez logueados en el sistema, son todas aquellas marcadas con asterisco (*) en la descripción de las funcionalidades anterior, a las que se suma también la gestión de los administradores:

- a) Crear un nuevo administrador en el sistema
- b) Modificación del nombre, apellido y contraseña de un administrador a partir de su correo electrónico

- c) Eliminar un administrador a partir de su número identificador.
- d) Consultar todos los estados de reserva
- e) Modificar el estado de una reserva y su descripción *

Al momento de la entrega todos los bugs detectados han sido corregidos. No tenemos conocimiento de ninguna funcionalidad que no funcione correctamente según lo especificado.

Comentario: Se planea mejorar el manejo de las imágenes de los hospedajes en la siguiente etapa, al implementar el front end. Actualmente se cuenta con un atributo único de tipo string, donde se definen la/s ruta/s de la/s imagen/es de los hospedajes (en el caso de ser varias, se espera se envíen concatenadas).

1.3. Organización de la Solución

La solución esta dividida en 7 paquetes principales:

- **WebApplication:** Es la parte que sirve de interacción con los usuarios. Es responsable de recolectar los datos de entrada del usuario, que pueden ser de variadas formas, y validarlos y transformarlos, ajustándolos a las especificaciones que demanda el backend para poder procesarlos. También se encarga de reportar errores de parte del usuario previo al ingreso de la información del sistema.
- **Domain:** Contiene las clases centrales que componen el sistema, y responden a las reglas de negocio que determinan cómo la información es creada, almacenada y modificada.
- **Logic:** En este paquete se realiza la gestión de la información. Cada clase de Logic es un controlador que responde a la entrada del usuario y realiza interacciones en los objetos del modelo de datos. El controlador recibe la entrada, la valida y luego pasa la entrada al modelo.
- **Persistence:** El paquete de persistencia contiene por un lado la configuración de los parametros y relaciones entre tablas de la base de datos y por otro lado clases que permiten agregar, quitar o consultar en la base de datos, estas clases de persistencia guarda los elementos del sistema de forma permanentemente en la base de datos.
- **Models:** Es un paquete auxiliar el cual contiene los modelos utilizados para la entrada y salida de datos a traves de la Web API.
- **LogicInterface / PersistenceInterface:** Estos paquetes contienen la colección de operaciones que se utilizan para especificar el servicio provisto por las clases de los paquetes Logic y Persistence.

Aquí se representan las relaciones entre los diferentes paquetes de la solución:

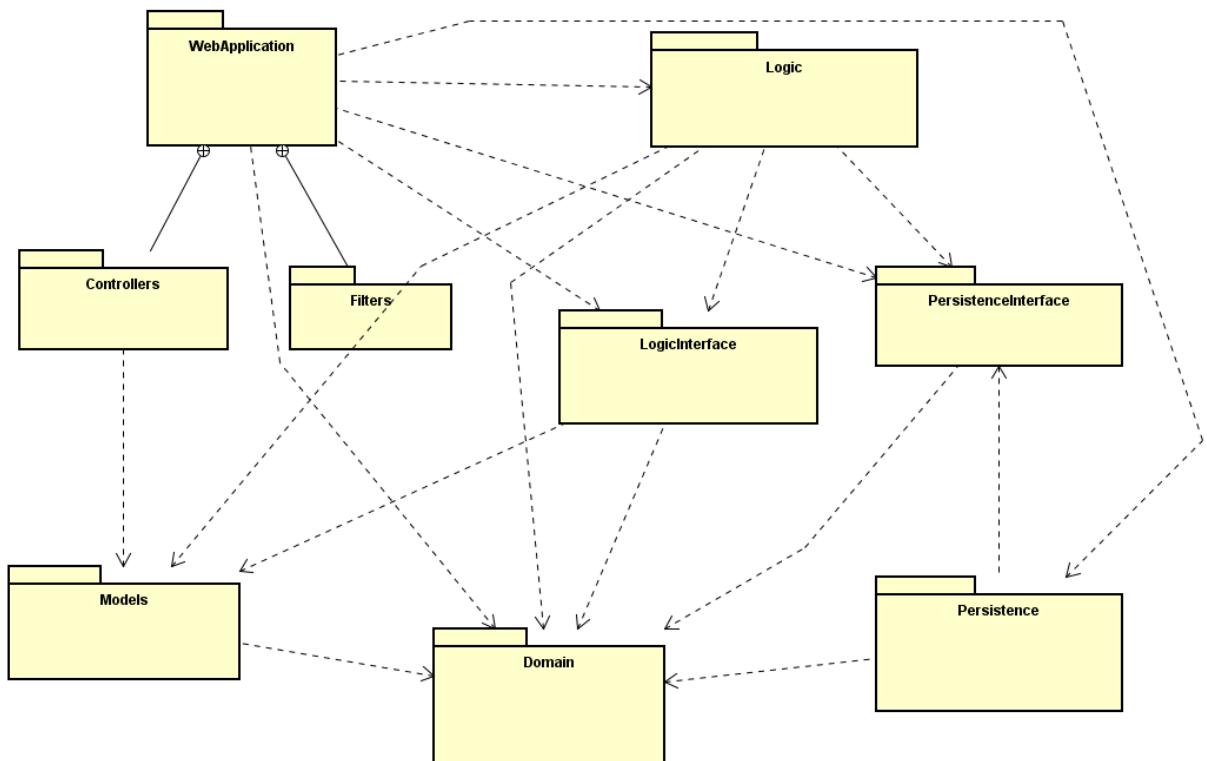


Figura 1.1: Diagrama de Paquetes - Mostrando los paquetes organizados por capas (layers) y sus dependencias.

Nota: Aunque el diagrama muestra la organización lógica del proyecto, en el proyecto físico existe una dependencia de la WebAPI hacia el paquete Logic. Esta dependencia se da debido a que la clase StartUp que contiene la configuración del runtime, se encuentra en el componente WebAPI. Esta clase es responsable de la inyección de dependencias, y por lo tanto debe conocer las implementaciones disponibles (del paquete Logic).

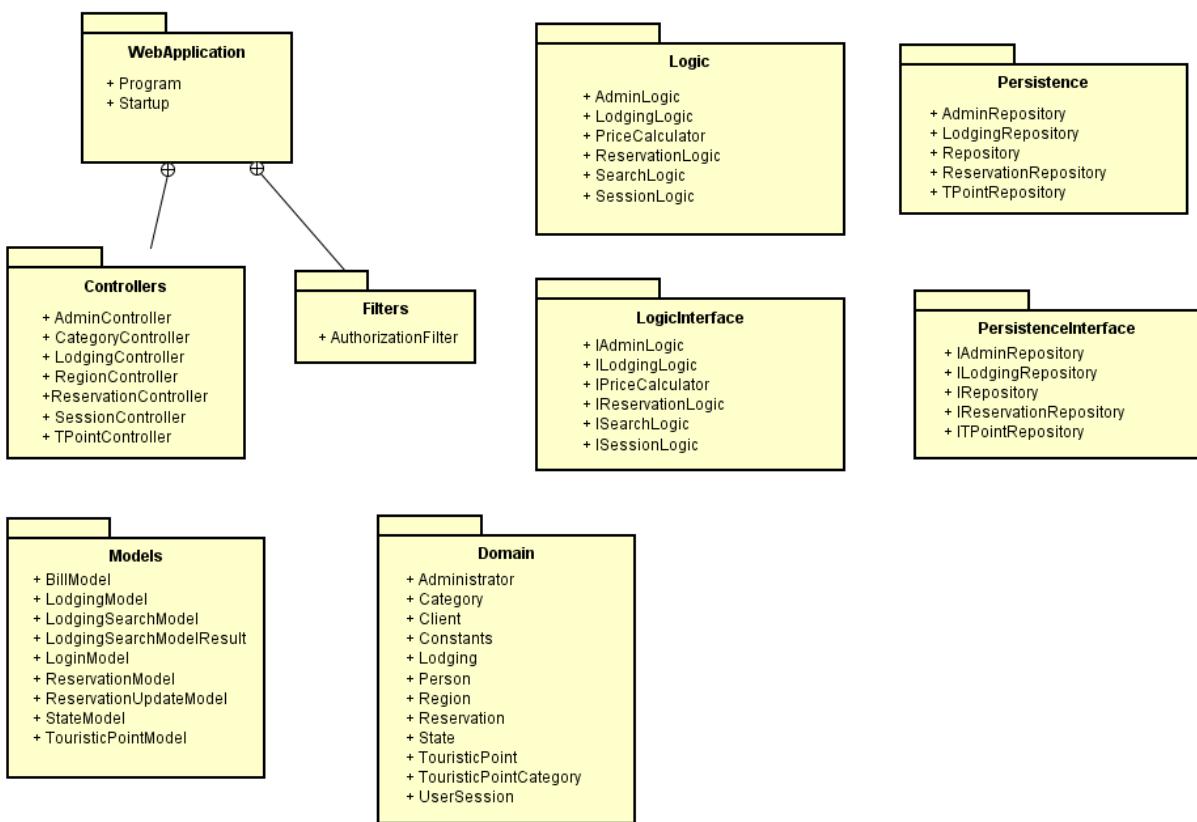
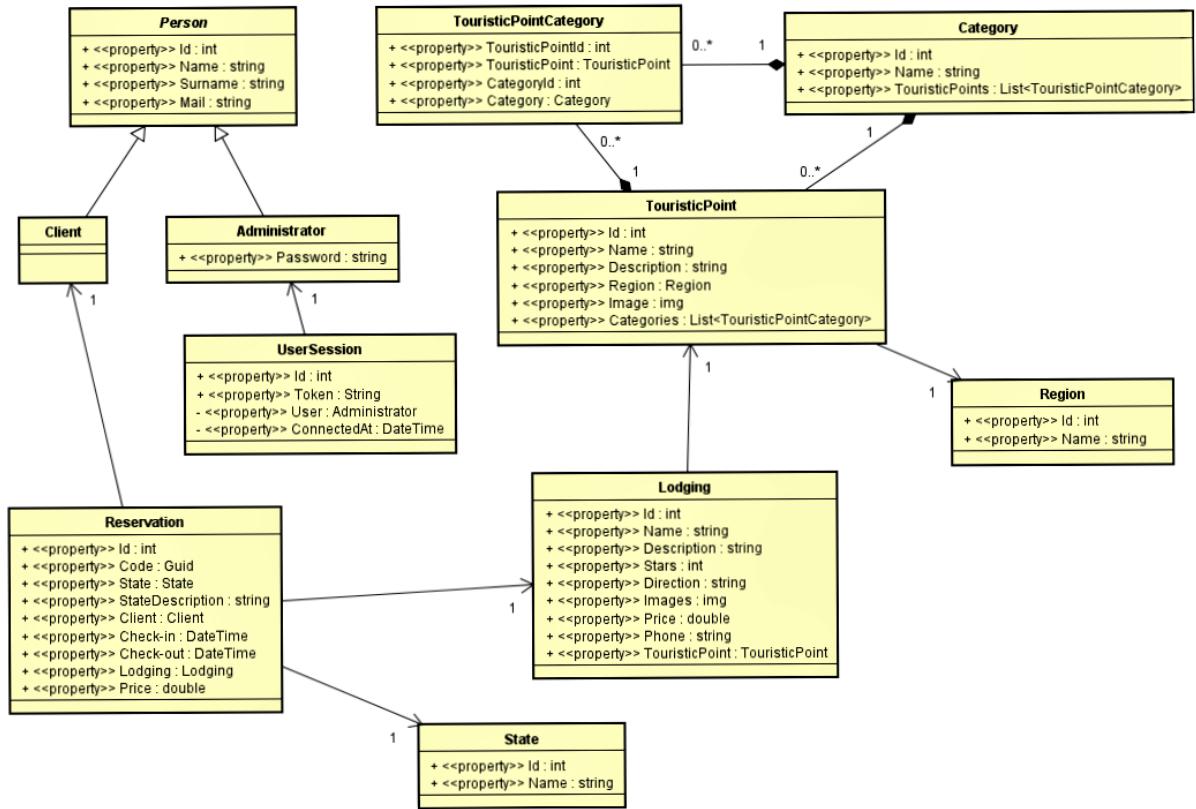
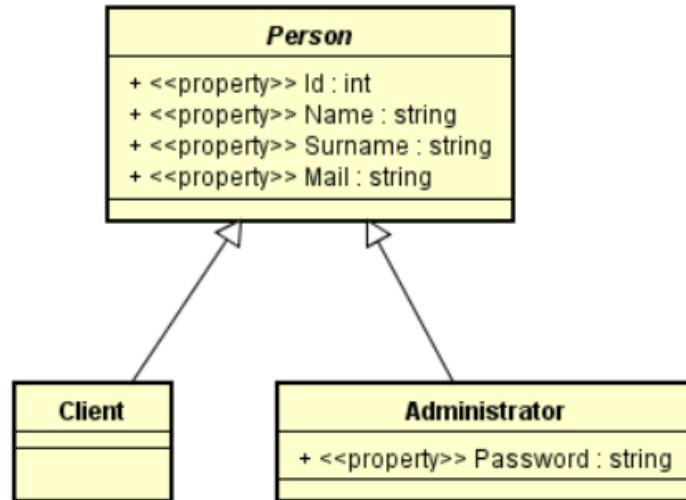


Figura 1.2: Diagrama de Paquetes - Utilizando conector de Nesting sin dependencias

1.3.1. Paquete: Domain

A continuación se presenta el diagrama de clases UML para el dominio, donde se denotan claramente los distintos tipos de relaciones entre las clases, con roles, cardinalidades y dependencias.





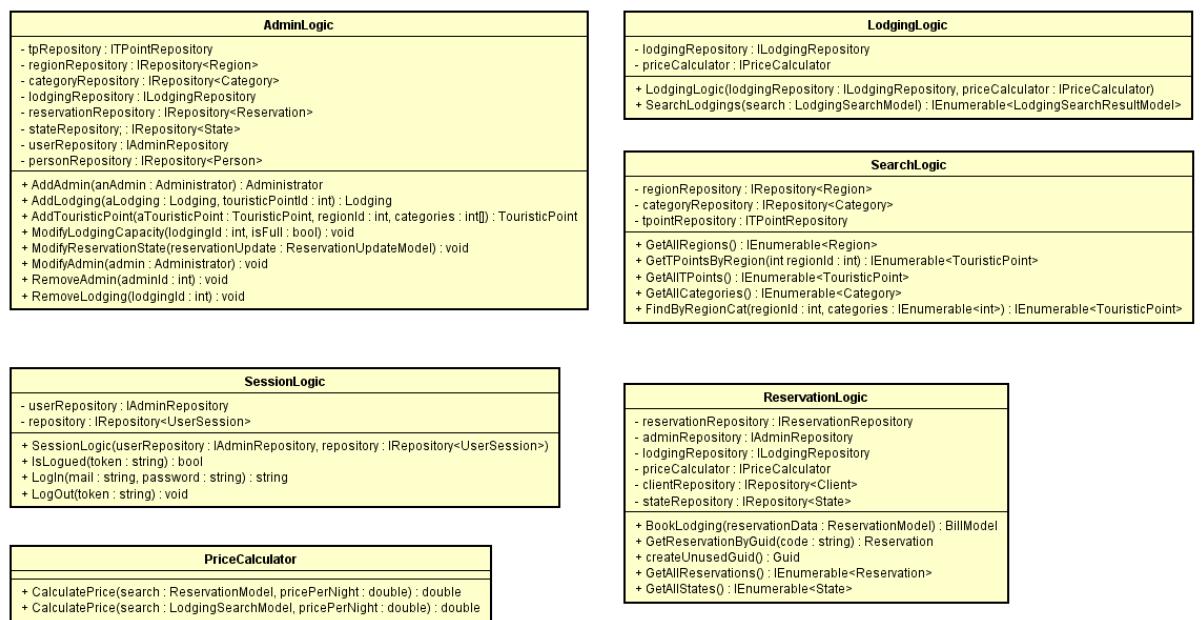
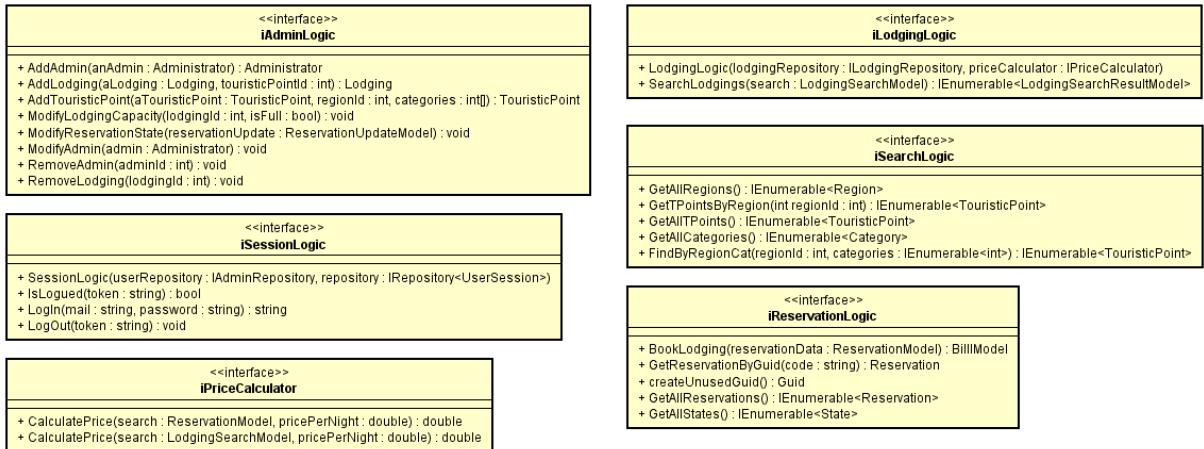
■ **Herencia:**

1. **Person - Client:** Client hereda de la clase abstracta Person implementando todos sus métodos abstractos y no agrega atributos.
2. **Person - Administrator:** Administrator hereda de la clase abstracta Person implementando todos sus métodos abstractos y agregando un atributo nuevo: una contraseña (password).

Se implementó esta herencia con el propósito que si, en un futuro, se agregan nuevas funcionalidades a “personas” las dos clases heredadas puedan ya se reutilizar código o sobreescibir la funcionalidad.

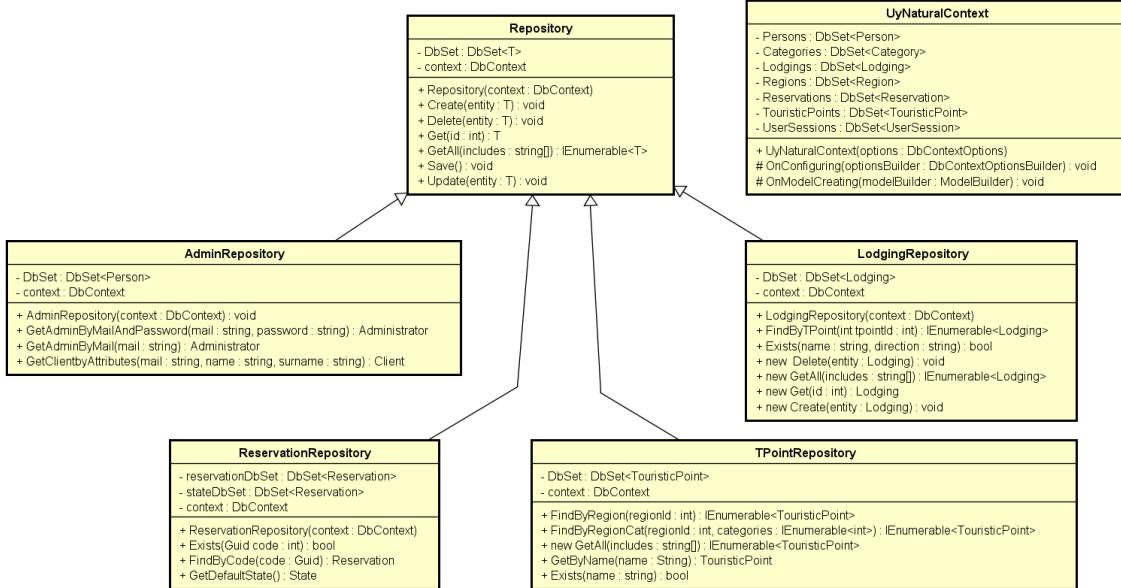
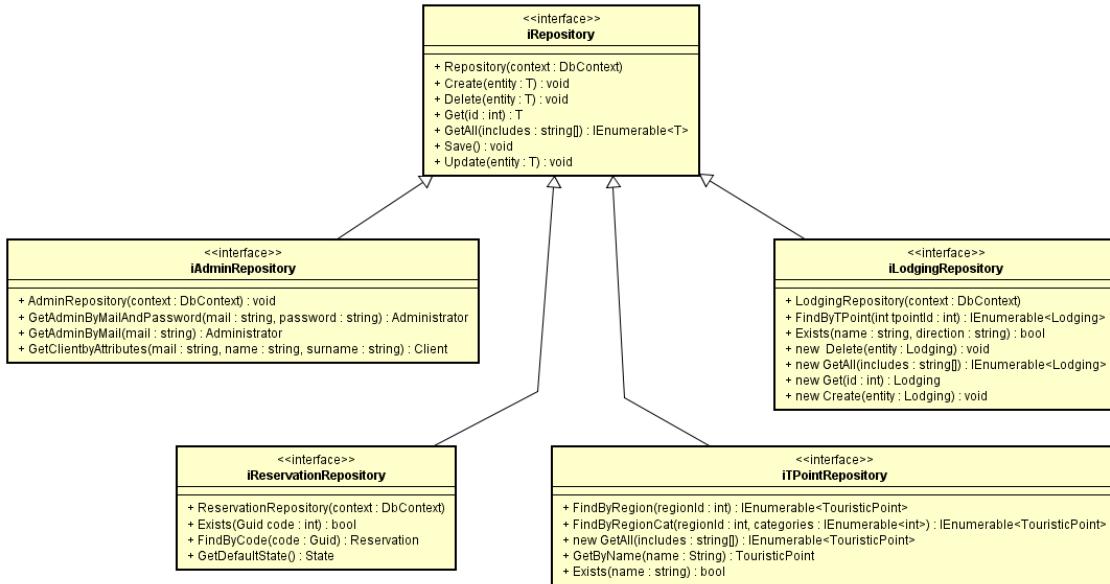
1.3.2. Paquetes: Logic y LogicInterface

Diagrama de Clases LogicInterface / Logic



1.3.3. Paquetes: Persistence y PersistenceInterface

Diagrama de Clases PersistenceInterface / Persistence



Almacenamiento de Datos

Entity Framework Core En esta versión la aplicación utilizamos Entity Framework Core (soportado por Microsoft en el desarrollo de aplicaciones .NET Core) para la persistencia de los datos en una base de datos relacional.

Entity framework nos habilitó a trabajar con datos usando las clases específicas del dominio, sin necesidad de enfocarnos en las estructuras de tablas de la base de datos subyacentes.

Utilizamos el enfoque **Code First**, lo que implica la priorización de la construcción del dominio de la aplicación frente al diseño de la estructura específica de tablas. Esto es posible ya que mediante un mapeo casi automático del dominio, EF es capaz de construir un modelo concreto de tablas sobre el que la aplicación puede trabajar.

Sin embargo, también fue necesario realizar ciertos ajustes específicos para indicar la forma exacta en que deseábamos que trabajara el modelo generado, mapeando ciertas características del dominio como relaciones y cardinalidades, restricciones de columnas, y gestión de jerarquías.

Fluent API Para esto hicimos uso de Fluent API[3], una forma avanzada de especificar configuraciones del modelo, superior a las Data Annotations que se realizan directamente sobre las clases del dominio.

Fluent API se utiliza sobreescritiendo el método **OnModelCreating** del contexto. Allí especificamos el mapeo de las relaciones, esencial para indicar a EF que ciertos elementos ya existen en el contexto, en los casos de atributos de clase de otros tipos (también almacenados en la base).

Esto se hace especificando características de los DbSets de cada tipo, según las siguientes referencias

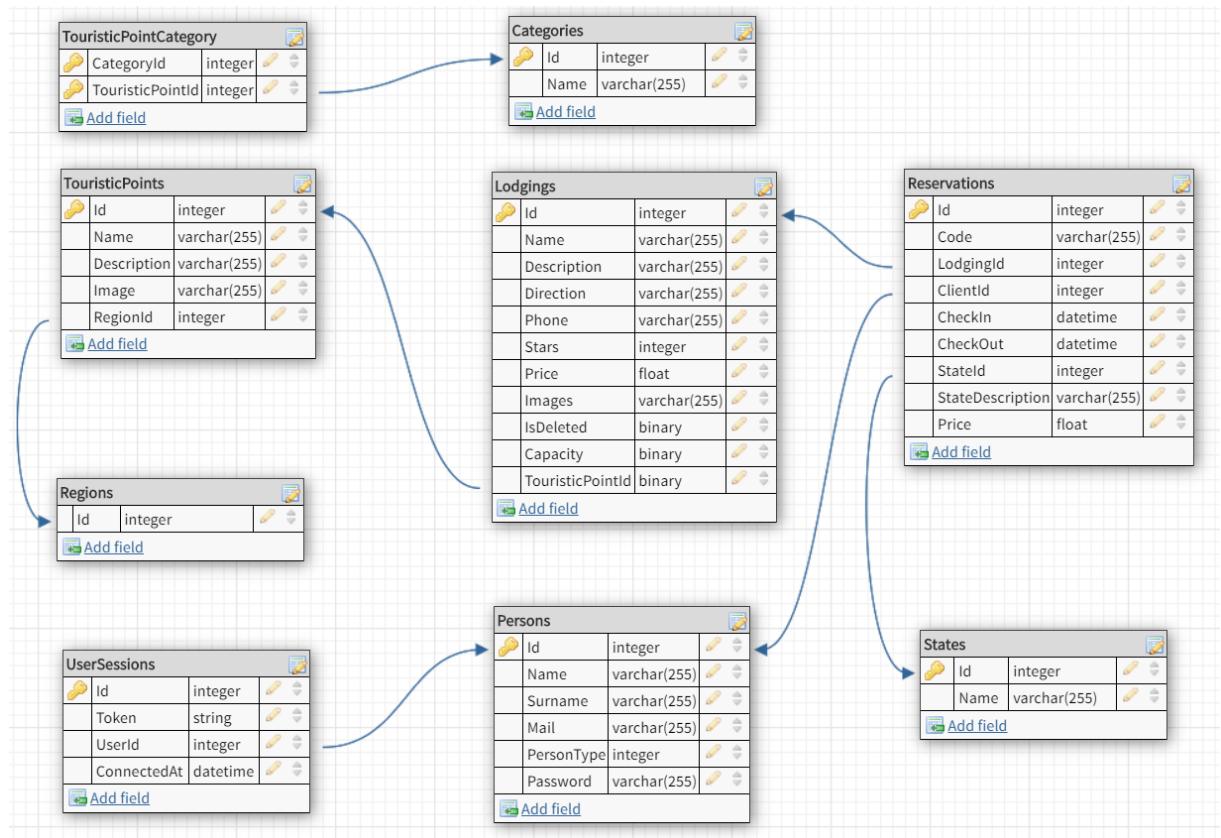
Método	Significado
.HasOne	Debe estar relacionado a un atributo del otro tipo
.HasMany	Puede estar relacionados con muchos atributos del otro tipo
.WithMany	Que a su vez pueden estar relacionados a muchos atributos del primer tipo
.HasKey	Indica las claves de la tabla
.HasForeignKey	Indica las claves foráneas de la tabla

Como se ve a continuación, también fue necesario mapear especialmente la jerarquía de Persons, para indicar que se deseaba almacenar las clases derivadas (Client y Administrator) en la misma tabla, diferenciando un tipo de otro a partir de la columna PersonType.

```
//Inheritance
modelBuilder.Entity<Person>()
    .HasDiscriminator<int>("PersonType")
    .HasValue<Client>(1)
    .HasValue<Administrator>(2);
```

Figura 1.3: Mapeo de Herencia para clase Person

Modelo de Tablas Finalmente, el modelo de tablas generado con EF es el siguiente:



Borrado Lógico Para la eliminación de la entidad hospedaje, optamos por la eliminación lógica. Utilizando un atributo de tipo booleano para indicar el estado deleted (True/False) de dicha entidad, “marcamos” a los elementos eliminados para así no considerarlos a la hora de listar, o buscar en el repositorio.

En el caso de los administradores optamos por utilizar el borrado fisico de la base de datos ya que no habia datos de otras entidades asociados a lo mismos.

```
public class Lodging
{
    public int Id { get; set; }

    public string Name { get; set; }

    public TouristicPoint TouristicPoint { get; set; }

    public string Description { get; set; }

    public string Direction { get; set; }

    public string Phone { get; set; }

    public int Stars { get; set; }

    public double Price { get; set; }

    public string Images { get; set; }

    public bool IsDeleted { get; set; }
    //Capacity in true means that it accepts guests
```

1.3.4. Paquete: WebApplication

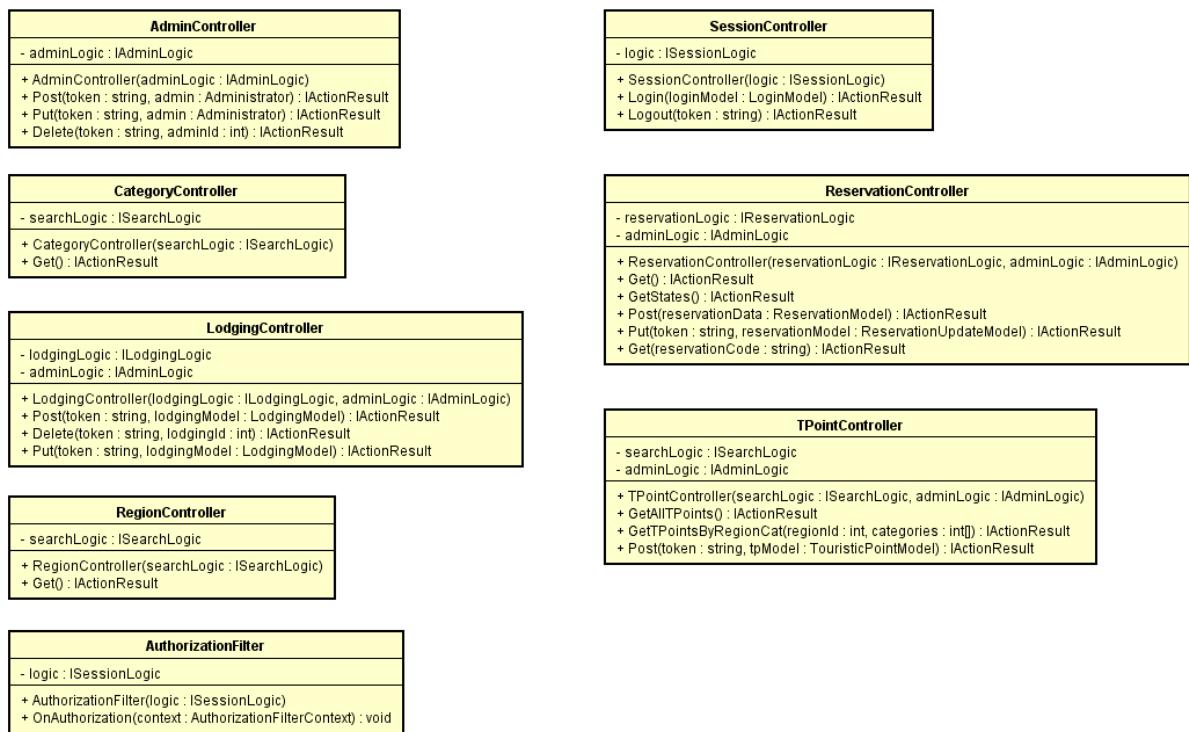


Figura 1.4: Diagrama de Clases - WebApplication

1.4. Diagrama de Componentes

La arquitectura de nuestra aplicación web se compone de 7 componentes básicos, sin tener en cuenta los componentes para las pruebas unitarias y de integración. De los anteriores, 2 de ellos son Interfaces, cuyo rol es agregar capas de indirección para que no exista una dependencia directa entre los paquetes.

El componente de la WebAPI se comunica con el paquete de lógica a través de la interfaz **LogicInterface**.

A su vez, el componente de la Logic se comunica con el paquete de persistencia a través de una interfaz **PersistenceInterface**.

Las dependencias entre los paquetes se analizarán en el diagrama de paquetes.

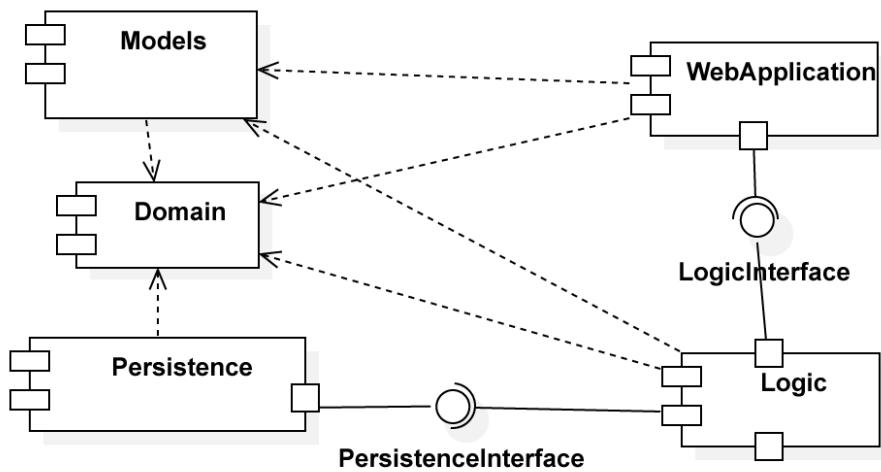


Figura 1.5: Diagrama de Componentes. Vista externa.

1.5. Patrones y Decisiones de Diseño

1.5.1. Inyección de Dependencias

Es una técnica para lograr la inversión de control o el principio de inversión de dependencia entre clases y sus dependencias. Para evitar la dependencias, las clases no crean los objetos que necesitan, sino que se los suministra otra clase que ejerce de arbitro, y que inyecta la implementación deseada a nuestro contrato. En nuestro caso, esta clase es la clase **Startup** del paquete WebAPI.

```
//Dependency injection Repositories
services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
services.AddScoped(typeof(ITPointRepository), typeof(TPointRepository));
services.AddScoped(typeof(IAdminRepository), typeof(AdminRepository));
services.AddScoped(typeof(ILodgingRepository), typeof(LodgingRepository));
services.AddScoped(typeof(IReservationRepository), typeof(ReservationRepository));
//Dependency injection Logic Interfaces
services.AddScoped<ISearchLogic, SearchLogic>();
services.AddScoped<ILodgingLogic, LodgingLogic>();
services.AddScoped<IAdminLogic, AdminLogic>();
services.AddScoped<IReservationLogic, ReservationLogic>();
services.AddScoped<ISessionLogic, SessionLogic>();
services.AddScoped<IPriceCalculator, PriceCalculator>();

services.AddScoped<AuthorizationFilter>();
```

Figura 1.6: Clase Startup.cs. Inyección de Dependencia

En tiempo de compilación, se le indica a la aplicación qué implementación de las interfaces utilizará durante la ejecución.

Mediante el comando *services.AddScoped* se indica el ciclo de vida de los servicios. En una RESTFUL api donde los servicios otorgados no mantienen estados sino que cada request es independiente de la anterior, el ciclo de vida elegido es Scoped ya que implica que se vuelve a crear una instancia de la clase en cuestión para cada request.

Hacer esto además presenta la ventaja de que habilita la utilización de **mocking** para pruebas automáticas, ya que se le puede proveer la instancia mockeada a la clase que se está probando, mediante su constructor.

1.5.2. Patrón Repository

El patrón Repository utiliza una interfaz única implementando una clase **IRepository<T>** para generalizar el comportamiento de los repositorios, independientemente del tipo concreto que se esté gestionando. Mediante firmas con **generics** el patrón establece el contrato de las operaciones básicas que todo repositorio necesita: Get, GetAll, Save, Update, Create, Delete.

Al utilizar las operaciones desde las clases de Logic, se utiliza la implementación **Repository<T>** con un tipo concreto. Así es que cada clase de la lógica tiene un atributo de tipo **Repository<ClaseEspecifica>**.

```

namespace Logic
{
    {
        16 referencias | Felipe, Hace 19 horas | 1 autor, 1 cambio
        public class SearchLogic : ISearchLogic
        {
            private IRepository<Region> _regionRepository;
            private IRepository<Category> _categoryRepository;
            private ITPointRepository _tpointRepository;

            7 referencias | Felipe, Hace 19 horas | 1 autor, 1 cambio
            public SearchLogic(IRepository<Region> regionRepository,
                               IRepository<Category> categoryRepository,
                               ITPointRepository tpointRepository)
            {
                this._regionRepository = regionRepository;
                this._categoryRepository = categoryRepository;
                this._tpointRepository = tpointRepository;
            }
        }
    }
}

```

Figura 1.7: Clase Startup.cs. Inyección de Dependencia

Nota: Para indicarle qué implementación de la interfaz se va a utilizar se hace una inyección de dependencia al igual que se hacía con las implementaciones de la Lógica, como se vio en el punto anterior.

En las situaciones en que se necesitaron algunas implementaciones distintas para un tipo específico, u otras operaciones, se crearon otras interfaces como ITPointRepository, o ILodgingRepository, que luego eran implementadas por clases que a su vez heredaban del Repository<T>para tener todas las operaciones (las comunes y las específicas).

```

namespace Persistence
{
    {
        8 referencias | Felipe, Hace 19 horas | 1 autor, 1 cambio
        public class TPointRepository : Repository<TouristicPoint>, ITPointRepository
        {
            private readonly DbSet<TouristicPoint> DbSet;
            private readonly DbContext context;

            6 referencias | Felipe, Hace 19 horas | 1 autor, 1 cambio
            public TPointRepository(DbContext context) : base(context)
            {
                this.DbSet = context.Set<TouristicPoint>();
                this.context = context;
            }
        }
    }
}

```

Figura 1.8: Clase Startup.cs. Inyección de Dependencia

Generalización del GetAll<T>

En la clase Repository<T>se utilizó una estrategia para generalizar el método GetAll<T>. Al implementar Entity Framework con Lazy Loading (como se profundizará más adelante), al traer objetos de la base de datos, era necesario también incluir sus atributos para seleccionarlos a la vez, haciendo uso del **Include**.

Para hacer esto de forma dinámica y evitar tener un GetAll específico por tipo, agregamos un parámetro a la firma, de tipo []string. En este array se reciben todos los atributos que se quieren incluir en la selección, y con un for loop, se los incluye de a uno en tiempo de ejecución al DBSet.

```

28 referencias | Felipe, Hace 19 horas | 1 autor, 1 cambio
public I Enumerable<T> GetAll(string[] includes)
{
    if (includes.Length > 0)
    {
        for (int i = 1; i < includes.Length; i++)
        {
            DbSet.Include(includes[i]);
        }

        return DbSet.ToList();
    }
    else
    {
        return DbSet.ToList();
    }
}

```

Figura 1.9: Clase Startup.cs. Inyección de Dependencia

1.5.3. Patrón Strategy

El patrón estrategia permite definir una familia de algoritmos, y encapsular uno para que sea el que se utiliza. Haciendo esto, los algoritmos pueden variar independientemente de los clientes que los usan.

En el caso del proyecto actual, el algoritmo que entendemos más probablemente pudiera presentar alteraciones era el de cálculo de precios de las estadías. Esto se debe a que está directamente vinculado a las reglas del negocio que rigen en la actualidad, según las cuales:

- El precio por noche de un hospedaje es constante sin diferenciar fechas. **Es probable que en algun futuro sea de interés presentar distintas tarifas para fines de semana, o ciertas épocas del año**
- Los huéspedes bebés pagan el 25 % de la tarifa y los jóvenes 50 %. **Esto podría cambiar facilmente segun las politicas de la empresa.**
- No se ofrecen promociones ni descuentos segun las formas de pago. **Es esperable que en algun momento se integren distintas opciones de pago que podrian alterar el calculo de la tarifa final**
- No se consideran reglas de negocio de factores externos. **Por ejemplo, la normativa impositiva del Uruguay, que varía a la vez que el PBI**

Para encapsular el algoritmo especificamos una interfaz que provee el servicio de cálculo de precio, pero sin determinar su implementación. En la actualidad solo existe una implementación del algoritmo, pero se podría expandir la funcionalidad fácilmente, o agregar otras implementaciones para instancias como el ambiente de testing, con comportamientos específicos.

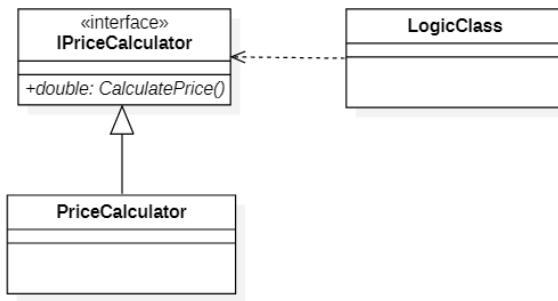


Figura 1.10: Interfaz IPriceCalculator

1.5.4. GRASP

Bajo Acoplamiento

Se trata de asignar responsabilidades a las clases de forma de mantener el acoplamiento entre ellas bajo. Esto se traduce en un menor riesgo a la hora de introducir cambios en una de las clases vinculadas.

Logramos mantener un bajo acoplamiento en nuestras clases de lógica de negocios (Paquete Logic), haciendo uso de la fragmentación del sistema en pequeños “subSistemas”.

Naturalmente, se sigue dando una interconexión lógica entre los mismos al utilizarse unos dentro de otros. Aún así las responsabilidades de cada uno están claramente diferenciadas según las clases que gestionan, y por eso residen en clases separadas.

También dividimos las responsabilidades del paquete Persistence según el tipo de objetos que gestiona cada una. Ciertas entidades del sistema requerían funcionalidades específicas distintas de las operaciones comunes para todos las otras clases del dominio. Por eso se dividió el paquete en varias clases para desacoplar las funcionalidades .

Alta Cohesión

La cohesión es una medida de tan fuertemente relacionadas están las responsabilidades de una clase.

En nuestro caso logramos una alta cohesión en todos nuestros paquetes.

- Paquete Domain: Cada clase del dominio tiene sus responsabilidades claramente asignadas.
- Paquete Models: Se optó por generar un paquete independiente contenido las estructuras de datos necesarias para vincular nuestra interfaz (Web Application) con el dominio, y así ocultar detalles de implementación al exterior.

- Paquete Logic: Se dividió el paquete Logic en **una clase de lógica por clase de dominio**, justamente para lograr esta alta cohesión. Cada clase gestiona y realiza tareas de su respectiva entidad en el dominio.
- Paquete Persistence: Como se comentaba en el punto anterior, también se dividió el paquete persistence en clases. Se profundizará en la explicación del patrón Repository.
- Paquete WebApplication: La WebApplication también tiene dividida sus clases. Se desarrolló un controlador para la gestión de cada entidad. Esto mantiene alta la independencia entre los controladores, y la cohesión de todas las operaciones implicadas cada uno de ellos.

Controlador

El patrón Controlador recomienda tener un responsable claro de manejar los eventos externos al sistema.

Siguiendo este patrón fue que decidimos introducir el paquete **Persistence** exclusivamente dedicado a la manipulación de la base de datos, que se puede considerar como un servicio externo.

Esta clase coordina la actividad en la base de datos, cumpliendo así un rol de **pivote** entre el interior de la aplicación y el servicio exterior.

Polimorfismo

Utilizamos una estructura polimórfica para modelar en nuestro dominio a los clientes y a los administradores como Personas.

En el dominio de nuestro problema, estrictamente solo se debían persistir los administradores, pero optamos por implementar la solución pensando en una futura evolución de la aplicación, en la que los clientes también puedan identificarse frente al sistema.

Por esta razón, la aplicación almacena la información de los clientes al momento que realizan una reserva. Si ya existen, se asocia al cliente pre-existente la nueva reserva, y sino, se crea uno nuevo y se persiste.

Creamos una clase abstracta **Person**, de la que heredan Administrator y Client. Person contiene todos los atributos comunes de ambas alarmas, y la única diferencia de Administrator es que además tiene una contraseña que le permite el acceso.

```

public abstract class Person
{
    9 referencias | Felipe, Hace 19 días | 1 autor, 1 cambio
    public int Id { get; set; }
    29 referencias | santitopo, Hace 23 días | 1 autor, 1 cambio
    public string Name { get; set; }
    18 referencias | santitopo, Hace 23 días | 1 autor, 1 cambio
    public string Surname { get; set; }
    36 referencias | santitopo, Hace 23 días | 1 autor, 1 cambio
    public string Mail { get; set; }

```

Figura 1.11: Person(Clase abstracta)

```

81 referencias | Felipe, Hace 12 días | 2 autores, 2 cambios
public class Administrator : Person
{
    21 referencias | santitopo, Hace 23 días | 1 autor, 1 cambio
    public string Password { get; set; }

    19 referencias | santitopo, Hace 23 días | 1 autor, 1 cambio

```

Figura 1.12: Administrator

1.5.5. SOLID

Los principios [6] S.O.L.I.D refieren a un conjunto de 5 técnicas de diseño orientadas al desarrollo de calidad en [4]P.O.O. Aplicar estas estrategias favorece la mantenibilidad y escalabilidad del software. Fueron introducidas por Robert C. Martin en el año 2000.

- **S - Principio de Responsabilidad Única:** Este principio propone que las clases deberían tener una y solo una razón para cambiar. Se encuentra fuertemente vinculado a la alta cohesión y al bajo acoplamiento buscado en GRASP.

En nuestro trabajo buscamos respetar el principio de responsabilidad única en: 1)La estructuración en sub sistemas en el paquete Logic 2) la separación de clases con responsabilidades específicas en el paquete Persistence, y 3) La separación de los controllers en la WebAPI; uno por endpoint.

- **O - Abierto/Cerrado:** La premisa detrás de este principio es que se debería poder extender el funcionamiento de una clase sin modificarla (abierto al cambio, cerrado a la modificación).

La forma por excelencia de lograr este cometido es mediante la implementación de clases abstractas e interfaces, que se pueden implementar múltiples veces para extender el comportamiento, sin necesidad de realizar cambios en las clases que las utilizan.

Como explicamos en la sección 4.3.3, utilizamos el patrón Strategy para facilitar la modificación del comportamiento del algoritmo de cálculo de precios, a la vez de reducir el impacto en la aplicación de futuros cambios en estas clases.

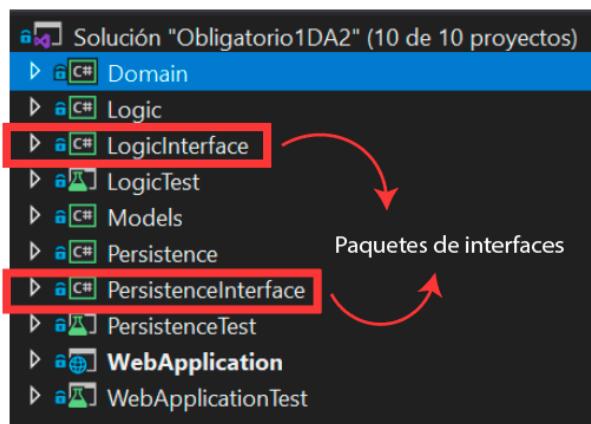
- **L - Principio de sustitución de Liskov:** No nos enfocamos en aplicar particularmente este principio.

- **I - Principio de segregación de Interfaces:** El principio postula que es bueno tener varias interfaces específicas para cada cliente antes que una sola que provea un servicio general. La idea de trasfondo es pensar en el consumidor del servicio a la hora de diseñarlo.

Esta idea fue un pilar fundamental en el diseño de nuestra arquitectura. El diseño implementado hace que el vínculo entre los distintos paquetes suceda estrictamente mediante interfaces.

- Los controladores de la Web API no tienen referencias directas a clases de la lógica, sino que a las interfaces (Contratos) que esta provee.
- La logica tampoco tiene referencias directas a las clases del Repositorio, sino que a las interfaces del paquete Persistence.

Esto también habilitó la realización de pruebas de integración mediante **Mocking** (profundizaremos en esto en la sección de pruebas).



- **D - Principio de Inversión de Dependencia:** Para evitar la dependencia de los servicios externos, este principio indica que se debe utilizar clases contractuales que especifiquen el comportamiento deseado, y que los proveedores de servicio deban implementar. Como se comentó en el punto anterior, toda nuestra arquitectura está basada en la provisión de servicios mediante interfaces.

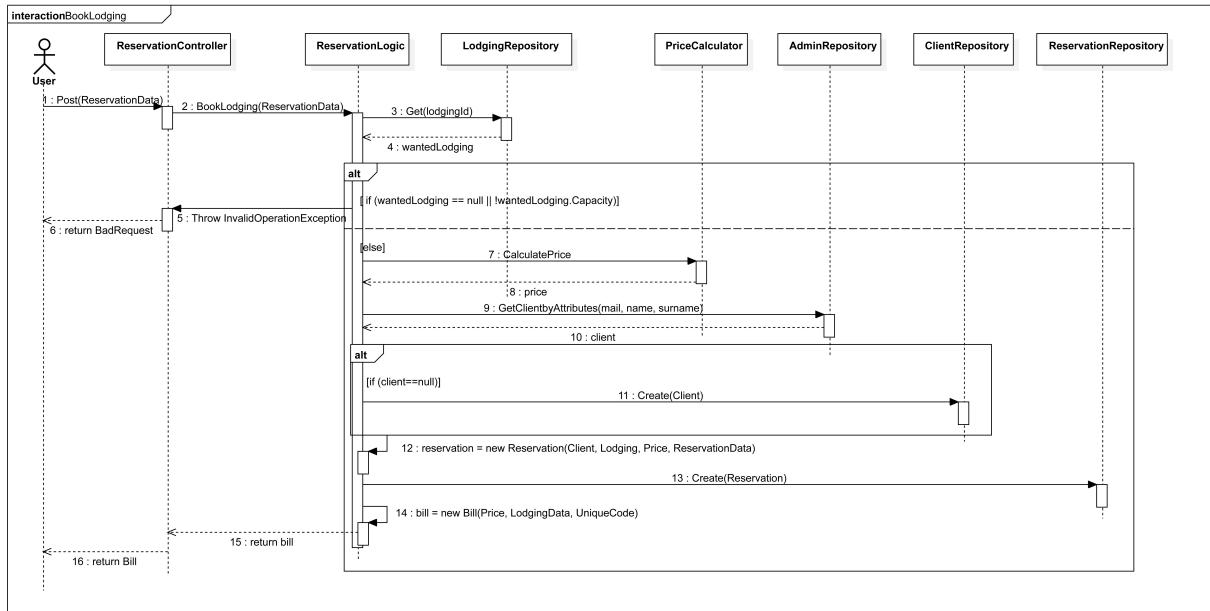
Esta estructura de solución está muy vinculada a la aplicación del patrón Inyección de Dependencias, mencionado previamente en la sección 4.3.1.

1.6. Funcionalidades Claves

Para algunas de las funcionalidades más complejas e importantes del sistema decidimos utilizar diagramas UML de secuencia. De esta manera podemos reflejar el funcionamiento interno con el nivel de abstracción deseado.

Crear una reserva

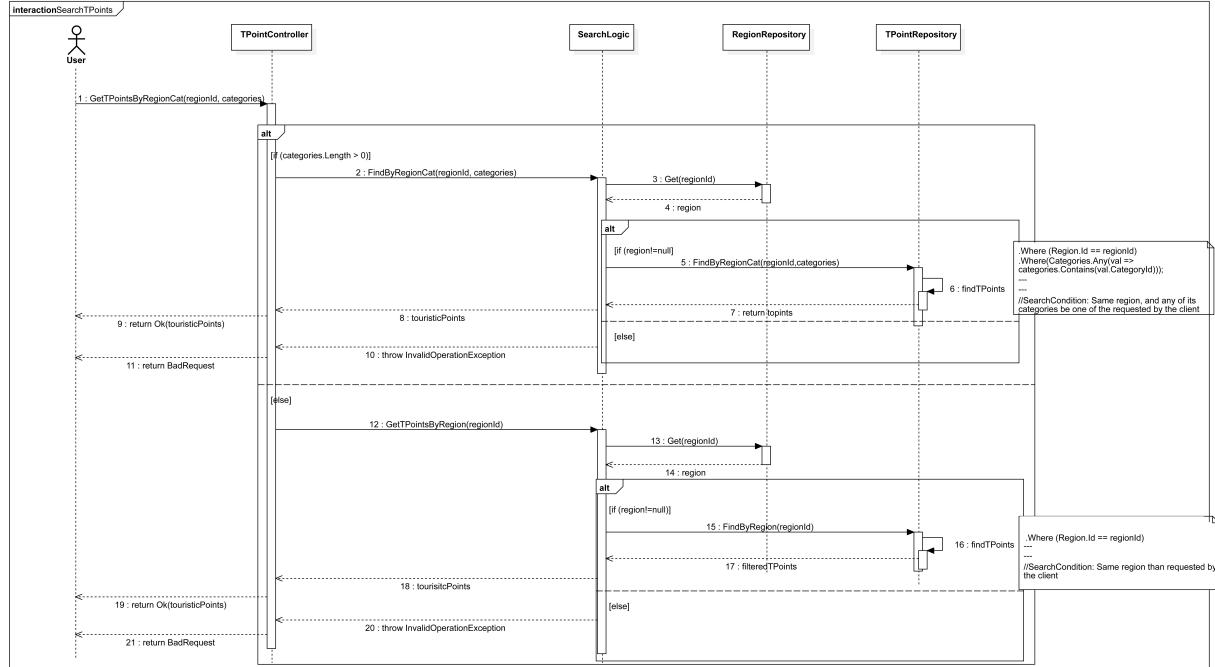
Funcionalidad de creación de una nueva reserva en el sistema.



Búsqueda de Puntos Turísticos por región y categorías (opcionalmente)

Funcionalidad de la API para la búsqueda de puntos turísticos.

Notar que el filtrado de los resultados es responsabilidad del paquete Persistence (no se realiza en memoria).



1.7. Manejo de Excepciones

Para el manejo de errores decidimos lanzar excepciones transversalmente desde los componentes de lógica de negocios y persistencia, capturándolas en los controllers de la WebApplication, para retornar al cliente códigos de error controlados y no dejar que la aplicación corte la ejecución.

En esta versión del programa utilizamos principalmente tres códigos de estado para devolver las excepciones en la Web API.

- 400 : (Bad Request)
- 404 : (Not Found)
- 401 : (Unauthorized)

En la segunda etapa quizás se implementen más funcionalidades y con ellas otros códigos de estado.

```
// POST: /admins
[HttpPost]
[ServiceFilter(typeof(AuthorizationFilter))]

public IActionResult Post([FromHeader] string token, [FromBody] Administrator admin)
{
    try
    {
        Administrator newAdministrator = adminLogic.AddAdmin(admin);
        return Ok(newAdministrator);
    }
    catch (Exception e)
    {
        return BadRequest(e.Message);
    }
}
```

Figura 1.13: Ejemplo: Captura de excepción - WebApplication

2. Evidencia del Diseño y especificación de la API

2.1. Cumplimiento de REST

REST o Representational State Transfer es un estilo de arquitectura de comunicación entre cliente y servidor. El término se originó en el año 2000, y fue introducido por Roy Fielding, para normalizar las arquitecturas de servicios web.

Para satisfacer los requisitos de REST, nuestra aplicación debió satisfacer algunas condiciones listadas a continuación, tomadas de la documentación oficial de REST [1]:

1. **Cliente-Servidor:** Supone la independencia entre cliente y servidor. Ambos deberían poder ser sustituidos sin inconvenientes.

Explicación: Nuestra arquitectura permite una independencia total entre cliente y servidor. Estableciendo una conexión HTTP, **n** clientes podrían consumir la API, de forma desacoplada. Actualmente el cliente utilizado para la realización de pruebas de ejecución es **Postman**. Como se comentó anteriormente, en la próxima etapa del proyecto se desarrollará un front-end cliente para consumir la API.

2. **Sin estados:** Al igual que HTTP, las APIs no guardan estado. Si se requieren privilegios por estar logueado, cada solicitud debería contener la información correspondiente a la autorización.

Explicación: Implementamos un subsistema de autenticación donde los usuarios pueden acceder con su usuario y contraseña y obtener un token único de sesión. Para todas las operaciones que requieren privilegios, se debe incluir en la request un header “token” con su valor.

3. **Interfaz uniforme:** Supone una resolución similar en los distintos endpoints. Manteniendo una coherencia en la forma de interactuar con la API, se facilita la adopción del sistema por parte del usuario que la consuma.

Explicacion: Logramos una coherencia en el manejo de los endpoints apegándonos al estándar para las RESTful APIs. Utilizamos los verbos de HTTP: Post,

Key	Value	Description
token	6F9619FF-8B86-D011-B42D-00C04FC964FF	

Get, Put y Delete, teniendo en cuenta sus comportamientos esperados, y sus convenciones. Por ejemplo, fuimos cuidados en mantener un único GET por resource, en solicitar los datos desde los query params en caso de las operaciones GET. También utilizamos los headers para la autenticación, y hasta dos niveles de ruteo, como indican las recomendaciones.

4. **Sistema basado en capas:** La aplicación en su totalidad se compone de diversos componentes, los cuales se comunican a través de interfaces, de tal modo que ninguno debería tener conocimiento de la estructura interna de los demás, ni con cual se está comunicando realmente.

Explicación: Se estructuró el sistema en componentes que se comunican mediante interfaces para satisfacer este punto. Se puede ahondar en esto en la sección que describe la organización del sistema, en particular en el diagrama de componentes. Como se ve en el mismo, ningun componente tiene conocimiento de la estructura interna de los componentes que utiliza, ya que los utiliza por medio de interfaces.

2.1.1. Recursos

Recurso en el contexto de las WebAPIs, se le llama a cualquier información que pueda ser nombrada. Un recurso puede ser un documento, una imagen, un servicio temporal, una colección, etc. En REST, el recurso tiene un identificador conocido por el cliente, para realizar sobre dicha entidad diferentes acciones. Para ello, se utilizan los verbos HTTP.

En el caso de nuestro sistema, cada recurso publicado tiene asociado un controlador en el paquete WebApplication. Estos son los recursos publicados:

Se observará que los recursos cumplen con las buenas prácticas de REST, siendo todos ellos sustantivos, en plural, en minúscula, e intuitivos ya que gestionan la entidad que lleva su nombre.

- **admins:** Para la gestión de los administradores. Incluye las operaciones de agregado, borrado y modificación de administradores (operaciones que requieren privilegios).

- **categories**: Para la solicitud de todas las categorías. El resto de las operaciones no son soportadas.
- **regions**: Para la solicitud de todas las regiones. El resto de las operaciones no son soportadas.
- **lodgings**: Para la gestión de los hospedajes. Soporta el agregado, eliminado, y modificación (De capacidad), así como también la búsqueda de los mismos por punto turístico, y en fechas determinadas, para la consulta de tarifas (Se puede ahondar en esta operación en la sección de funcionalidades Clave, donde se presenta un diagrama de secuencia mostrando el flujo completo).
- **reservations**: Para la gestión de las reservas. Permite obtener las reservas, los estados de reserva disponibles y actualizar la descripción de estado de una reserva.
- **sessions**: Permite el inicio de sesión y el fin de sesión (elimina la sesión de la base de datos, inhabilitando el token).
- **tpoints**: Permite el agregado de nuevos puntos turísticos, así como la consulta de los mismos por parte de los clientes, a partir de una región y **n** categorías seleccionadas.

Nota: Se optó por trabajar con nomenclatura en inglés para lograr un producto más cercano a la práctica profesional.

2.1.2. Endpoints

Los endpoints son el extremo de la API que está expuesto al cliente, exponiendo los diferentes servicios y comportamientos del sistema. Cada endpoint es la ubicación dentro de la API a la que se va a acceder para interactuar con los recursos necesarios para la función.

A continuación se presenta la documentación de los endpoints del sistema, especificando los parámetros de cada endpoint y su comportamiento:

Verbo	Endpoint	Parámetros	Respuesta	Comportamiento
[POST]	/sessions/login	[Body] Mail, Password	token	Identifica al usuario. Establece la sesión.
[DELETE]	/sessions/logout	[Header] token		Finaliza la sesión indicada

Figura 2.1: Endpoint sessions

Verbo	Endpoint	Parámetros	Respuesta	Comportamiento
[POST]	/admins	[Body] Administrator(*) [Header] token	Administrator	Crea un admin y lo retorna.
[PUT]	/admins	[Body] Administrator [Header] token		Modifica al admin preexistente, a partir del email.
[DELETE]	/admins/{id}	[Header] token, [URI] adminId		Elimina al administrador por id

Figura 2.2: Endpoint admins

Verbo	Endpoint	Parámetros	Respuesta	Comportamiento
[GET]	/categories		[]Category(*)	Devuelve todas las categorías de la base de datos

Figura 2.3: Endpoint categories

Verbo	Endpoint	Parámetros	Respuesta	Comportamiento
[GET]	/regions		[]Region(*)	Devuelve todas las regiones de la base de datos

Figura 2.4: Endpoint regions

Verbo	Endpoint	Parámetros	Respuesta	Comportamiento
[GET]	/tpoints		[]TouristicPoint(*)	Devuelve todos los puntos turísticos de la base de datos
[PUT]	/tpoints/filter	[Query] regionId [Query] [int] categories	[]TouristicPoint	Busca puntos turísticos de la región solicitada. Si se pasaron categorías, también se considera que tengan al menos una en común.
[POST]	/tpoints	[Header] token, [Body] TouristicPointModel(*)	TouristicPoint	Crea un punto turístico y lo retorna.

Figura 2.5: Endpoint tpoints

Verbo	Endpoint	Parámetros	Respuesta	Comportamiento
[GET]	/lodgings/filter	[Query] LodgingsearchModel(*)	[]TouristicPoint(*)	Devuelve todos los puntos turísticos de la base de datos
[POST]	/lodgings	[Header] token, [Body] LodgingModel(*)	Lodging(*)	Busca puntos turísticos de la región solicitada. Si se pasaron categorías, también se considera que tengan al menos una en común.
[DELETE]	/lodgings/{id}	[Header] token, [URI] lodgingId		Elimina al administrador por id
[PUT]	/lodgings	[Body] LodgingModel [Header] token		Modifica el lodging preexistente, ubicándolo a partir del id.

Figura 2.6: Endpoint lodgings

Verbo	Endpoint	Parámetros	Respuesta	Comportamiento
[GET]	/reservations		[] Reservation(*)	Devuelve todas las reservas de la base de datos
[GET]	/reservations/states		[]State (*)	Devuelve todos estados de reserva de la base de datos
[GET]	/reservations/{code}	[URI] reservationCode	StateModel (*)	Busca la reserva y retorna el estado actual
[POST]	/reservations	[Body] ReservationModel(*)	BillModel (*)	Calcula el precio del hospedaje seleccionado según la estrategia de cálculo, y retorna una factura con código único de reserva
[PUT]	/reservations	[Body] ReservationUpdateModel(*) [Header] token		Modifica una reserva, actualizando su descripción y asignándole uno de los estados de la BD.

Figura 2.7: Endpoint reservations

A continuación se deja evidencia de las estructuras de datos no triviales señaladas en la documentación de los endpoints con (*):

Administrator	<pre>public string Name { get; set; } public string Surname { get; set; } public string Mail { get; set; } public string Password { get; set; }</pre>	
Category	<pre>public int Id { get; set; } public string Name { get; set; } public List<TouristicPointsCategory> TouristicPoints { get; set; }</pre>	
TouristicPointCategory	<pre>public int TouristicPointId { get; set; } public TouristicPoint TouristicPoint { get; set; } public int CategoryId { get; set; } public Category Category { get; set; }</pre>	
Region	<pre>public int Id { get; set; } public string Name { get; set; }</pre>	
TouristicPoint	<pre>public int Id { get; set; } public string Name { get; set; } public string Description { get; set; } public Region Region { get; set; } public List<TouristicPointsCategory> Categories { get; set; } public string Image { get; set; }</pre>	
TouristicPointModel	<pre>public string Name { get; set; } public string Description { get; set; } public string Image { get; set; } public int RegionId { get; set; } public int[] Categories { get; set; }</pre>	
LodgingsearchModel	<pre>public int TPointId { get; set; } public string Checkin { get; set; } public string Checkout { get; set; } public int AdultsNum { get; set; } public int ChildsNum { get; set; } public int BabiesNum { get; set; }</pre>	

LodgingModel	<pre>public int Id { get; set; } public string Name { get; set; } public int TPointId { get; set; } public string Description { get; set; } public string Direction { get; set; } public string Phone { get; set; } public int Stars { get; set; } public double Price { get; set; } public string Images { get; set; } public bool IsFull { get; set; }</pre>	
BillModel	<pre>public Guid ReservationCode { get; set; } public string Phone { get; set; } public string Description { get; set; } public double PricePaid { get; set; }</pre>	
LodgingSearchResultModel	<pre>public Lodging Lodging{ get; set; } public double TotalPrice { get; set; }</pre>	
ReservationModel	<pre>public int LodgingId { get; set; } public string Checkin { get; set; } public string Checkout { get; set; } public int AdultsNum { get; set; } public int ChildrensNum { get; set; } public int BabiesNum { get; set; } public string Name { get; set; } public string Surname { get; set; } public string Mail { get; set; }</pre>	
ReservationUpdateModel	<pre>public int ReservationId{ get; set; } public int StateId { get; set; } public string StateDescription { get; set; }</pre>	
StateModel	<pre>public string Name { get; set; } public string Description { get; set; }</pre>	
State	<pre>public int Id { get; set; } public string Name { get; set; }</pre>	

2.1.3. Mecanismos de autenticación

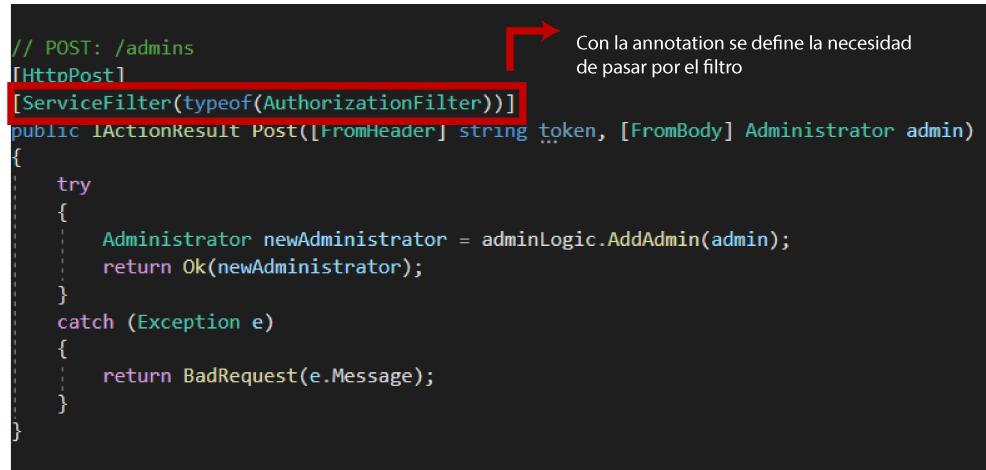
En cualquier sistema basado en la arquitectura REST conviven funcionalidades orientadas a distintos perfiles de usuarios, con distintos roles y privilegios. En nuestro sistema, todas las funcionalidades de mantenimiento de las entidades, de agregado, modificación y eliminación, así como también en algunos casos la selección de información protegida (como las reservas), son responsabilidad de los administradores, y no se desea que se pueda acceder por parte de un usuario común.

Como todas estas operaciones comparten los recursos (por ejemplo la consulta del estado de una reserva pensada para los usuarios, y la actualización del estado de una reserva pensada para los administradores) fue necesario implementar mecanismos de control de acceso a las funcionalidades deseadas.

Para esto se implementaron los filtros. Los filtros nos permiten ejecutar código antes o después de determinadas fases en el procesamiento de una solicitud HTTP. Cada tipo de filtro es ejecutado en una fase diferente de la solicitud, es decir tienen un orden de ejecución según su responsabilidad. En nuestro caso solamente se utilizó el filtro de Autorización, cuya función es ejecutarse previo a la entrada al método o clase en cuestión, para corroborar la validez de la sesión.

Se crea el filtro heredando de Attribute (esto permite crear una anotación para invocar luego en los controladores), e implementando la interfaz IAuthorizationFilter. El filtro se configuró para que realice la validación sobre el token que se recibe por header. Primero se verifica que no sea nulo, y luego que efectivamente exista en

la tabla de sesiones activas. Si cualquiera de estas condiciones falla, se devuelve al cliente un código de error.



```
// POST: /admins
[HttpPost]
[ServiceFilter(typeof(AuthorizationFilter))]
public IActionResult Post([FromHeader] string token, [FromBody] Administrator admin)
{
    try
    {
        Administrator newAdministrator = adminLogic.AddAdmin(admin);
        return Ok(newAdministrator);
    }
    catch (Exception e)
    {
        return BadRequest(e.Message);
    }
}
```

Con la annotation se define la necesidad de pasar por el filtro

Figura 2.8: Aplicación del filtro en método AdminController

```

public class AuthorizationFilter : Attribute, IAuthorizationFilter
{
    private ISessionLogic logic;
    public AuthorizationFilter(ISessionLogic logic)
    {
        this.logic = logic;
    }
    public void OnAuthorization(AuthorizationFilterContext context)
    {
        string token = context.HttpContext.Request.Headers["token"];
        if (token == null)
        {
            context.Result = new ContentResult()
            {
                StatusCode = 401,
                Content = "Error. No existe la sesión."
            };
            return;
        }
        if (!logic.IsLogued(token))
        {
            context.Result = new ContentResult()
            {
                StatusCode = 403,
                Content = "Error. No estas logueado."
            };
            return;
        }
    }
}

```

Herencia de clase Attribute e implementación de interfaz IAuthorizationFilter

Si el header token viene vacío se retorna código 401

Si la sesión no existe se retorna código 403

Figura 2.9: Configuración del filtro

2.1.4. Códigos de Error HTTP

Como se comentará más adelante en la sección de manejo de excepciones, el sistema comunica los fallos producidos en tiempo de ejecución en todo el sistema, a través de códigos de error HTTP retornados en los endpoints.

En esta versión del programa utilizamos principalmente tres códigos de estado para devolver las excepciones en la Web API.

- 400 : (Bad Request)
- 404 : (Not Found)
- 401 : (Unauthorized)

3. Evidencia de la aplicación de TDD y Clean Code

3.1. Clean Code

La calidad del sistema está dada en gran parte por el nivel de prolijidad del código. Para mantener un código de alta calidad y limpio, utilizamos algunas buenas prácticas mencionadas en el libro “Clean Code” de Robert Martin.

Aquí las prácticas que se aplicaron sobre el sistema:

■ Nombres

- Nombres nemotécnicos: La mayoría de las variables del sistema utilizan nombres nemotécnicos, excepto el paquete tests donde se realizan pruebas unitarias.
- Desinformación: Se evitó la desinformación de colecciones de objetos, algunas de las colecciones eran llamadas listas pero eran arrays.
- Nombres pronunciables: Se tuvo en cuenta que los nombres de variables y métodos sean pronunciables
- Utilización de constantes: Se utilizaron constates para algunas clases del sistema. Por ejemplo para el nombre del estado de reserva por defecto.
- Nombres de Métodos: Siempre se incluye un verbo para nombrar a cada uno de los métodos y se trató de asignar nombres que describan la tarea a realizar. Todos los métodos comienzan con mayúscula según estándar del lenguaje.

■ Funciones

- Única Tarea: Un método realiza una única tarea.
- Step-Down Rule: Las funciones se leen desde arriba hacia abajo.
- Pocos parámetros: Se intentó mantener funciones con pocos parámetros. Para reducir la cantidad de argumentos se implementaron los modelos, que encapsulan mucha información en un solo objeto.

■ Comentarios

- Código auto explicativo: Se intento hacer el código lo mas explicativo posible, evitando utilizar comentarios de no ser necesarios.
- Clarificación: Para algunos métodos fue necesario clarificar el código utilizando comentarios objetivos y concisos.
- Evitamos marcadores de posición / agrupadores.

■ Formato

- Formato vertical
 - Apertura vertical: Los métodos se separaron 1 linea verticalmente.
 - Declaración de atributos al comienzo.
 - Ordenamiento de funciones por dependencia.
- Formato horizontal
 - Largo de lineas: La mayoría de las lineas tienen un largo menor a 100 caracteres
 - Identación: Se utilizo una correcta identación de métodos y clases.
 - Llaves según del estándar del lenguaje

■ Manejo de Errores

- Se capturaron los errores utilizando try-catchs para no mostrar fallos al usuario. En los casos donde se sabe se pueden producir fallos, se intentó dar mensajes de error claros.

3.2. TDD

El Test-Driven Development (desarrollo dirigido por tests) es una práctica de programación que consiste en escribir primero las pruebas (generalmente unitarias), después escribir el código fuente que pase la prueba satisfactoriamente y, por último, refactorizar el código escrito.

Con esta práctica se conseguimos un código más robusto, más seguro, más mantible y una mayor rapidez en el desarrollo.

3.2.1. Adaptación de TDD y evidencia de uso

En nuestro proyecto nos apegamos a la metodología de trabajo propuesta por TDD, según la cual todos los métodos se desarrollan para satisfacer los casos de prueba previamente diseñados. Sin embargo, por una cuestión de optimización del tiempo, no fuimos meticulosos a la hora de commitear para dejar constancia de cada iteración. En cambio, “adaptamos” la metodología para poder trabajar con ella de una manera más ágil. Por esta razón la evidencia de aplicación de TDD que se puede verificar es que en cada commit en el historial del repositorio cumple la condición de integrar de forma conjunta funcionalidades y sus respectivas pruebas unitarias.

3.2.2. Enfoque

En nuestro proyecto utilizamos distintos enfoques para las pruebas unitarias.

Por un lado, para las pruebas de los paquetes WebAPI y Logic, utilizamos el enfoque denominado “Outside-In”. Esta estrategia se utiliza en los escenarios cuando el objetivo central es verificar que el comportamiento de los objetos es el esperado, y no sus estados en sí mismo. Este abordaje permite ahorrar el trabajo de crear y mantener objetos reales sustituyéndolos por dobles de test (“**mocks**”).

Por otro lado, para las pruebas del paquete de Persistencia, trabajamos con una base de datos de prueba como se explicó en la sección de la documentación correspondiente al Testing. En estas circunstancias optamos por crear y mantener objetos, trabajando con el enfoque Inside-Outza que el objetivo central era verificar el estado de los objetos, ya que las colaboraciones entre los objetos eran de menor complejidad. En la imagen se puede observar el uso de los Assert para corroborar estados.

```

[TestMethod]
0 referencias | Felipe, Hace 18 horas | 1 autor, 1 cambio
public void SearchLodgings()
{
    var mock1 = new Mock<ILodgingRepository>(MockBehavior.Strict);
    var mock2 = new Mock<IPriceCalculator>(MockBehavior.Strict);
    LodgingLogic logic = new LodgingLogic(mock1.Object, mock2.Object);

    List<Lodging> lst = new List<Lodging>();
    TouristicPoint tpoint = new TouristicPoint()...;
    Lodging lod1 = new Lodging()...;
    Lodging lod2 = new Lodging()...;
    lst.Add(lod1);
    lst.Add(lod2);

    LodgingSearchModel search = new LodgingsearchModel()...;
    mock1.Setup(x => x.FindByTPoint(search.TPointId)).Returns(lst);
    mock2.Setup(x => x.CalculatePrice(It.IsAny<LodgingsearchModel>(), 20.0)).Returns(40.0);

    IEnumerable<LodgingsearchModel> ret = logic.SearchLodgings(search);

    mock1.VerifyAll();
    Assert.AreEqual(ret.Count(), 1);
    Assert.AreEqual(ret.ElementAt(0).TotalPrice, 40.0);
}

```

Figura 3.1: Prueba OutsideIn

```

[TestMethod]
0 referencias | Felipe, Hace 18 horas | 1 autor, 1 cambio
public void FindByTPoint()
{
    var options = new DbContextOptionsBuilder<UyNaturalContext>()
        .UseInMemoryDatabase(databaseName: "TestDB")
        .Options;

    using (var context = new UyNaturalContext(options))
    {
        var repository = new LodgingRepository(context);
        Region region1 = new Region()...;
        Region region2 = new Region()...;
        TouristicPoint tpoint1 = new TouristicPoint()...;
        TouristicPoint tpoint2 = new TouristicPoint()...;
        Lodging lod1 = new Lodging()...;
        Lodging lod2 = new Lodging()...;

        context.Set<Region>().Add(region1);
        context.Set<Region>().Add(region2);
        context.Set<TouristicPoint>().Add(tpoint1);
        context.Set<TouristicPoint>().Add(tpoint2);
        context.Set<Lodging>().Add(lod1);
        context.Set<Lodging>().Add(lod2);
        context.SaveChanges();

        IEnumerable<Lodging> tpoints = repository.FindByTPoint(1);
        Assert.AreEqual(1, tpoints.Count());
        Assert.AreEqual(tpoints.ElementAt(0).Id, 1);

        context.Set<Region>().Remove(region1);
        context.Set<Region>().Remove(region2);
        context.Set<TouristicPoint>().Remove(tpoint1);
        context.Set<TouristicPoint>().Remove(tpoint2);
        context.Set<Lodging>().Remove(lod1);
        context.SaveChanges();
    }
}

```

Figura 3.2: Prueba InsideOut

3.3. Repositorio GitHub

Adjuntamos el link al repositorio de GitHub donde se realizó el desarrollo de la aplicación. Este repositorio se encuentra dentro de la organización ORT-DA2.

<https://github.com/ORT-DA2/DA2TopolanskyNajson.git>

3.4. Base de datos secundaria

Utilizamos una base de datos secundarias creada en memoria para realizar los tests del paquete “Persistence”, decidimos utilizar otra base para estas pruebas para así, no impactar directamente en la base real y sobrecargar la misma.

Para comprobar que las pruebas funcionan de manera correcta, realizamos una “limpieza” a la base de datos cada vez que se finalizaba una prueba unitaria.

```
[TestMethod]  
  
public void GetDefaultStateCreateIt()  
{  
  
    var options = new DbContextOptionsBuilder<UyNaturalContext>()  
        .UseInMemoryDatabase(databaseName: "TestDB") CREACIÓN  
        .Options;  
  
    using (var context = new UyNaturalContext(options))  
    {  
        var repository = new ReservationRepository(context);  
  
        State ret = repository.GetDefaultState();  
  
        State check = context.Set<State>().Where(x => x.Name == Constants.  
            Assert.AreEqual(ret.Name, check.Name);  
            Assert.AreEqual(ret.Id, check.Id);  
  
            LIMPIEZA  
            context.Set<State>().Remove(check);  
            context.SaveChanges();  
    }  
}
```

Figura 3.3: Ejemplo Prueba Unitaria en PersistenceTest

3.5. Cobertura de Pruebas Unitarias

Estas pruebas consisten en aislar una parte del código y comprobar que funciona a la perfección. Son pequeños tests que validan el comportamiento de una funcionalidad particular de una clase y su lógica aislada.

Estas resultaron de gran ayuda, ya que con ellas pudimos detectar errores que hubieran sido más difícil de detectar en fases más avanzadas del proyecto.

En total se realizaron **109 pruebas unitarias** sobre el sistema, enfocándonos en los paquetes Logic, Persistence y WebApplication.

En estas pruebas se intento probar todos los flujos posibles de la aplicación entre ellas: comportamiento esperado, manejo de excepciones, mensajes de estado, etc

Prueba	Duración	Rasgos	Mensaje de error
LogicTest (39)	634 ms		
LogicTest (39)	634 ms		
AdminLogicTest (14)	523 ms		
LodgingLogicTest (1)	28 ms		
PriceCalculatorTest (4)	6 ms		
ReservationLogicTest (7)	48 ms		
SearchLogicTest (7)	15 ms		
SessionLogicTest (6)	14 ms		
PersistenceTest (30)	1,7 s		
PersistenceTest (30)	1,7 s		
AdminRepositoryTest (6)	1,3 s		
LodgingRepositoryTest (9)	179 ms		
RegionRepositoryTest (1)	17 ms		
RepositoryTest (4)	39 ms		
ReservationRepositoryTest (4)	39 ms		
TPointRepositoryTest (6)	168 ms		
WebApplicationTest (40)	479 ms		
WebApplicationTest (40)	479 ms		
AdminControllerTest (6)	391 ms		
AuthorizationFilterTest (3)	34 ms		
CategoryControllerTest (2)	7 ms		
LodgingControllerTest (7)	10 ms		
RegionControllerTest (1)	1 ms		
ReservationControllerTest (10)	21 ms		
SessionControllerTest (4)	5 ms		
TPointControllerTest (7)	10 ms		

Figura 3.4: Evidencia de pruebas - Visual Studio 2019

Para realizar la cobertura de pruebas unitarias utilizamos la herramienta integrada de Visual Studio 2019, con esta herramienta se obtuvo un porcentaje de **cobertura de mayor al 95 %** sobre los paquetes Logic, Persistence y WebApplication (Filters y Controllers).

Las pruebas unitarias demuestran que la WebAPI, lógica, persistencia de la aplicación se encuentran estable y verificada, y que funcionará en la mayoría de los casos.

Jerarquía	No cubiertos (bloques)	No cubiertos (% de bloques)	Cubiertos (bloques)	Cubiertos (% de bloques)
felip_FELIPE-PC 2020-10-13 17...	1003	18,57 %	4398	81,43 %
▷ domain.dll	101	49,03 %	105	50,97 %
▷ logic.dll	15	4,78 %	299	95,22 %
▷ { } Logic	15	4,78 %	299	95,22 %
▷ logictest.dll	40	2,76 %	1410	97,24 %
▷ { } LogicTest	40	2,76 %	1410	97,24 %
▷ models.dll	14	12,50 %	98	87,50 %
▷ { } Models	14	12,50 %	98	87,50 %
▷ persistence.dll	767	61,56 %	479	38,44 %
▷ { } Persistence	8	1,64 %	479	98,36 %
▷ { } Persistence.Migrations	759	100,00 %	0	0,00 %
▷ persistencetest.dll	0	0,00 %	887	100,00 %
▷ webapplication.dll	66	24,91 %	199	75,09 %
▷ { } Filters	0	0,00 %	23	100,00 %
▷ { } WebApplication	66	100,00 %	0	0,00 %
▷ { } WebApplication.Control...	0	0,00 %	176	100,00 %
▷ webapplicationtest.dll	0	0,00 %	921	100,00 %

Figura 3.5: Cobertura de Pruebas Unitarias - Visual Studio 2019

Nota: No se realizaron pruebas unitarias sobre las clases del dominio ya que las mismas no contienen métodos y solo properties.

4. Evidencia de la ejecución de las pruebas de la API con Postman

4.1. Casos de Prueba

Nota 1: Al crear la base de datos, esta ya contiene una sesión iniciada por el administrador “admin” para realizar los requests de Postman. Este token por defecto fue el que se utilizó para realizar las siguientes pruebas. El mismo se puede modificar desde la colección para realizar otras pruebas.

Nota 2: Si bien hay pruebas que no pasan, se profundiza esto en la sección “Deuda técnica”. Las pruebas que NO pasan están marcadas en gris.

Funcionalidad	Tipo de Prueba	Entrada	Resultado
Buscar hospedajes para un cierto punto turístico con los parámetros especificados	Caso Uso Normal	TPointId = 1 Checkin = 15/10/2019 Checkout = 16/10/2019 AdultsNum = 2 ChildsNum = 2 BabiesNum = 4	OK
	Valores Inválidos	TPointId = 0 Checkin = 15/10/2019 Checkout = 16/10/2019 AdultsNum = 2 ChildsNum = -1 BabiesNum = 4	OK
	Valores Inválidos	TPointId = 1 Checkin = 11/10/2019 Checkout = 13/10/2019 AdultsNum = 2 ChildsNum = 2 BabiesNum = 4	OK
	Valores Límites	TPointId = 8 Checkin = 15/10/2019 Checkout = 16/10/2019 AdultsNum = 99 ChildsNum = 99 BabiesNum = 99	OK
	Omisión de campos	TPointId = 10 Checkin = 11/10/2019 Checkout = 14/10/2019	OK
	Query Vacía		Retorna Vacío
Realizar una reserva de un hospedaje.	Caso Uso Normal	LodgingId = 1 Checkin = 16/10/2020 Checkout = 18/10/2020 AdultsNum = 1 ChildsNum = 1 BabiesNum = 1 Name = Santiago Surname = Topolansky Mail = s@gmail.com	OK
	Valores Inválidos	LodgingId = 0 Checkin = 16/10/2020 Checkout = 18/10/2020 AdultsNum = 1 ChildsNum = 1 BabiesNum = 1 Name = Santiago Surname = Topolansky	Excepción de sistema

		Mail = s@gmail.com	
	Valores Inválidos	LodgingId = 1 Checkin = 16/10/2020 Checkout = 18/10/2020 AdultsNum = -1 ChildsNum = -2 BabiesNum = 1 Name = Santiago Surname = Topolansky Mail = s@gmail.com	Realiza la reserva
	Valores Inválidos	LodgingId = 1 Checkin = 10/10/2020 Checkout = 12/10/2020 AdultsNum = 1 ChildsNum = 2 BabiesNum = 5 Name = Federico Surname = Rodriguez Mail = federo@gmail.com	Realiza la reserva
	Valores Límites	LodgingId = 1 Checkin = 16/10/2020 Checkout = 18/10/2020 AdultsNum = 99 ChildsNum = 99 BabiesNum = 50 Name = Federico Surname = Rodriguez Mail = federo@gmail.com	OK
	Omisión de campos	LodgingId = 1 Checkin = 16/10/2020 Checkout = 18/10/2020 AdultsNum = 1 ChildsNum = 1 BabiesNum = 1	Excepción de sistema
	Body Vacío		Excepción de sistema
Dar de alta un nuevo hospedaje o borrar uno existente, para un punto turístico existente	Caso Uso Normal Alta	Name = Castillo Piria Description = Excelente lugar para vacacionar Direction = Piriapolis 1234 Phone = 091657062 Images = img1.png Price = 22.50 Stars= 4 TPointId = 1	OK
	Alta No Autorizado	Name = Castillo Piria Description = Excelente lugar para vacacionar Direction = Piriapolis 1234 Phone = 091657062 Images = img1.png	Excepción del sistema

		Price = 22.50 Stars= 4 TPointId = 1 No existe token en header	
	Agrego hospedaje existente	Name = Castillo Piria Description = Excelente lugar para vacacionar Direction = Piriapolis 1234 Phone = 091657062 Images = img1.png Price = 22.50 Stars= 4 TPointId = 1	Lanza excepción
	Valores Inválidos	Name = Castillo Hermano de Piria Description = Excelente lugar para vacacionar Direction = Piriapolis 1234 Phone = 091657062 Images = img1.png Price = -10 Stars= 25 TPointId = 1	Agrega hospedaje
	Omisión de campos	Name = Ranchito 18 Description = Rancho Direction = 18 de Julio 1234 Phone = 091657062	Agrega hospedaje
	Body Vacío		Excepción De sistema
	Caso Uso Normal Baja	Id = 1	OK
	Baja no autorizado	Id = 1 No existe token en header	Excepción del sistema
	Caso Invalido Baja	Id = 99 (No existe)	Excepción del sistema
Modificar la capacidad actual de un hospedaje.	Caso Uso Normal	Id = 2 IsFull = false	OK
	No autorizado	Id = 2 IsFull = false No existe token en header	Excepción del Sistema
	Valores Inválidos	Id = 0 IsFull = true	Excepción del Sistema
	Omisión de campos	Id = 1	Deja con capacidad por defecto

			(Capacity = true)
	Body Vacío		Excepción del Sistema
Cambiar el estado de una reserva, indicando una descripción	Caso Uso Normal	ReservationId = 1 StatId = 3 StateDescription = Completado	OK
	No Autorizado	ReservationId = 1 StatId = 3 StateDescription = Completado No existe token en header	Excepción del Sistema
	Valores Inválidos	ReservationId = 0 StatId = 4 StateDescription = Rechazada, tarjeta invalida	Excepción del Sistema
	Valores Límites	ReservationId = 1 StatId = 5 StateDescription = La reserva ha expirado	OK
	Omisión de campos	ReservationId = 1 StateDescription = La reserva ha expirado	Excepción del Sistema
	Omisión de campos	ReservationId = 1 StatId = 5	OK
	Body Vacío		Excepción del Sistema

4.2. Evidencia de ejecución de pruebas

Adjuntamos el link al video subido a la plataforma YouTube, donde se muestra la evidencia de la ejecución de las pruebas.

https://youtu.be/0yq4_kYnqSA

5. Conclusiones

5.1. Deuda Técnica

Al momento de la entrega todos los bugs detectados han sido corregidos. No tenemos conocimiento de ninguna funcionalidad que no funcione correctamente según lo especificado.

Comentario: Aunque estamos conformes con el producto logrado, en las últimas semanas de desarrollo identificamos distintos puntos de potencial mejora en la calidad del diseño de nuestra aplicación. De forma consciente y estratégica hemos priorizado y optado por lograr una última versión funcional y aceptable desde todas las distintas perspectivas, así como una documentación sólida, aunque esto implicara dejar pendiente cierta deuda técnica que se detalla a continuación (para tomar en consideración en futuras versiones de la aplicación...):

- **Imágenes de Hospedajes.** e planea mejorar el manejo de las imágenes de los hospedajes en la siguiente etapa, al implementar el front end. Actualmente se cuenta con un atributo único de tipo string, donde se definen la/s ruta/s de la/s imagen/es de los hospedajes (en el caso de ser varias, se espera se envíen concatenadas).

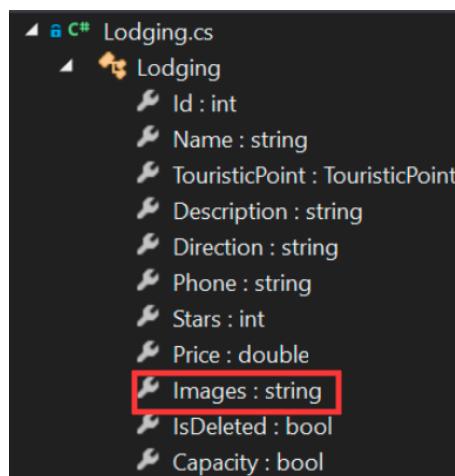


Figura 5.1: “Images” de tipo string

- **Códigos de Error HTTP** El manejo de excepciones de cara al cliente que se implementó, aunque es funcional, no aporta información realmente específica

y por ende útil a quien consume la API, para resolver los fallos ocurridos. Esto se debe a que utilizamos solo algunos pocos códigos HTTP: el 400 (de BadRequest), el 404 (de Not Found) y el 401 (de Unauthorized)

Además, mezclamos la captura de las excepciones generales del sistema con las excepciones de reglas de negocio, por lo que el cliente podría llegar a recibir un mensaje de error incomprensible.

En una próxima versión de la aplicación se debería priorizar la gestión de las excepciones teniendo en cuenta los distintos códigos de error que provee HTTP. Aunque existen más de 70 códigos de error para casos muy específicos, los 8 más comunes son los siguientes:

- 200 - Ok
- 400 - Bad Request
- 500 - Internal Server Error
- 201 - Created
- 304 - Not Modified
- 404 - Not Found
- 401 - Unauthorized
- 403 - Forbidden

```
// POST: /lodgings/1
[HttpDelete("{lodgingId}")]
[ServiceFilter(typeof(AuthorizationFilter))]
public IActionResult Delete([FromHeader] string token)
{
    try
    {
        adminLogic.RemoveLodging(lodgingId);
        return Ok();
    }
    catch (Exception e)
    {
        return BadRequest(e.Message); → Código HTTP 400.
    }
}
```

No distingue entre el tipo de error
y se retorna el mensaje

Figura 5.2: Ejemplo de BadRequest (Código 400)

- **Validaciones** En esta primera etapa del proyecto, debido a la limitante del tiempo, priorizamos lograr el correcto funcionamiento de todas las operaciones, y no pudimos lograr una robustez total en cuanto a validaciones de reglas negocio ni en los controladores de la API. Este punto fue especialmente alarmante a la hora de ejecutar las pruebas de casos de uso, donde quedó en evidencia que muchos casos límite con datos incoherentes, o parámetros faltantes en ciertos

endpoints, no eran considerados por la aplicación como erróneos.

Sin embargo, quita gravedad al asunto el hecho de que la webAPI será consumida por un cliente desarrollado por nuestro propio equipo, y que se operará mediante una interfaz gráfica, por lo que el sistema final no debería resultar tan propenso a errores si se realizaran validaciones en front-end. De cualquier manera, el aumento de las validaciones es una tarea que se considerará **prioritaria** en la siguiente etapa del proyecto.

5.2. Reflexión Final

Ya habiendo culminado esta primera etapa del proyecto, **UYNatural**, consideramos que pudimos cumplir con los objetivos de la primera entrega del obligatorio. Desarrollamos una WebAPI utilizando ASP.NET, un framework con el que ninguno de los integrantes del equipo tenía experiencia previa. Además, tuvimos que familiarizarnos con una arquitectura novedosa, REST, comprendiéndola desde un punto de vista teórico a la vez que se hacía un esfuerzo por implementarla en el terreno práctico, respetando sus buenas prácticas y restricciones.

Nos parece importante destacar que supimos tomar decisiones de diseño bien fundadas y con agilidad, luego de haber transcurrido el curso previo de Diseño de Aplicaciones 1, lo que consideramos una evolución en la calidad de desarrollo de nuestro equipo. La aplicación de patrones de diseño en momentos oportunos, y la atención dedicada en dejar puntos de extensión para el sistema, fueron logros importantes.

Creemos que logramos un producto final que satisface los requerimientos especificados y en la mayor medida posible respeta las decisiones de diseño más importantes comentadas en el curso, como la organización del proyecto en capas y la aplicación de varios de los patrones estudiados. Aplicamos ciertas metodologías de desarrollo que profesionalizaron el proceso de trabajo, como TDD para el desarrollo del sistema, o **GitFlow** como proceso de SCM para el manejo consistente del repositorio y el control de versiones. Siguiendo por la misma línea, tomamos como premisa el desarrollo completo en **idioma Inglés**, incluyendo comentarios y commits.

Por último, evaluamos positivo nuestro desempeño como equipo de trabajo, ya que pudimos distribuir las responsabilidades de forma equitativa y realizar un trabajo continuo y constante a lo largo del plazo otorgado.

Bibliografía

- [1] <https://restfulapi.net/>. What is rest.
- [2] Entity Framework Tutorial. Entity framework.
- [3] Tutorialspoint. Fluent API.
- [4] Wikipedia. Programacion Orientada a Objetos, conceptos fundamentales.
- [5] tuprogramacion.com. ¿Qué es LINQ?
- [6] M. Azevedo. S.O.L.I.D principles: what are they and why projects should use them.