

Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 2

Obligatorio 2

Felipe Najson (232863)
Santiago Topolansky (228360)

Repositorio:

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

Entregado como requisito de la materia Diseño de
Aplicaciones 2

26 de noviembre 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. Introducción	4
1.1. Descripción general del trabajo y del sistema	5
2. Descripción del Diseño	9
2.1. Vista de Desarrollo	10
2.1.1. Diagrama de Componentes	12
2.2. Vista Lógica	13
2.2.1. Paquete: Domain	13
2.2.2. Paquetes: Logic y LogicInterface	15
2.2.3. Paquetes: Persistence y PersistenceInterface	16
2.2.4. Paquete: WebApplication	20
2.2.5. Funcionalidades Clave	21
2.3. Patrones y Estrategias de Diseño	23
2.3.1. Inyección de Dependencias	23
2.3.2. Patrón Repository	24
2.3.3. Patrón Strategy	26
2.3.4. Reflection	26
2.3.5. GRASP	27
2.3.6. SOLID	29
2.3.7. Clean Code	31
2.4. Manejo de Excepciones	33
2.5. Front-End	34
2.5.1. Estructura de la solución	34
2.5.2. Material Design	35
3. Conclusiones	37
3.1. Mejoras del Diseño	37
3.2. Deuda Técnica	40
3.3. Métricas	42
3.3.1. Cohesión Relacional (H)	42
3.3.2. Inestabilidad (I)	43
3.3.3. Abstracción (A)	44
3.3.4. Distancia (D)	44
3.3.5. Conclusión	46
3.4. Consideraciones para el Despliegue	46
3.4.1. Importadores	47

3.4.2. Rutas a archivos de importación	47
3.4.3. Rutas a imágenes	47
3.5. Reflexión Final	48
4. Anexo: Especificación de la API y Cobertura	49
4.1. Cobertura de Pruebas Unitarias	49
4.1.1. Base de datos secundaria	52
4.2. Cumplimiento de REST	53
4.2.1. Recursos	54
4.2.2. Endpoints	55
4.2.3. Mecanismos de autenticación	58
4.2.4. Códigos de Error HTTP	60
Bibliografía	61

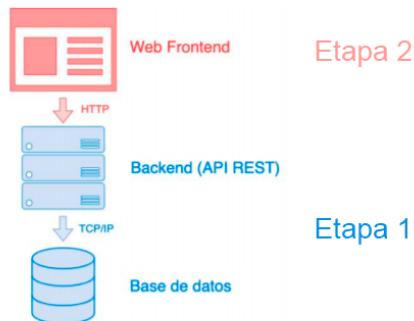
1. Introducción

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 la 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 esa etapa la persistencia de los datos de la aplicación utilizando una base de datos.

En la segunda etapa, se desarrolló el front-end de la aplicación utilizando el framework de desarrollo web Angular. Se creó una interfaz usuario amigable e intuitiva basada en “Material Design”, que consume la API expuesta por el back-end para proveer las funcionalidades al usuario final.

Para la ejecución de este proyecto se tuvo en cuenta permanentemente la necesidad de un diseño que facilitara la mantenibilidad y la interconexión entre sistemas. Por esta razón, se procuró seguir patrones de diseño que facilitaran estos objetivos. El producto final es una Web API con una serie de endpoints disponibles para ser consumidos por cualquier cliente, y un primer cliente web que provee una interfaz de usuario.



1.1. 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, precio, y un puntaje que se actualiza según las reseñas de los clientes. El sistema permite las siguientes operaciones sobre los hospedajes:

- a) Consultar hospedajes por punto turístico
- b) Consultar hospedajes por similitud de nombre
- c) Agregar un nuevo hospedaje *
- d) Borrar un hospedaje *
- e) 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. Importadores de Hospedajes: El sistema integra una función que le permite la importación dinámica de hospedajes (Y la potencial creación de puntos turísticos en caso de no existir el punto turístico asociado). Se provee una interfaz que un desarrollador externo puede implementar. Una vez compilada la herramienta, agregando el archivo “.dll” a una carpeta interna del proyecto dedicada a almacenar los mecanismos de importación, el sistema es capaz de utilizar la nueva funcionalidad consumida en tiempo de ejecución:

- a) Consultar importadores disponibles. Aquellos que se encuentran en la carpeta ImporterDLLs, del paquete WebApplication, y que satisfacen la interfaz requerida. Se retorna su nombre además de los parámetros esperados para su funcionamiento. *
- b) Consultar hospedajes por similitud de nombre
- c) Importar. Dado un mecanismo de importación y cargados sus respectivos parámetros, el sistema intenta importar los hospedajes, devolviendo un listado con aquellos hospedajes que se pudieron importar y aquellos que no, acompañados de un mensaje correspondiente justificando el error. *

6. 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 %.

En cuanto a los clientes jubilados, se creó una promoción según la cual dada una pareja de jubilados, uno de ellos paga el 30 % de la taza.

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 un reporte de reservas. Se define un punto turístico y un rango de fechas, y el sistema genera un listado de los hospedajes de dicho punto turístico, con la cantidad de reservas que se encuentran en ese periodo de tiempo, ordenados de mayor a menor cantidad de reservas.
 - d) Consultar todos los estados de reserva *
 - e) Crear una nueva reserva
 - f) Modificar el estado de una reserva y su descripción *
7. **Reseñas:** El sistema permite a los clientes el ingreso de una reseña sobre su experiencia con el hospedaje reservado. Dado un código de reserva que todavía no haya recibido una reseña, es posible ingresar un comentario además de una puntuación de 1 a 5. El sistema toma este ultimo para calcular el puntaje promedio del hospedaje, y muestra las reseñas a los nuevos clientes a la hora de navegar por los hospedajes disponibles. En concreto, el sistema permite las siguientes operaciones sobre las reseñas:
- a) Agregar una nueva reseña
 - b) Consultar una reseña a partir de un código de reserva
 - c) Consultar todas las reseñas de un hospedaje a partir de su código identificador
8. **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) Obtener un listado de todos los administradores del sistema *
- b) Crear un nuevo administrador en el sistema *
- c) Modificación del nombre, apellido y contraseña de un administrador a partir de su correo electrónico *
- d) Eliminar un administrador a partir de su número identificador. *

9. Sesiones: El sistema permite el manejo de sesiones de usuario. Se ahonda más en el detalle de funcionamiento de esta funcionalidad en la sección dedicada al mecanismo de autenticación.

Las acciones que pueden realizar los administradores en relación a las sesiones en el sistema son las siguientes:

- a) LogIn. Se proveen las credenciales de administrador y se genera una nueva sesión, o se retorna el identificador de sesión si ya se encontraba iniciada.
- b) LogOut. Se provee un identificador de la sesión y esta se da por finalizada.
- c) Consultar por existencia de una sesión

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: Para el manejo de las imágenes de puntos turísticos y hospedajes, actualmente se cuenta con un atributo único de tipo string, donde se define la ruta de la imagen. Aunque en la primera etapa se planificó poder manejar más de una imagen por entidad (concatenadas en la misma propiedad), en la etapa de desarrollo de front-end se optó por manejar una única imagen por cada hospedaje/punto turístico. En una versión futura se podría modificar el front-end para poder desplegar más de una imagen por cada entidad.

2. Descripción del Diseño

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.

2.1. Vista de Desarrollo

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

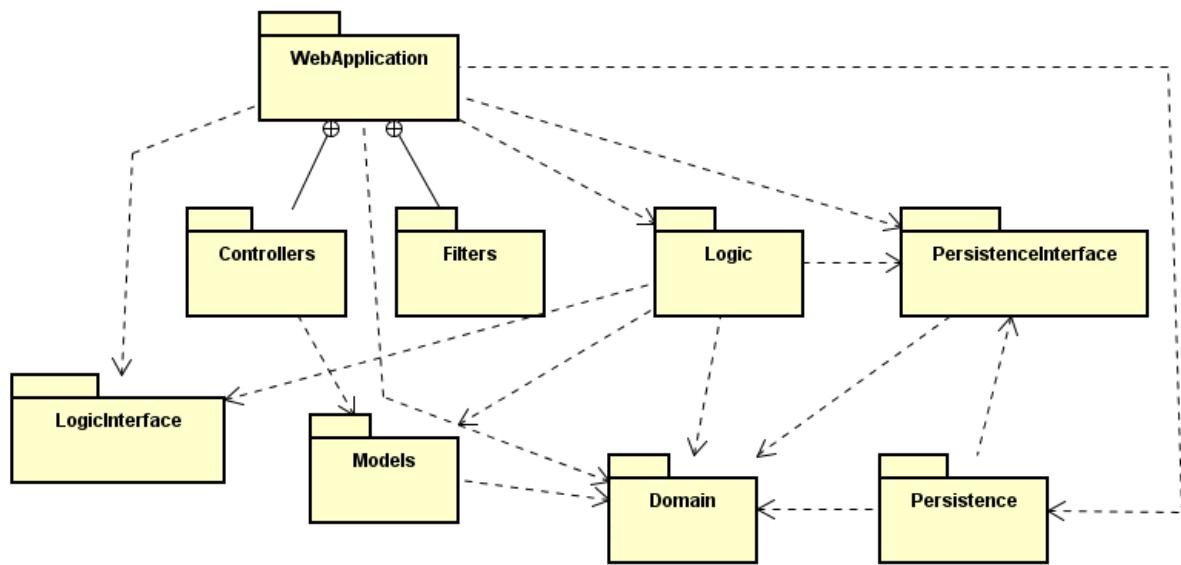


Figura 2.1: Diagrama de Paquetes - Mostrando los paquetes organizados por capas (layers) y sus dependencias.

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).

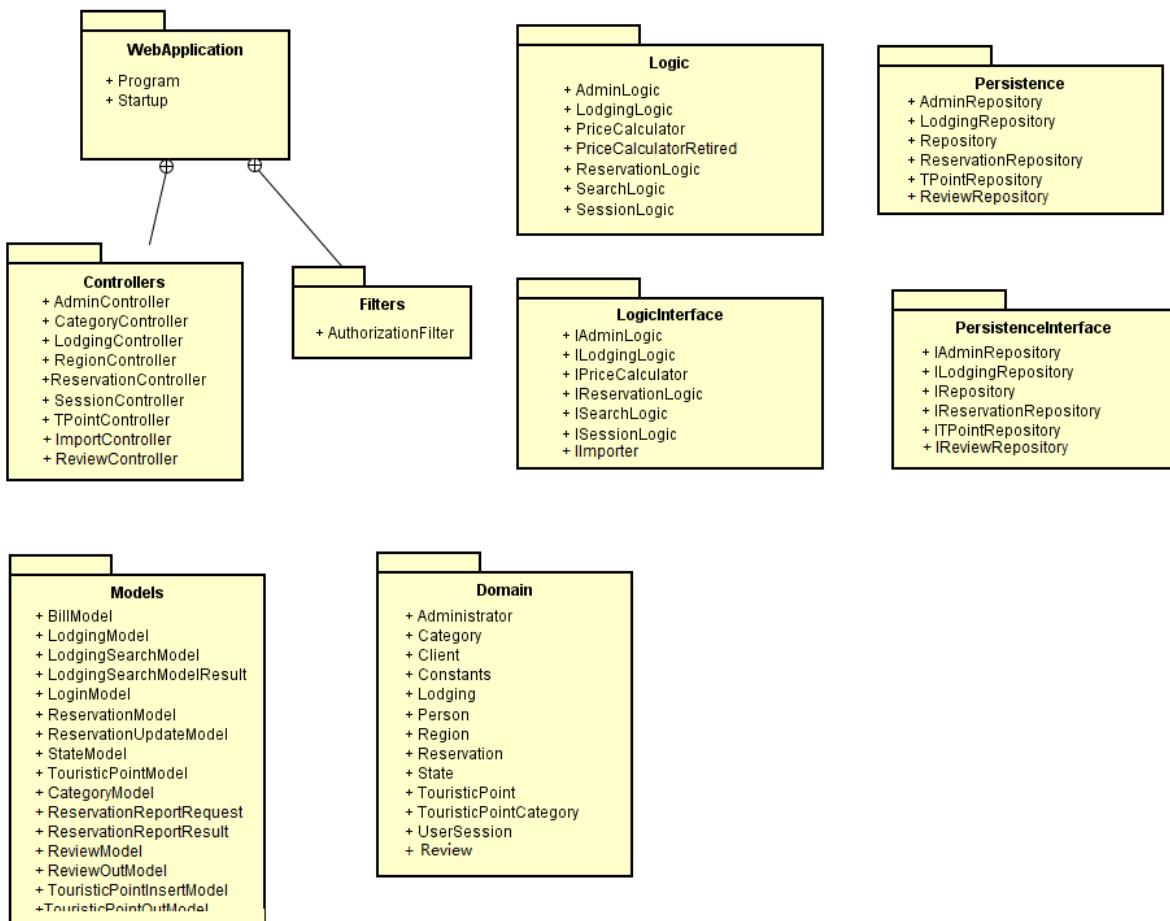


Figura 2.2: Diagrama de Paquetes - Utilizando conector de Nesting sin dependencias

2.1.1. Diagrama de Componentes

La arquitectura de nuestra aplicación 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 analizan en el diagrama de paquetes.

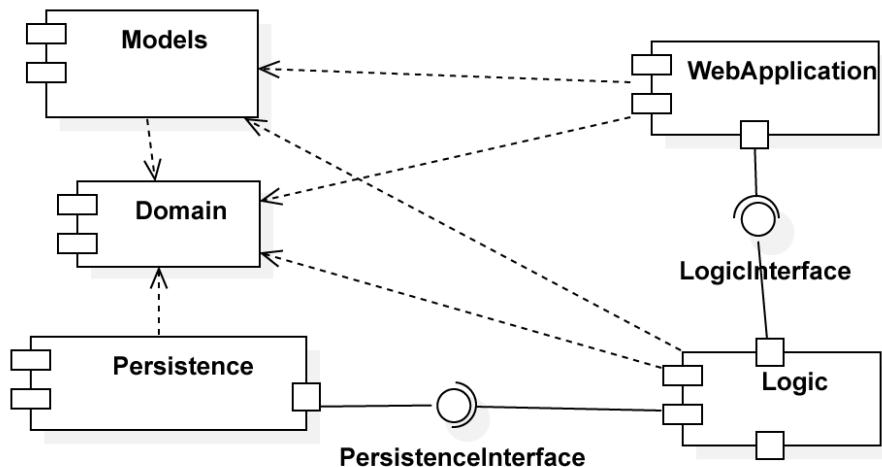
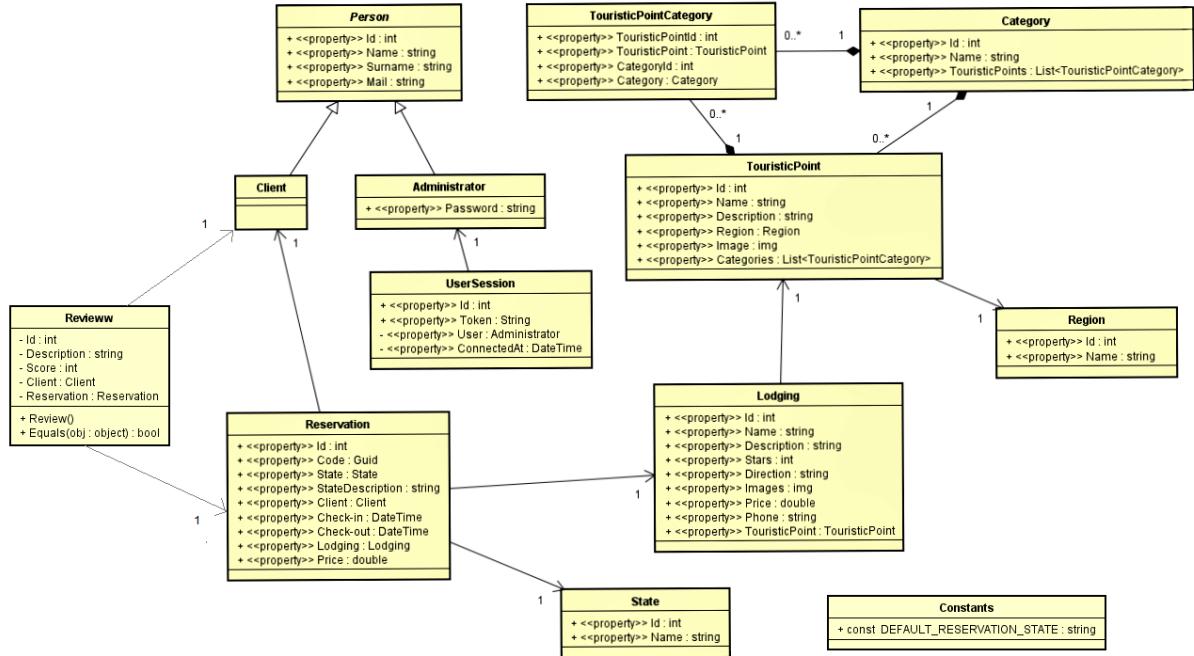


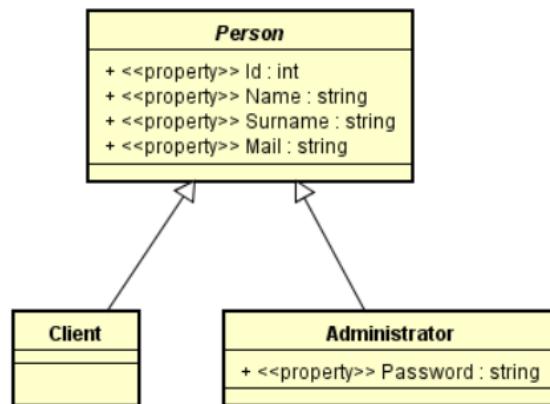
Figura 2.3: Diagrama de Componentes. Vista externa.

2.2. Vista Lógica

2.2.1. 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.





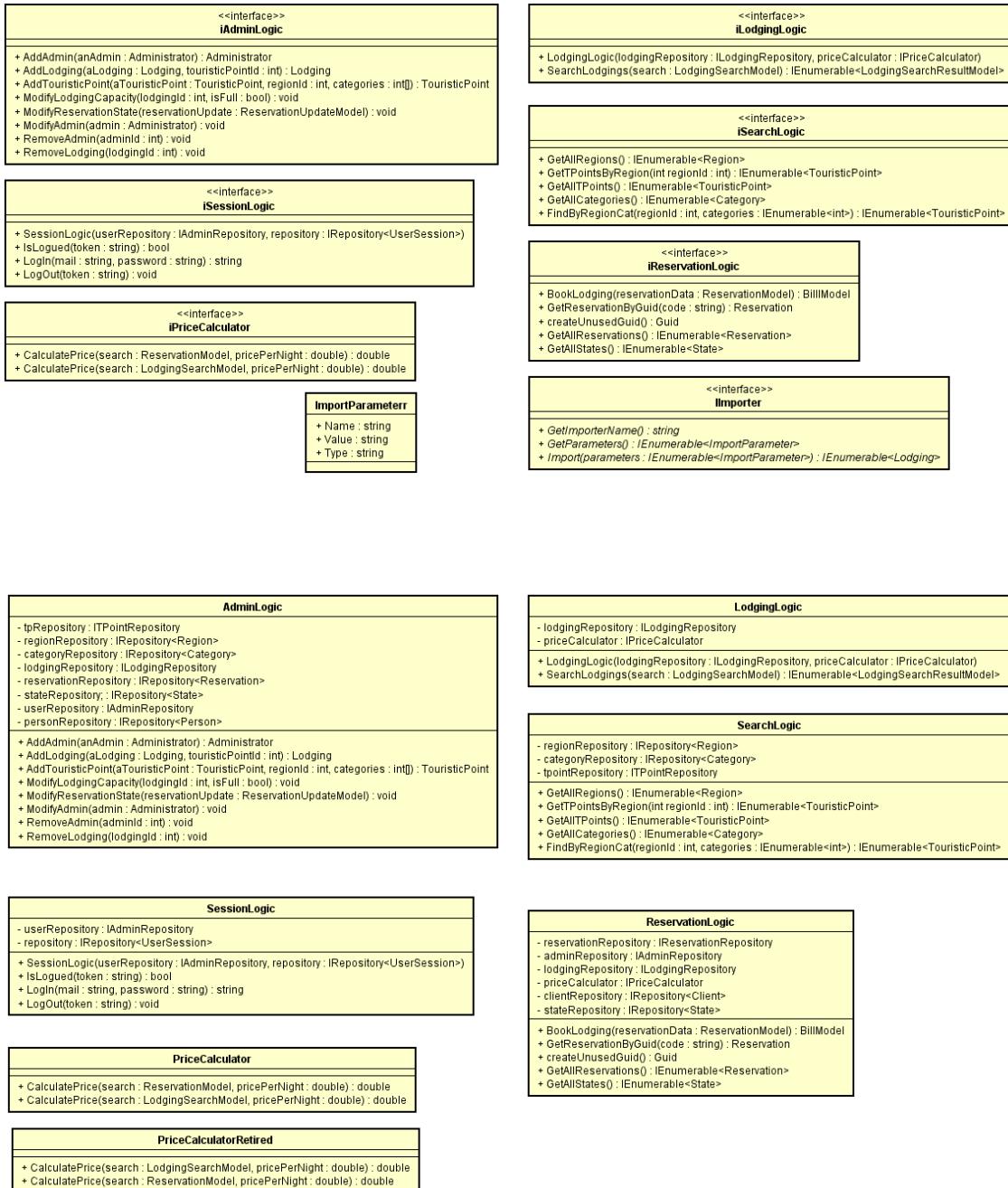
■ **Herencia:**

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 implemento esta herencia con el propósito de que si en un futuro se requiere agregar nuevos perfiles de persona o nuevas funcionalidades, las dos clases heredadas puedan ya ser extendidas, reutilizando código o sobreescibiendo la funcionalidad.

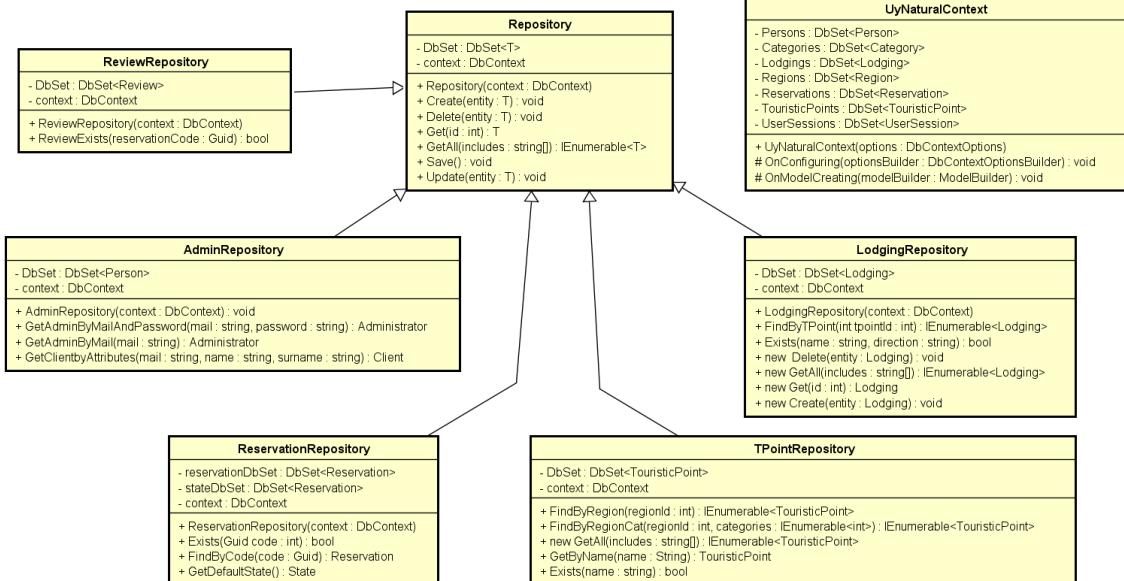
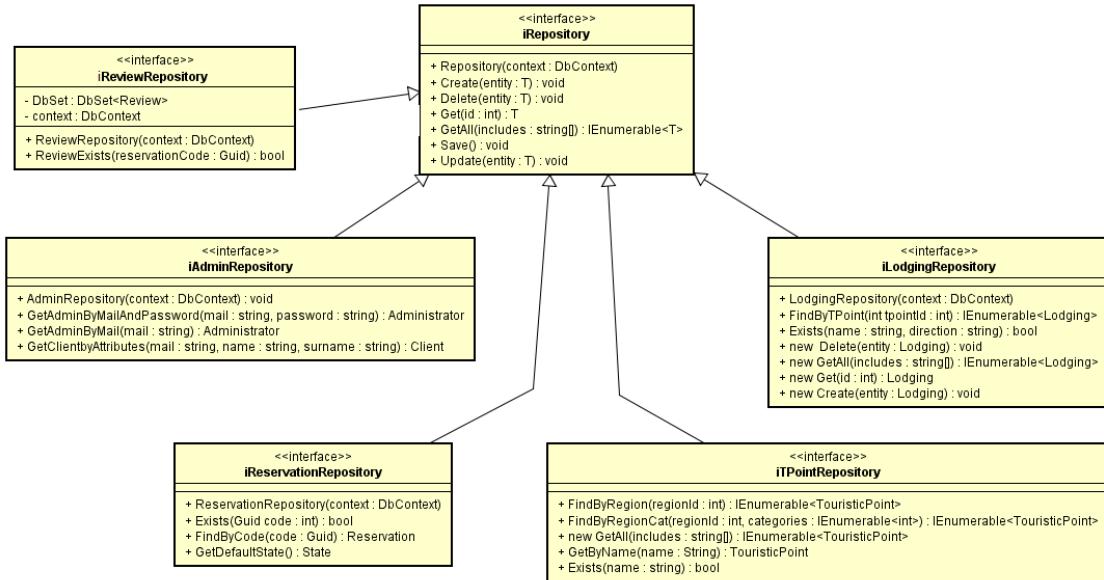
2.2.2. Paquetes: Logic y LogicInterface

Diagrama de Clases LogicInterface / Logic



2.2.3. 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[4], 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

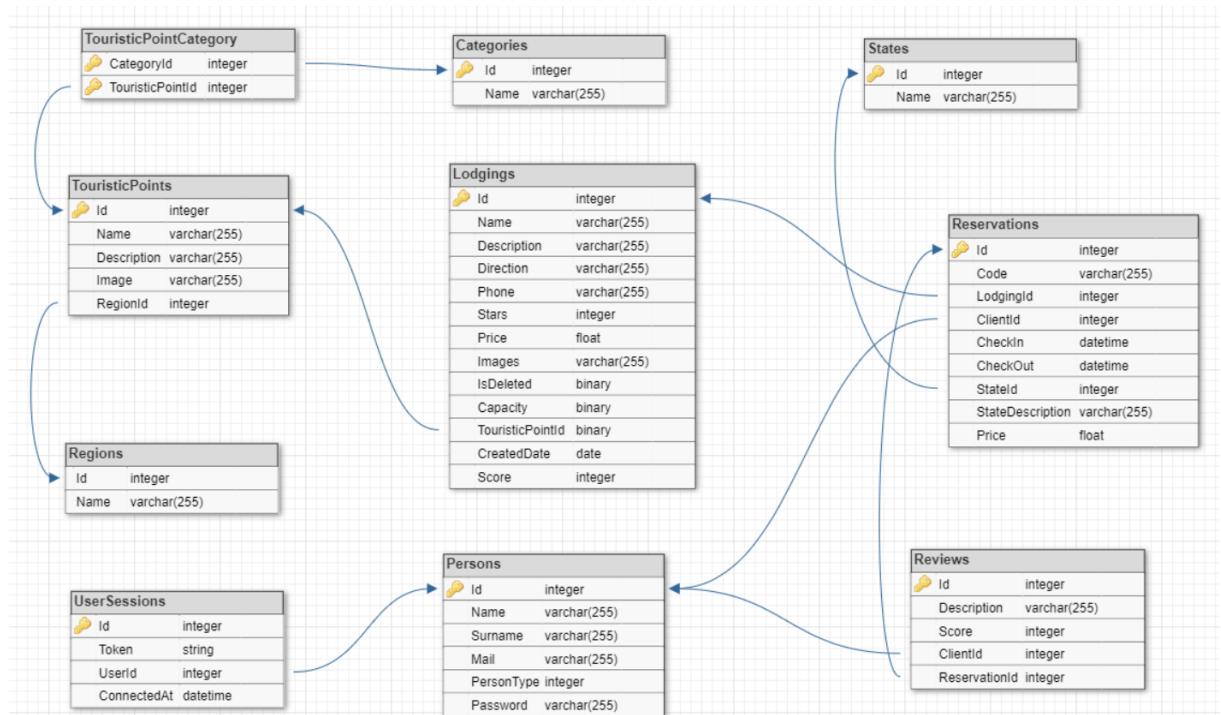
Método	Significado
.HasOne	Debe estar relacionado a un atributo del otro tipo
.HasMany	Puede estar relacionados con muchos atributos del otro tipo
.WithMany	Que a su vez pueden estar relacionados a muchos atributos del primer tipo
.HasKey	Indica las claves de la tabla
.HasForeignKey	Indica las claves 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 2.4: 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
```

2.2.4. Paquete: WebApplication

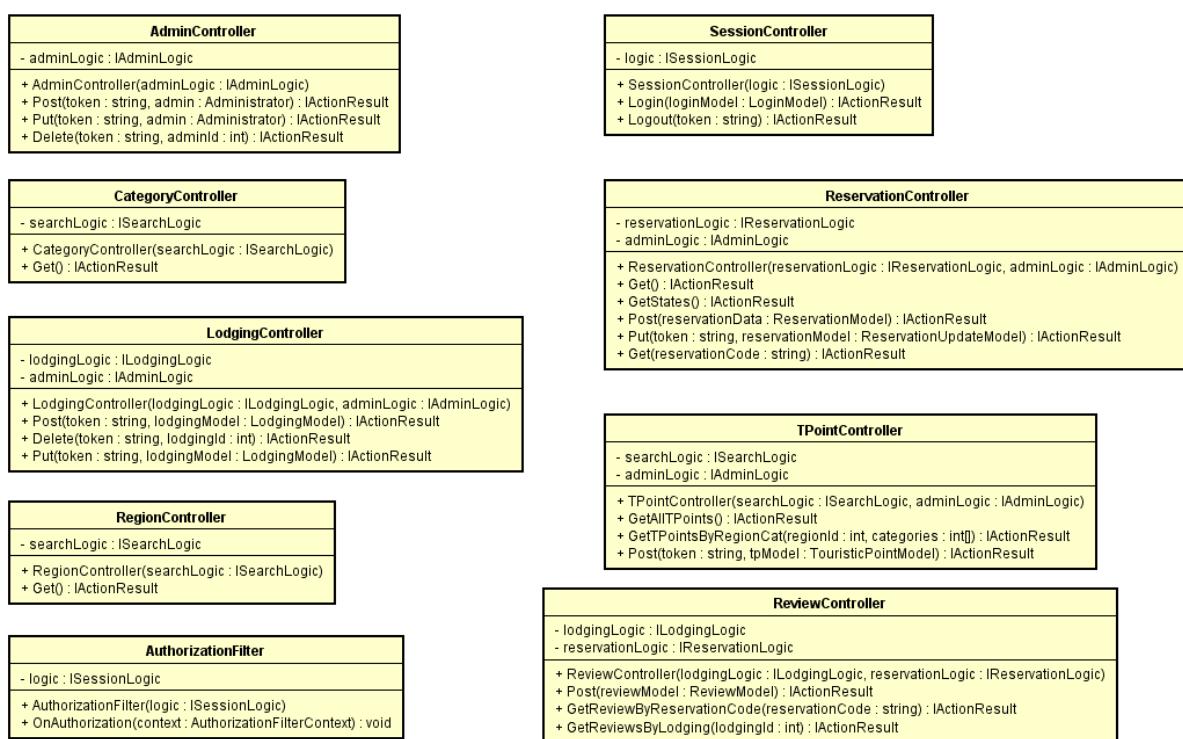


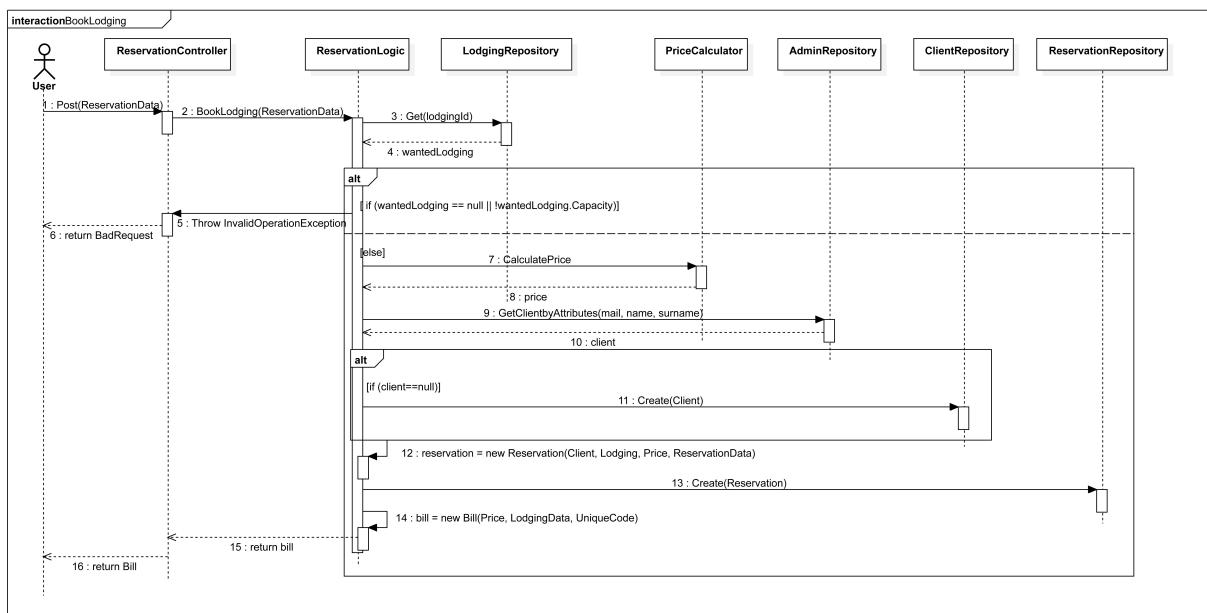
Figura 2.5: Diagrama de Clases - WebApplication

2.2.5. Funcionalidades Clave

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

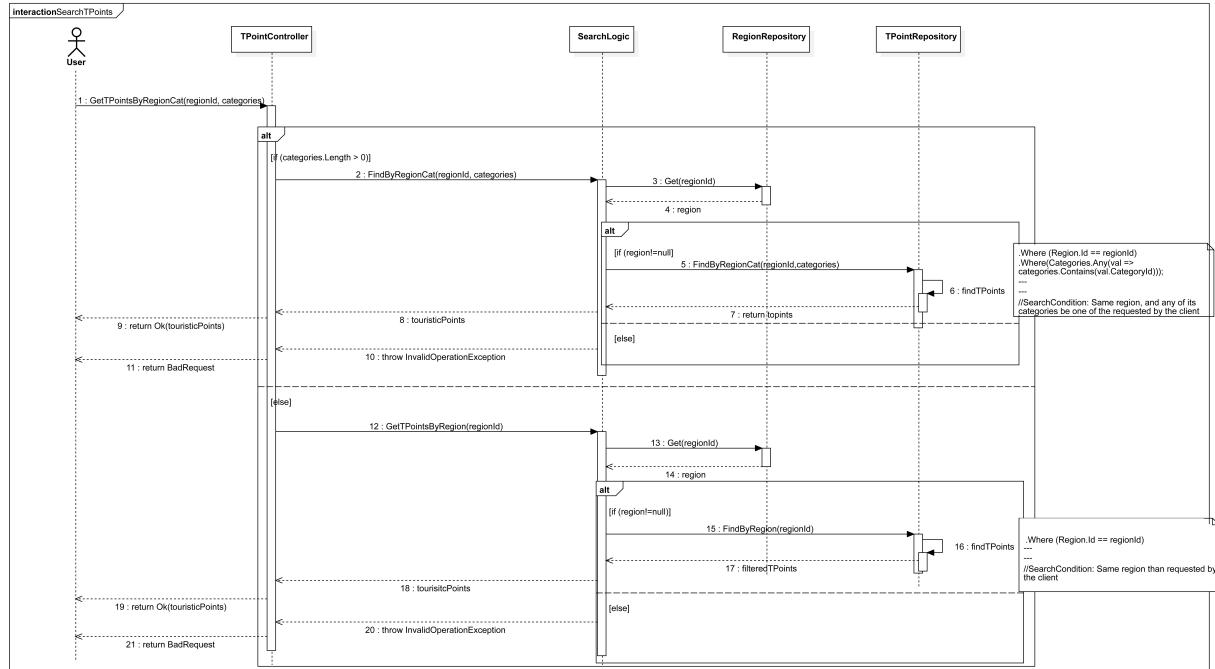
Funcionalidad de creación de una nueva reserva en el sistema.



Búsqueda de Puntos Turísticos por región y categorías (opcionalmente)

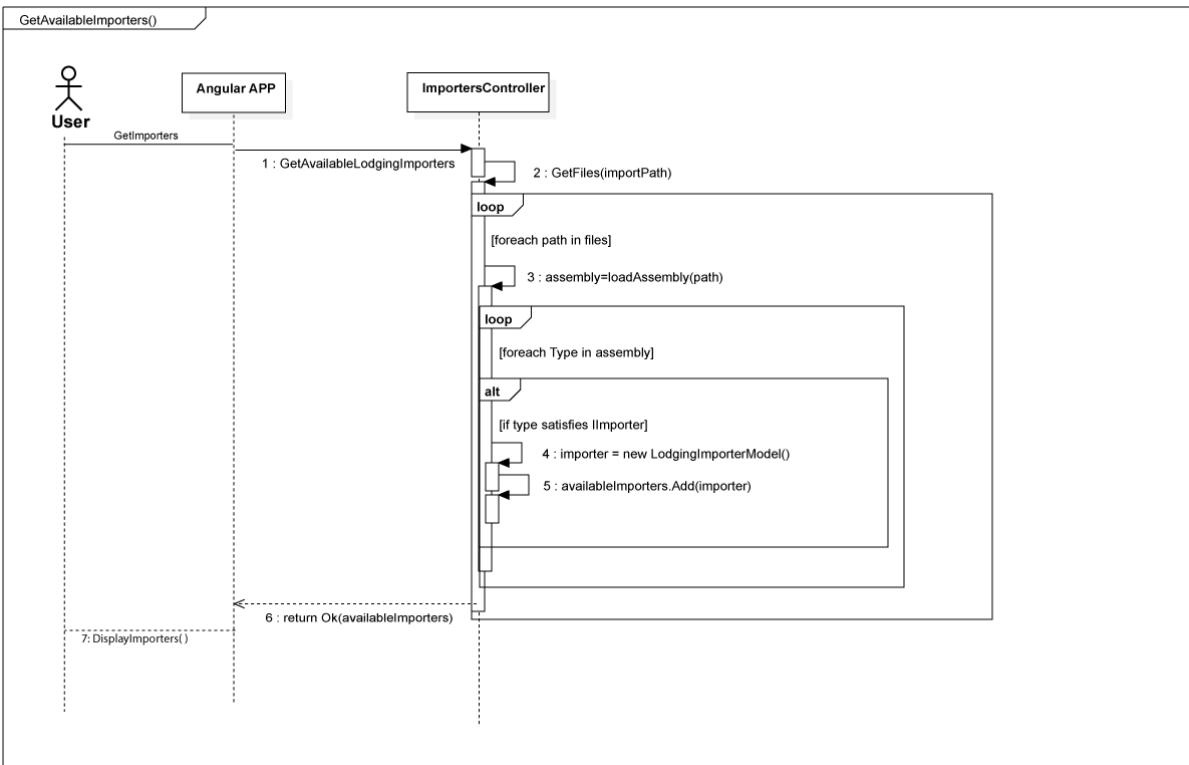
Funcionalidad de la API para la búsqueda de puntos turísticos.

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



Consultar importadores disponibles

Funcionalidad de consulta de importadores de hospedajes disponibles.



2.3. Patrones y Estrategias de Diseño

2.3.1. 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 2.6: 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.

2.3.2. 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<ClaseEspecífica>**.

```
namespace Logic
{
    public class SearchLogic : ISearchLogic
    {
        private IRepository<Region> regionRepository;
        private IRepository<Category> categoryRepository;
        private ITPointRepository tpointRepository;

        public SearchLogic(IRepository<Region> regionRepository,
                           IRepository<Category> categoryRepository,
                           ITPointRepository tpointRepository)
        {
            this.regionRepository = regionRepository;
            this.categoryRepository = categoryRepository;
            this.tpointRepository = tpointRepository;
        }
    }
}
```

Figura 2.7: 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
{
    public class TPointRepository : Repository<TouristicPoint>, ITPointRepository
    {
        private readonly DbSet<TouristicPoint> DbSet;
        private readonly DbContext context;

        public TPointRepository(DbContext context) : base(context)
        {
            this.DbSet = context.Set<TouristicPoint>();
            this.context = context;
        }
    }
}

```

Figura 2.8: 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 IEnumerable<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 2.9: Clase Startup.cs. Inyección de Dependencia

2.3.3. 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 en primera instancia entendimos más probablemente pudiera presentar alteraciones era el de cálculo de precios de las estadías, ya que está directamente vinculado a las reglas del negocio del momento.

Efectivamente fue una buena estrategia utilizar este patrón, ya que permitió recepcionar y gestionar fácilmente el cambio solicitado en la segunda etapa del proyecto, el cual modificaba las reglas de negocio para el cálculo de las cotizaciones de los hospedajes. Implementar este cambio fue tan sencillo como utilizar una nueva implementación de la interfaz IPriceCalculator, PriceCalculatorRetired.

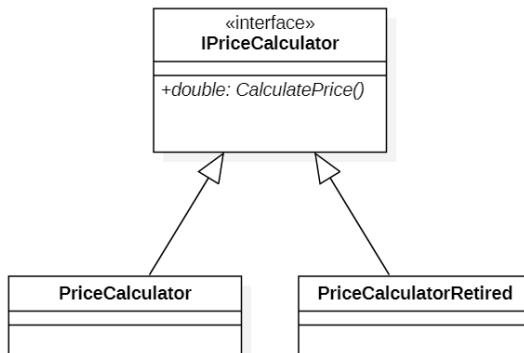


Figura 2.10: Interfaz IPriceCalculator con sus dos implementaciones

Las nuevas reglas de negocio vigentes, que se adhieren a las anteriores, es que los jubilados (nuevo tipo de huésped), tienen un descuento de un 30 % cuando van de a dos (Uno de los dos recibe el beneficio).

2.3.4. Reflection

Reflection es la habilidad de un programa de autoexaminarse, para cargar en tiempo de ejecución código pre-compilado, provisto por un tercero. En la segunda etapa del proyecto se implementó esta funcionalidad para permitir a los administradores utilizar mecanismos independientes de la aplicación para la carga por lotes de hospedajes (y sus respectivos puntos turísticos en caso de no existir).

Por ejemplo, desarrollamos a modo de ejemplo dos importadores que permiten la lectura y deserialización de archivos .Json y .Xml para cargar estas entidades.

Los detalles de implementación y funcionamiento se pueden ver en el diagrama de secuencia de la sección Funcionalidades clave.

Desde una perspectiva de diseño, esta técnica aporta un importante factor de extensibilidad a la solución, ya que permite aprovechar código de terceros o involucrar a desarrolladores externos, sin necesidad de modificar o re-compilar la aplicación original.

2.3.5. 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 conteniendo 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 2.11: 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 2.12: Administrator

2.3.6. SOLID

Los principios [7] S.O.L.I.D refieren a un conjunto de 5 técnicas de diseño orientadas al desarrollo de calidad en [5]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.

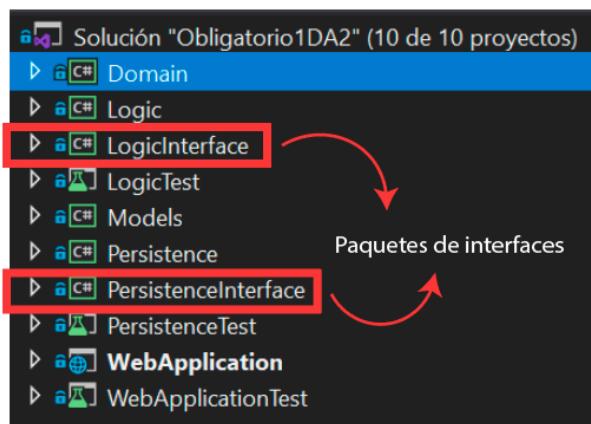
- **L - Principio de sustitución de Liskov:** No nos enfocamos en aplicar particularmente este principio.

- **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.

2.3.7. 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 intentó hacer el código lo más 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.

2.4. 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 2.13: Ejemplo: Captura de excepción - WebApplication

2.5. Front-End

Para el front-end desarrollamos en esta segunda etapa del proyecto, una aplicación sobre el framework de desarrollo web basado en typescript, Angular.

2.5.1. Estructura de la solución

Contamos con una solución distribuida en 7 módulos, un modulo principal “app-module” desde donde se levanta el programa. Dentro de este modulo se agregaron 6 módulos divididos según funcionalidades del sistema y también se agrego un modulo “share” para las cosas que comparten entre todos como la barra de navegación y el footer.

También se agrego el modulo “app-routing-module” para realizar la navegación entre los diferentes componentes del programa.

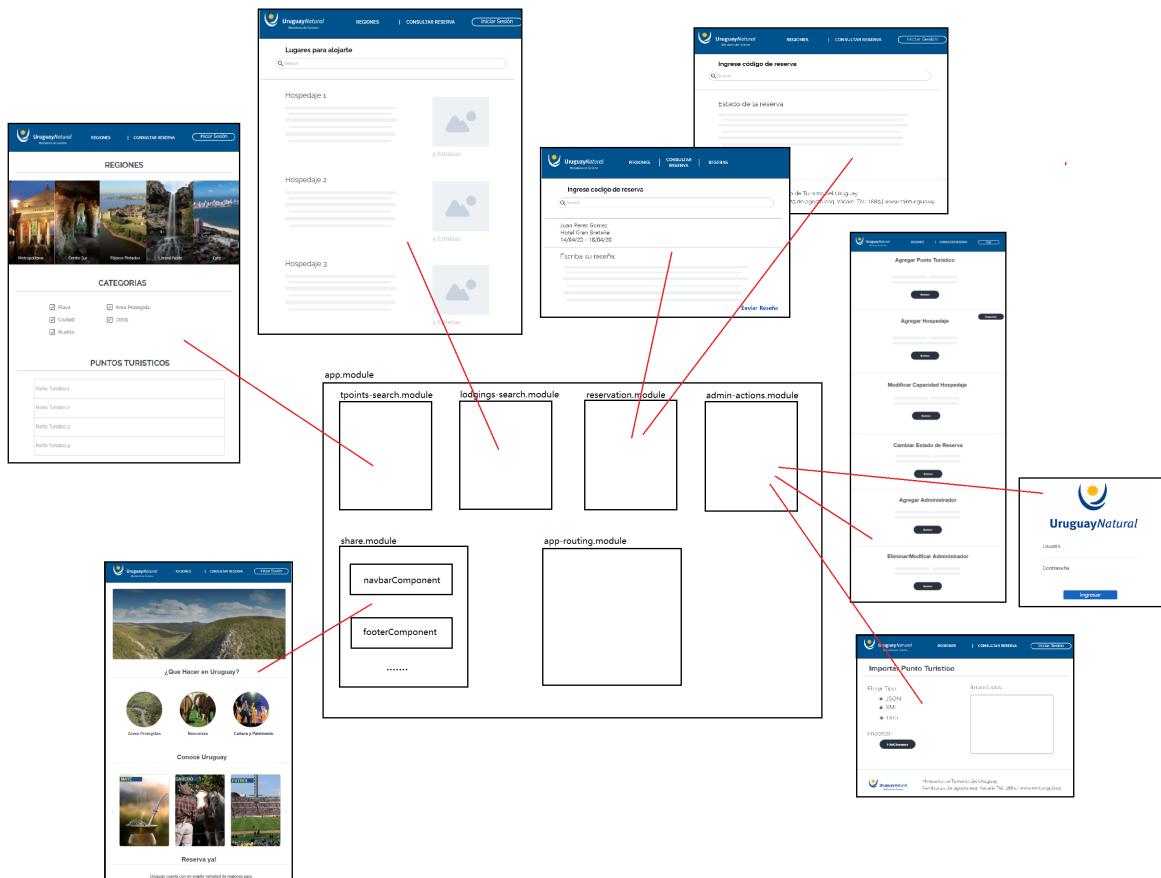


Figura 2.14: Estructura Front-end

2.5.2. Material Design

Para el diseño nos inspiramos en la pagina oficial de UruguayNatural.

Librerías: Utilizamos las librerías de AngularComponents y Bootstrap para los componentes de la pagina, generando así un aspecto moderno en la aplicación web.



Paleta de Colores: Se eligió una paleta de colores primarios para lograr una atractiva para el usuario. Para mensajes de alerta se eligieron los clásicos colores que generalmente se utilizan en los sistemas modernos.

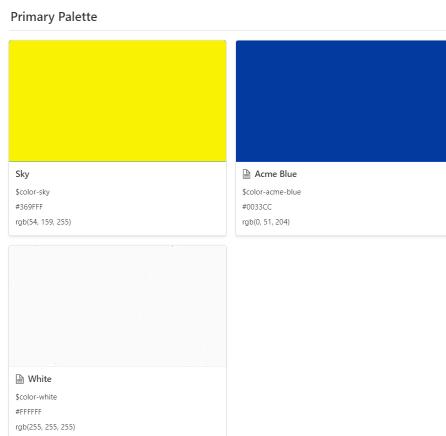


Figura 2.15: Paleta Primaria

Figura 2.16: Paleta Secundaria

Iconos: Se utilizaron los iconos de Angular Components para darle frescura a los iconos de la aplicación.

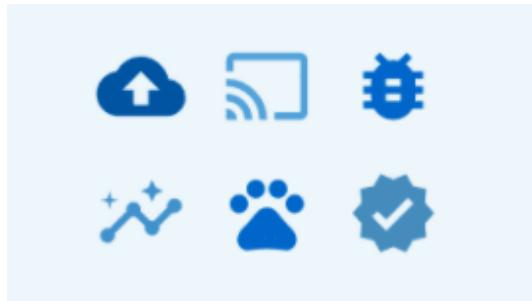


Figura 2.17: Pack Iconos - Angular Components

Fuentes: Se reemplazo la fuente de todos los textos del sistema a “Raleway” para darle un aspecto moderno a la aplicación.

A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! @ # \$ % ^ & * ()

Figura 2.18: Familia de fuentes - Raleway

3. Conclusiones

3.1. Mejoras del Diseño

A raíz de las devoluciones surgidas de la primera entrega, pudimos realizar algunas mejoras concretas en el diseño de nuestra aplicación. A continuación se listan las críticas constructivas recibidas, y la solución provista:

- Violación de REST. Un defecto de nuestra WebAPI en su primera versión, era que se habían implementado más de una operación con el mismo verbo, en endpoints comunes, cuando la recomendación de buenas prácticas para REST, propone utilizar un único verbo por endpoint. Además, se había utilizado la palabra “filter”, que es un verbo y por ende también escapa de las recomendaciones de REST.

Se corrigieron estos problemas unificando el punto de entrada de todos los métodos Get en uno común, y distinguiéndo el flujo requerido según los parámetros recibidos. Ejemplos de esto son el controlador **tpoints** y **lodgings**

The screenshot shows a portion of a C# code file. It includes several private methods: `GetAllTPoints()`, `GetTPointsByRegionCat([FromQuery] int regionId, [FromQuery] int[] categories)`, and `Get([FromQuery] int regionId, [FromQuery] int[] categories)`. The `Get` method is annotated with a red box and the label "Punto de Entrada común". Inside the `Get` method, there is a try-catch block. If `regionId != 0`, it calls `GetTPointsByRegionCat(regionId, categories)` and is labeled "Opción llamada 1". If `regionId == 0`, it calls `GetAllTPoints()` and is labeled "Opción llamada 2".

```
private IActionResult GetAllTPoints()...
///tpoints?regionId=3&categories=2&categories=3 --> filter by reg OR by reg & cats
private IActionResult GetTPointsByRegionCat([FromQuery] int regionId, [FromQuery] int[] categories)...

[HttpGet]
public IActionResult Get([FromQuery] int regionId, [FromQuery] int[] categories)
{
    try
    {
        if (regionId != 0)          Opción llamada 1
        {
            return GetTPointsByRegionCat(regionId, categories);
        }
        else
        {
            return GetAllTPoints(); Opción llamada 2
        }
    }
    catch (Exception)
    {
        return BadRequest("Error Desconocido");
    }
}
```

Figura 3.1: Punto de entrada unificado del verbo GET

Nota: Se puede ahondar en estas diferencias en el anexo al final del documento, donde se detallan los endpoints, y se detallan las entradas esperadas y las distintas posibles salidas.

- Violación de Clean Code con comentarios. Se hizo énfasis en la presencia de comentarios explicativos en muchas clases de la función, que violan la política de comentarios limpios de Clean Code. Según Robert C. Martin, el código de buena calidad debería ser autoexplicativo. Por esta razón redujimos la cantidad de comentarios dentro de los métodos, dejando únicamente aquellos que entendemos pertinentes y fundamentales para aportar entendimiento al código.
- Variables con nombres extraños: Otra violación de Clean Code de la primera entrega correspondía al uso de nombres de variables minimalistas y no nemotécnicos como “lst”, “l”, o “ret”. Intentamos realizar una sesión de refactoring en la que modificamos estas variables para lograr un código más limpio y entendible de forma sencilla. Sin embargo, también fue un desafío no introducir demasiadas palabras al código, transformándolo en uno más complejo de leer.
- Descuidos en el manejo de versiones. Se criticó el envío de archivos de **build** locales, que no deberían ser enviados al repositorio. En la nueva aplicación front-end, fuimos especialmente cuidadosos en configurar correctamente el archivo **.gitignore** para ignorar aquellos archivos autogenerados que no son de interés en el control de versiones.
- Aplicación de TDD. Aunque se nos hizo notar que existía en la primera entrega una rama llamada **increaseTestCoverage**, nos apegamos estrictamente a TDD durante todo el proceso, y esa ultima etapa de aumento de cobertura fue exclusivamente dedicada a comprobar los casos de Excepciones no controladas principalmente en los controladores. En la segunda entrega seguimos haciendo uso de TDD como metodología corriente en el desarrollo de las nuevas funcionalidades del back-end.

También hubo otros defectos que no pudimos abarcar en el plazo para la segunda entrega, y que se acumulan como deuda técnica para una posible futura versión del sistema.

- Choque de trenes. Este problema está relacionado con la concatenación de llamadas sin almacenar nunca los resultados intermedios en variables auxiliares. Esto complejiza las sesiones de debug ya que es más difícil identificar los estados intermedios y los posibles puntos de error. No pudimos dedicar recursos a mejorar este punto.
- Repetición de código en las pruebas. Al no utilizar Setups, se repitió mucho código a la hora de preparar los escenarios para las pruebas automáticas. Esto conlleva una lectura más dificultosa de la clase. En esta etapa no pudimos resumir los escenarios en escenarios genéricos.
- Más de un assert en las pruebas. Se nos hizo notar la presencia de más de una assert en ciertas pruebas. Sin embargo, decidimos no “corregir” este defecto de diseño ya que entendemos que en los casos en que utilizamos más de un assert, es porque es la única manera de comprobar la integridad del objeto

que se quiere comprobar correcto. Por ejemplo, en ciertos casos es de interés corroborar que el resultado de una búsqueda del repositorio es una lista de un elemento, y que el elemento es el que se pretende. Según nuestra forma de verlo, no valdría la pena realizar más de una prueba para verificar ambas condiciones.

3.2. 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 implica dejar pendiente cierta deuda técnica que se detalla a continuación (para tomar en consideración en futuras versiones de la aplicación...):

- **Imágenes de Hospedajes.** El manejo de las imágenes de los hospedajes ya era un pendiente en la deuda técnica de la primera entrega, ya que se contaba con un único atributo de tipo string en las entidades **lodging** y **tpoint**. Se planificaba que al implementar el front-end, estas se enviaran concatenadas por algún carácter específico para poder diferenciarlas y así cumplir con el requerimiento de manejar multiples imágenes.

En la segunda etapa, la implementación del front-end nos exigió mucho tiempo y una gran curva de aprendizaje que tuvimos que transitar rápidamente. Por esta razón, desde un principio tomamos la decisión consciente de dejar de lado la gestión de varias imágenes por entidad, y en su lugar asumimos que siempre contaremos con una única imagen para cada una. Estas se despliegan de forma dinámica en la aplicación Angular.

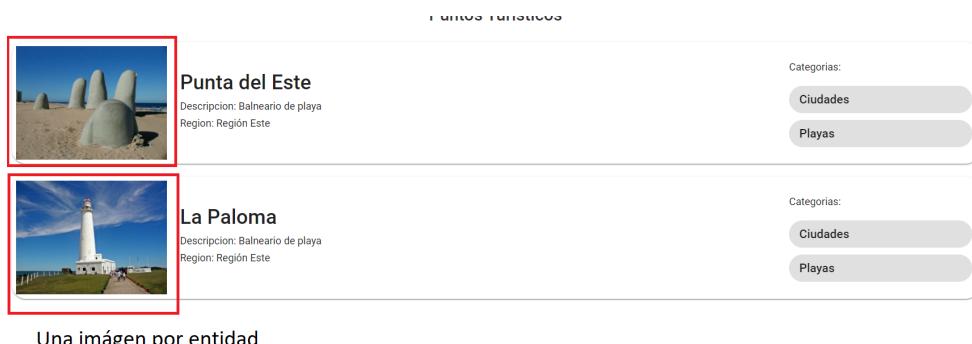
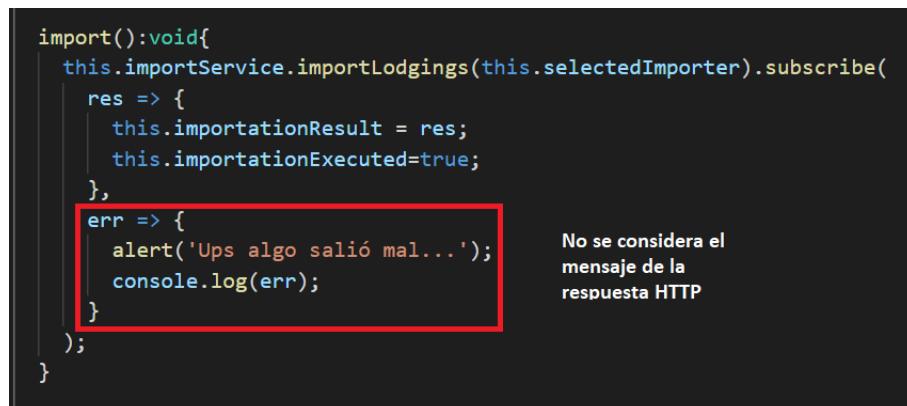


Figura 3.2: “Images” de tipo string

- **Errores del back-end** Una de los detalles de calidad que habían quedado pendientes en la primera entrega era el manejo de excepciones de cara al cliente. Se implementaron en una primera instancia algunas respuestas utilizando los códigos HTTP: 400 (de BadRequest), el 404 (de Not Found) y el 401 (de Unauthorized)

En esta etapa tuvimos el desafío de la interconexión de sistemas. El front-end desarrollado se comunica mediante llamadas HTTP con la WebAPI y esta le retorna códigos de éxito o de error. Sin embargo, no tuvimos tiempo como para hacer un manejo consistente de los errores retornados por la API, y de aprovechar el potencial de los mensajes que acompañan a los códigos de estado. En su lugar, muchas veces simplemente consideramos un error inesperado y alertamos al usuario con un aviso amigable “Ups.. Algo salió mal”.

En una próxima versión de la aplicación se debería priorizar la gestión de los errores para aprovechar al máximo las funcionalidades desarrolladas previamente en el back-end, y aportar luz en los caso de fallas de cara al usuario final.



```
import():void{
  this.importService.importLodgings(this.selectedImporter).subscribe(
    res => {
      this.importationResult = res;
      this.importationExecuted=true;
    },
    err => {
      alert('Ups algo salió mal...');
      console.log(err);
    }
  );
}
```

No se considera el
mensaje de la
respuesta HTTP

Figura 3.3: Ejemplo de no aprovechamiento del código de error

- Inconsistencia en los Datos. A lo largo del desarrollo del back-end fuimos creando modelos similares para algunas entidades del dominio. Nos dimos cuenta que esto generó complicaciones y malentendidos a la hora de llevar estos modelos al front-end. Se pudo haber realizado un solo modelo para cada entidad y obtener los datos necesarios para esa llamada a la API.

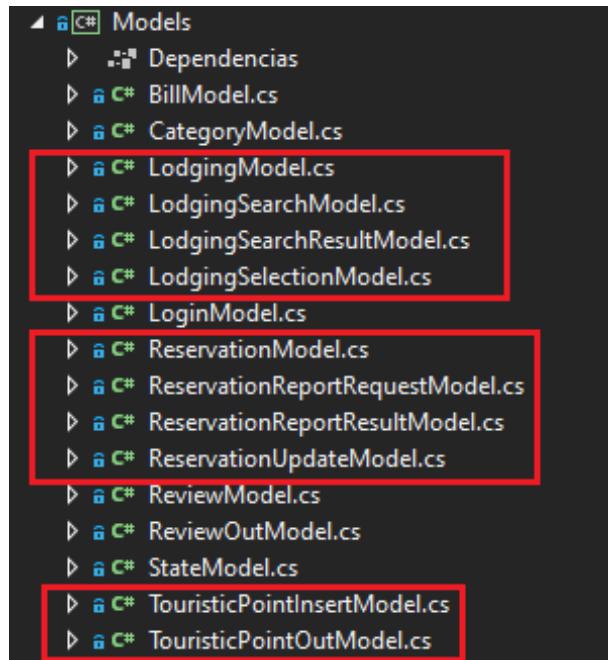


Figura 3.4: Modelos de la API

3.3. Métricas

Las métricas son herramientas que usaremos para establecer conclusiones cuantitativas de nuestro diseño. En esta caso discutiremos la calidad del diseño en base a las métricas de diseño de:

- **Cohesión Relacional (H)**: Mide que tan cohesivo es un paquete. Lo hace a través de medir la relación entre las clases del paquete.
 - **Inestabilidad (I)**: Mide de que tan complicado es cambiar un paquete.
 - **Abstracción (A)**: Mide que tan abstracto es un paquete.
 - **Distancia (D)**: nos permite saber que tan lejos esta un paquete de la secuencia principal. En esta secuencia principal se encuentran los paquetes que no son ni muy abstractos, ni muy estables.

3.3.1. Cohesión Relacional (H)

Esta se realiza a través de medir la relación entre las clases del paquete.

Tiene en cuenta dos valores:

- R = cantidad de relaciones entre clases internas al paquete
 - N = cantidad de clases e interfaces dentro del paquete

Calculo: $H = (R + 1) / N$

1. WebApplication: $H = (0 + 1) / 9 = 0.11$
2. Logic: $H = (0 + 1) / 7 = 0.14$
3. LogicInterface: $H = (0 + 1) / 8 = 0.13$
4. Persistence: $H = (5 + 1) / 7 = 0.85$
5. PersistenceInterface: $H = (5 + 1) / 6 = 1.0$
6. Domain: $H = (13 + 1) / 13 = 1.1$

3.3.2. Inestabilidad (I)

Esto lo vemos analizando la cantidad de dependencias entrantes vs la cantidad de dependencias salientes. Los paquetes con más dependencias entrantes deberán exhibir un alto grado de estabilidad.

Tiene en cuenta dos valores:

- C_a = Acoplamiento aferente (dependencias entrantes). Número de clases o interfaces fuera del paquete que dependen de clases o interfaces dentro del paquete.
- C_e = Acoplamiento eferente (dependencias salientes). Número de clases o interfaces fuera del paquete de las cuales clases o interfaces dentro del paquete dependen.

$$\text{Calculo: } I = C_e / (C_e + C_a)$$

1. WebApplication: $I = 5 / (5 + 0) = 1$
2. Logic: $I = 4 / (4 + 1) = 0.8$
3. LogicInterface: $I = 0 / (0 + 2) = 0$
4. Persistence: $I = 2 / (2 + 1) = 0.67$
5. PersistenceInterface: $I = 1 / (1 + 3) = 0.25$
6. Domain: $I = 0 / (0 + 5) = 0$

Una inestabilidad cercana a cero indica un paquete estable, mientras que un valor cercano a uno indica un paquete inestable.

3.3.3. Abstracción (A)

La abstracción de un paquete se mide calculando el ratio entre el numero de clase abstractas e interfaces del paquete y el numero de clases concretas en el paquete.

Tiene en cuenta dos valores:

- N_a = Cantidad de clases abstractas e interfaces en el paquete.
- N_c = Cantidad de clases abstractas e interfaces + clases concretas en el paquete.

Calculo: $A = N_a / N_c$

1. WebApplication: $A = 0 / 9 = 0$
2. Logic: $A = 0 / 7 = 0$
3. LogicInterface: $A = 7 / 8 = 0.88$
4. Persistence: $A = 6 / 6 = 1.0$
5. PersistenceInterface: $A = 6 / 6 = 1$
6. Domain: $A = 1 / 13 = 0.1$

Una abstracción cercana a cero indica un paquete concreto, mientras que un valor cercano a uno indica un paquete abstracto.

3.3.4. Distancia (D)

Finalmente, la distancia de un paquete se mide en base a las dos métricas anteriores Abstracción(A) e Inestabilidad(I) del paquete.

Calculo Normalizado: $D' = | A + I - 1 |$

1. WebApplication: $D' = | 0 + 1 - 1 | = 0$
2. Logic: $D' = | 0 + 0.8 - 1 | = 0.2$
3. LogicInterface: $D' = | 0.88 + 0 - 1 | = 0.12$
4. Persistence: $D' = | 1 + 0.67 - 1 | = 0.67$
5. PersistenceInterface: $D' = | 1 + 0.25 - 1 | = 0.25$
6. Domain: $D' = | 0.1 + 0 - 1 | = 0.9$

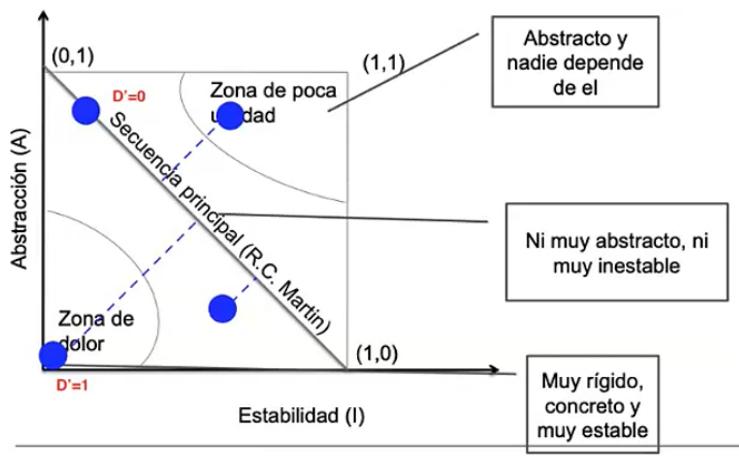


Figura 3.5: Interpretación cálculo de distancia

La utilizamos como una forma de validar el cumplimiento de **SAP (Principio de Abstracciones estables)**: I debe aumentar cuando A disminuye. Es decir, los paquetes concretos deben ser inestables mientras que los paquetes abstractos deben ser estables.

Estos valores obtenidos de distancia se pueden interpretar de la siguiente forma:

- **Valores cercanos a 0** Indican que el paquete está directamente sobre la secuencia principal.
- **Valores cercanos a 1** Indican que el paquete está muy lejos de la secuencia principal. En este caso el paquete se puede encontrar en:
 - **Zona de Dolor** (esquina abajo-izquierda) indican que el paquete es concreto y estable (responsable)/ Estos paquetes no son buenos porque no son extensibles y si cambian impactan en otros.
 - **Zona de poca utilidad** (esquina arriba-derecha) indican que el paquete es abstracto e inestable. El paquete es extensible pero tiene pocos paquetes que dependen de él.

En nuestra solución contamos tanto con paquetes que se encuentran sobre la secuencia principal como otros que se encuentran muy alejados de ella. Quizás en base a esta métrica se podrían mejorar el diseño de los paquetes que se encuentran alejados de la secuencia principal para así tener una solución reusable a futuro.

3.3.5. Conclusión

En base a las métricas podemos concluir que:

1. Todos los paquetes **cumple con REP (Principio de Equivalencia Reuso/- Liberación)**, en nuestra solución no pasa que para poder reusar cosas del paquete X tenga que depender de varios ensamblados.
2. Nuestra solución **cumple con SDP (Principio de dependencias estableas)**, ya nuestros paquetes dependen solamente de paquetes que son más estables que ellos.

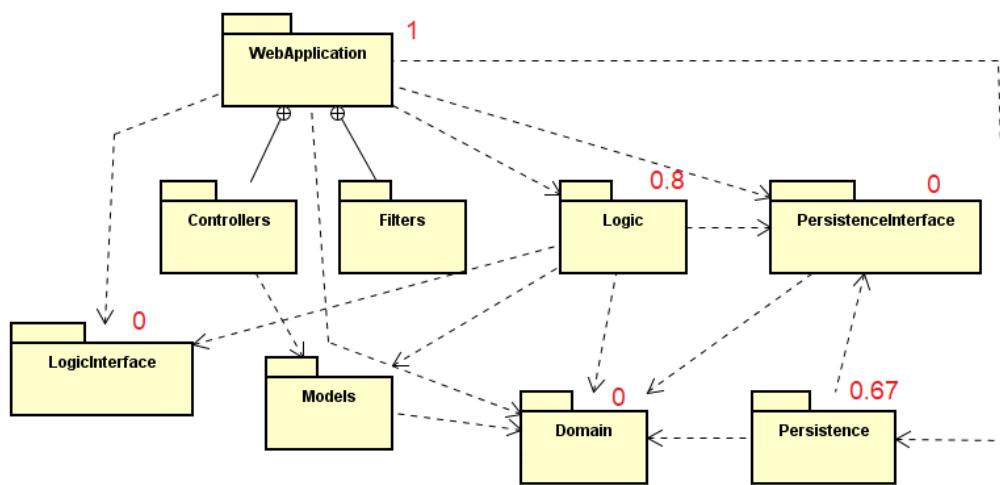


Figura 3.6: Cumplimiento SDP

3. CCP (Principio de Clausura Común) y CRP (Principio de Reuso Común): Los paquetes de nuestra solución cumplen con CCP ya que las clases perteneciente a un mismo paquete, cambian por el mismo tipo de cambio. Si una cambio afecta a un paquete, afecta a todas las clases del mismo y a ningún otro paquete. Esto está vinculado con la separación lógica de los componentes (Ver diagrama de componentes), y el principio de alta cohesión, presente en todo el proyecto.
Además, por encontrarse en una etapa de desarrollo el sistema está más orientado a favorecer el mantenimiento constante y la introducción de cambios.

Por estar razón, podríamos afirmar que no favorece CRP, que se suele impulsar en las etapas en que es relevante la reutilización de los componentes.

3.4. Consideraciones para el Despliegue

A continuación se presentan algunas consideraciones a tener en cuenta a la hora de realizar el despliegue en producción de la aplicación.

3.4.1. Importadores

Para poder utilizar la nueva funcionalidad de importadores de hospedajes, es necesario contar con la carpeta **ImporterDLLs**, dentro del entorno de ejecución donde se ejecuta la WebAPI. Es importante respetar esta consideración ya que la aplicación está compilada considerando que dentro del paquete WebApplication se encuentra dicha carpeta, con una ruta relativa estática.

Allí dentro se deben disponer los importadores compilados, con terminación **.dll**.

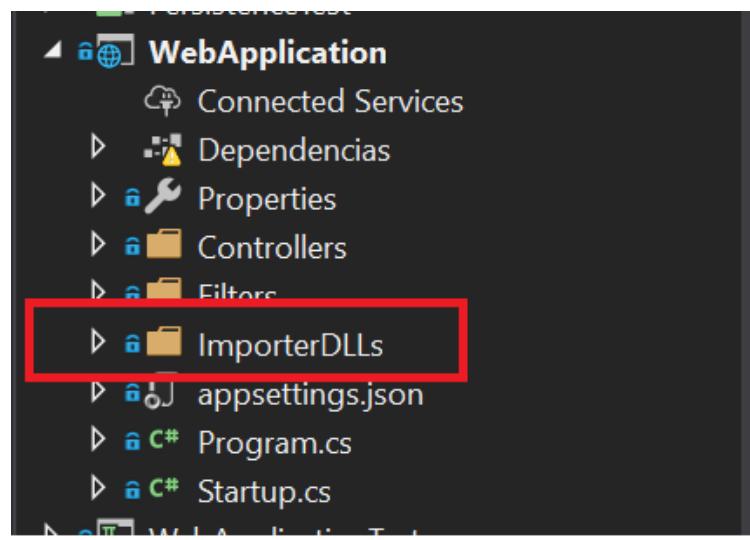


Figura 3.7: Carpeta del proyecto

3.4.2. Rutas a archivos de importación

Algunos importadores pueden solicitar como parámetros de entrada rutas a ubicación de archivos. Es importante destacar que siempre se consideran las rutas disponibles en la máquina local donde se despliega el componente del **back-end** (NO el front-end).

En otra instancia se podría implementar una transferencia de archivos para rea- lizar una comunicación más real.

3.4.3. Rutas a imágenes

En cuanto a las imágenes, es importante destacar que solo se consideran aquellas imágenes disponibles en el entorno de ejecución del front-end. Esto significa que para dar de alta una nueva entidad con una imagen, se debería proveer una ruta interna del proyecto (aplicación angular) que se sepa existente, o de lo contrario proveer un enlace a una imagen pública de internet, a la que se pueda acceder para desplegar.

Para la primera alternativa, se recomienda utilizar la carpeta del proyecto **images**, que fue declarada como asset en los archivos de configuración de angular, para que se levante en producción al momento de ejecutar la aplicación.

3.5. Reflexión Final

Ya habiendo culminado el proyecto **UYNatural**, procedemos a dejar unas reflexiones finales de lo que fue el obligatorio. En poco menos de tres meses, desarrollamos una WebAPI utilizando ASP.NET, un framework con el que ninguno de los integrantes del equipo tenía experiencia previa, y luego en un segundo **sprint**, una aplicación web front-end dinámica, conectada al back-end mediante llamadas HTTP. Para lograr esto tuvimos que familiarizarnos con una serie de tecnologías y paradigmas novedosos, como REST para la webAPI, y el framework Angular para el front-end. En cuestión de meses transcurrimos por una serie de aprendizajes que nos dejan orgullosos del trabajo realizado.

Aunque siempre hay puntos de mejora (como los que detallamos en la sección de Deuda Técnica), quedamos muy conformes con nuestras decisiones de diseño, que ya consideramos lo suficientemente bien fundadas como para elevar la calidad de nuestros productos. En esta segunda etapa, nos motivó el desafío de lograr una buena experiencia usuaria, inspirada en las interfaces que solemos usar en nuestras vidas diarias. Además, en lo que corresponde al diseño, la aplicación de patrones y principios en momentos oportunos nos permitió introducir cambios luego sin mayores dificultades. Esto 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. También tuvimos la experiencia y el desafío de aplicar metodologías de desarrollo profesionales en el proceso de trabajo, como TDD, o **GitFlow** como proceso de SCM para el manejo consistente del repositorio y el control de versiones. Este último punto se vio particularmente afectado en la segunda entrega, cuando tuvimos que manejar las versiones de dos aplicaciones en un único repositorio, y eso complicó el uso de **ramas** y **features**. Siguiendo por la misma línea, tomamos como premisa el desarrollo completo en **idioma Inglés**, incluyendo comentarios y commits.

Por último, una vez más evaluamos positivo nuestro desempeño como equipo de trabajo, ya que pudimos distribuir las responsabilidades de forma equitativa y acompañarnos en el enfrentamiento con nuevas tecnologías.

4. Anexo: Especificación de la API y Cobertura

4.1. 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 mas difícil de detectar en fases más avanzadas del proyecto.

En total se realizaron **147 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 (60)	421 ms		
LogicTest (60)	421 ms		
AdminLogicTest (18)	282 ms		
LodgingLogicTest (6)	53 ms		
PriceCalculatorRetiredTest (8)	5 ms		
PriceCalculatorTest (4)	1 ms		
ReservationLogicTest (11)	50 ms		
SearchLogicTest (7)	11 ms		
SessionLogicTest (6)	19 ms		
PersistenceTest (35)	1,9 s		
PersistenceTest (35)	1,9 s		
AdminRepositoryTest (6)	1,2 s		
LodgingRepositoryTest (9)	227 ms		
RegionRepositoryTest (1)	6 ms		
RepositoryTest (4)	95 ms		
ReservationRepositoryTest (7)	164 ms		
ReviewRepositoryTest (1)	48 ms		
TPointRepositoryTest (7)	181 ms		
WebApplicationTest (52)	316 ms		
WebApplicationTest (52)	316 ms		
AdminControllerTest (7)	218 ms		
AuthorizationFilterTest (3)	24 ms		
CategoryControllerTest (2)	7 ms		
ImportControllerTest (2)	8 ms		
LodgingControllerTest (9)	16 ms		
RegionControllerTest (1)	1 ms		
ReservationControllerTest (11)	22 ms		
ReviewControllerTest (5)	8 ms		
SessionControllerTest (5)	4 ms		
TPointControllerTest (7)	8 ms		

Figura 4.1: 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 obtuvieron los siguientes porcentajes de cobertura:

- 88 % sobre el paquete Logic.
- 98 % sobre el paquete Persistence.
- 82 % sobre el paquete Controllers.
- 100 % sobre el paquete Filters.

Jerarquía ▲	No cubiertos (bloques)	No cubiertos (% de bloques)	Cubiertos (bloques)	Cubiertos (% de bloques)
• felip_FELIPE-PC 2020-11-25 18_07_43.coverage	1292	16,27 %	6650	83,73 %
• domain.dll	85	34,41 %	162	65,59 %
▷ { } Domain	85	34,41 %	162	65,59 %
• logic.dll	63	11,86 %	468	88,14 %
▷ { } Logic	63	11,86 %	468	88,14 %
▷ logicinterface.dll	3	50,00 %	3	50,00 %
▷ logictest.dll	50	2,20 %	2218	97,80 %
▷ models.dll	38	21,84 %	136	78,16 %
• persistence.dll	896	51,35 %	849	48,65 %
▷ { } Persistence	12	1,39 %	849	98,61 %
▷ { } Persistence.Migrations	884	100,00 %	0	0,00 %
▷ persistencetest.dll	9	0,68 %	1312	99,32 %
• webapplication.dll	148	32,10 %	313	67,90 %
▷ { } Filters	0	0,00 %	23	100,00 %
▷ { } WebApplication	83	100,00 %	0	0,00 %
▷ { } WebApplication.Controllers	65	18,31 %	290	81,69 %
▷ webapplicationtest.dll	0	0,00 %	1189	100,00 %

Figura 4.2: Cobertura de Pruebas Unitarias - Visual Studio 2019

Cabe destacar que se mantuvo la misma consistencia en las pruebas en esta segunda parte de trabajo pero se redujo el porcentaje de cobertura en algunos paquetes debido a la nueva funcionalidad de importadores la cual era muy compleja de probar al 100 %. Si no fuera por este percance, los porcentajes de cobertura mucho más altos.

• { } WebApplication.Controllers	65	18,31 %	290	81,69 %
▷ AdminController	3	8,82 %	31	91,18 %
▷ CategoryController	0	0,00 %	9	100,00 %
▷ ImportController	43	46,74 %	49	53,26 %
▷ ImportController.ImportResult	1	10,00 %	9	90,00 %
▷ ImportController.LodgingImporterModel	0	0,00 %	6	100,00 %
▷ ImportController.Tuple	4	100,00 %	0	0,00 %
▷ LodgingController	0	0,00 %	43	100,00 %
▷ RegionController	0	0,00 %	6	100,00 %
▷ ReservationController	5	9,26 %	49	90,74 %
▷ ReviewController	2	7,41 %	25	92,59 %
▷ SessionController	2	7,41 %	25	92,59 %
▷ TPointController	5	11,63 %	38	88,37 %

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.

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

4.1.1. 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);  
  
            context.Set<State>().Remove(check); LIMPIEZA  
            context.SaveChanges();  
    }  
}
```

Figura 4.3: Ejemplo Prueba Unitaria en PersistenceTest

4.2. 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.

Key	Value	Description
Accept	*/*	
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
token	6F9619FF-8B86-D011-B42D-00C04FC964FF	

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.

Explicación: Logramos una coherencia en el manejo de los endpoints apagándonos al estándar para las RESTful APIs. Utilizamos los verbos de HTTP: Post, 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, ningún componente tiene conocimiento de la estructura interna de los componentes que utiliza, ya que los utiliza por medio de interfaces.

4.2.1. 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.
- **reviews:** Permite el agregado de reseñas sobre las reservas previamente realizadas. Esto actualiza automáticamente el puntaje de los hospedajes. También se muestran estas reseñas al momento de buscar cotizaciones.
- **imports:** Permite la utilización de mecanismos externos para la importación de hospedajes y potencialmente de sus puntos turísticos asociados.

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

4.2.2. 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	Respuestas HTTP	Autent	Comportamiento
[POST]	/sessions/login	[Body] Mail, Password	200 (token) 400 (Exception)	NO	Identifica al usuario. Establece la sesión.
[DELETE]	/sessions/logout	[Header] token	200 () 404 (Exception)	SÍ	Finaliza la sesión indicada
[GET]	/sessions/existing	[Header] token	200 (bool) 404 (Exception)	SÍ	Responde si existe la sesión indicada con el token
[POST]	/admins	[Body] Administrator [Header] token	200 (Administrator) 400 (Exception)	SÍ	Crea un admin y lo retorna
[PUT]	/admins	[Body] Administrator [Header] token	200 () 400 (Exception)	SÍ	Modifica el admin preexistente, a partir del email.
[DELETE]	/admins/{id}	[Header] token, [URI] adminId	200 () 404 (Exception)	SÍ	Elimina al administrador por id
[GET]	/categories		200 ([]Category) 400 ("Error del sistema")	NO	Devuelve todas las categorías de la base de datos.
[GET]	/regions		200([] Region)	No	Devuelve todas las regiones de la base de datos.
[GET]	/tpoints		200([] TouristicPointOutModel) 400 ("Error desconocido")	No	Devuelve todos los puntos turísticos de la base de datos
[GET]	/tpoints	[Query] regionId [Query] [int] categories	200([] TouristicPoint) 404 ("No se encontraron puntos turísticos para la región")	No	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	200 (TouristicPoint) 400(Exception)	Sí	Crea un punto turístico y lo retorna
[GET]	/lodgings	[Query] LodgingSearchModel	200 ([] LodgingsearchModel) 404 ("Error de procesamiento")	No	Devuelve cotización para los hospedajes en el punto turístico indicado, con las fechas y huéspedes indicados.
[GET]	/lodgings	[Query] LodgingSelectionModel	200 ([] Lodging) 404 ("Error de procesamiento")	No	Devuelve los hospedajes cuyo nombre sea similar al solicitado, y cuyo punto turístico coincida con el solicitado.
[POST]	/lodgings	[Header] token, [Body] LodgingModel	200 (Lodging) 400(Exception)	Sí	Crea un hospedaje nuevo y lo retorna

[DELETE]	/lodgings/{id}	[Header] token, [URI] lodgingId	200 () 404 (Exception)	Sí	Elimina el hospedaje indicado por id
[PUT]	/lodgings	[Body] LodgingModel [Header] token	200 () 404 (Exception)	Sí	Modifica el lodging preexistente, ubicándolo a partir del id.
[GET]	/reservations		200 ([] Reservation)	Sí	Busca la reserva y retorna el estado actual
[GET]	/reservations/{code}	[URI] reservationCode	200 (StateModel) 400 ("No existe la reserva") 404 ("Error procesando la solicitud")	Sí	Busca la reserva y retorna el estado actual
[GET]	/reservations/states		200 ([] State)	Sí	Devuelve todos los estados de reserva de la base de datos
[GET]	/reservations/reports	[Query] ReservationReportRequestMode 1	200 (ReservationResultReport Model) 400 (Exception) 400("Error desconocido")	No	Busca la reserva y retorna su estado actual
[POST]	/reservations	[Body] ReservationMode 1	200 (BillModel) 400 (Exception) 400("Error inesperado procesando la reserva")	Sí	Calcula el precio del hospedaje seleccionado según la estrategia de cálculo activa, y retorna una factura con código único de reserva.
[PUT]	/reservations	[Body] ReservationUpdateModel [Header] token	200 () 404 (Exception)	Sí	Modifica una reserva, actualizando su descripción y asignándole uno de los estados de la BD.
[GET]	/imports		200 ([] LodgingImporterModel) 400 ("Error cargando los dll disponibles")	Sí	Lista los importadores de hospedajes disponibles
[POST]	/imports	[Body] LodgingImporterModel	200 (ImportResult) 400 ("Error en la importación")	Sí	Utiliza el mecanismo indicado con el parámetros enviados para importar hospedajes.
[POST]	/reviews	[Body] ReviewModel	200 (Review) 400 (Exception)	Sí	Publica una reseña
[GET]	/reviews/{code}	[URI] reservationCode	200 (bool) 400("Error procesando la solicitud")	No	Retorna la reseña asociada a un código de reserva
[GET]	/reviews/lodging/{id}	[URI] lodgingId	200 ([] Review) 400 ("Error procesando la solicitud")	No	Retorna todas las reseñas asociadas a un hospedaje

4.2.3. 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.

Uso de token de autenticación

Para almacenar la “sesión” del usuario administrador, como las APIs REST no manejan estados, se optó por implementar el uso de un **header** en todas las operaciones que requieren acceso administrativo. Este código es llamado **token**, y su cuyo valor es un **Guid** (identificador único global)[2].

Estos son los pasos que se siguen para la creación del token y su almacenamiento:

1. El usuario administrador se loguea en la aplicación, haciendo uso del endpoint **sessions**, pasando por parámetro su usuario y contraseña.
2. El back-end valida que el usuario exista, y de estar correcta la información de autenticación, verifica si no tiene una sesión previamente iniciada, consultando en la tabla de Sesiones en la base de datos.
3. Si ya existía una sesión iniciada para ese usuario, se le retorna el código identificador de la sesión. De otra forma, se crea la sesión, se almacena en la base de datos, y se le retorna el código.
4. De ahí en adelante, cada acción administrativa que el usuario quiera realizar, deberá contener el token como **header** de la request, ya que las operaciones administrativas lo solicitarán.

Filtros

Cómo se realiza el control de acceso a las funcionalidades restringidas?

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 4.4: 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 4.5: Configuración del filtro

4.2.4. 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)

Bibliografía

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