

Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 2

Obligatorio 2

Entregado como requisito de la materia Diseño de
Aplicaciones 2

Santiago Toscanini (232566)

Sofia Tejerina (209239)

Tutores: Daniel Acevedo, Ignacio Valle

26 de noviembre de 2020

Declaraciones de autoría

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

Resumen

Para la materia Diseño de Aplicaciones 2, en este segundo obligatorio se trabajó en base a la API REST para el Ministerio de Turismo elaborada para la primer entrega. Se implementaron nuevos cambios y funcionalidades, tales como: un nuevo tipo de turista con una nueva forma de descuento, la incorporación de reseñas, posibilidad de hacer reportes, entre otras. También se implementó la interfaz de usuario web que hará uso de la API.

Como se mencionó, además de las funcionalidades brindadas en la entrega anterior, los administradores ahora podrán realizar reportes sobre los hospedajes, también se les da la posibilidad de importar hospedajes desde archivos, los turistas podrán dejar una reseña y ahora los jubilados tienen un descuento especial.

El objetivo de este proyecto es aplicar los conocimientos adquiridos en Diseño de Aplicaciones 1, REST, Mocking, Filters, Inyección de dependencias, Modelo 4 + 1, Patrones de Diseño, Reflection, Métricas, Principios de diseño y Angular utilizando buenas prácticas.

En este documento hablaremos de la solución implementada para este problema; decisión de diseño seguida, con su justificación y diagramas que permitan un mejor entendimiento de la misma.

Índice general

1. Nociones generales del sistema	2
1.1. Errores conocidos y funcionalidades no implementadas	3
2. Arquitectura	5
2.1. Clean Architecture	5
2.2. Implementación	5
3. Desarrollo guiado por pruebas	12
3.1. Pruebas unitarias	13
3.2. Pruebas de integración	14
4. Código fuente	16
4.1. Entities	17
4.2. Infrastructure	18
4.3. Application Core	20
4.4. Factory	21
4.5. Web	21
4.6. Importers	22
5. Métricas	23
6. Anexos	27
6.1. Clean Code y TDD	27
6.1.1. Estándares	27
6.1.2. TDD	29
6.2. Descripción de API	31
6.2.1. REST	31
6.2.2. Descripción de los códigos devueltos	33
6.3. Recursos de la API	33
6.4. Implementación de importer	38
Bibliografía	39

1. Nociones generales del sistema

El Ministerio de Turismo a través de su conocida marca “Uruguay Natural” se encuentra trabajando en un plan de remodelación tecnológica de algunas de sus plataformas, con el fin de poder aprovechar la pausa en la actividad turística ocasionada por la pandemia, pudiendo así retomar la misma en un futuro potenciando la llegada de nuevos turistas tanto nacionales como internacionales.

Particularmente, luego de varios meses de trabajo detectaron algunos problemas con su sitio web actual, siendo el principal de ellos que el contenido es estático y demasiado genérico, por lo que es difícil poder atrapar los potenciales turistas sin ofrecerles una propuesta más atractiva y completa.

En consecuencia, se tomó la decisión de permitir que los posibles turistas tengan una experiencia end-to-end: desde poder explorar lugares turísticos, hasta evaluar y reservar potenciales paquetes turísticos para cada cliente. Para ello, se decidió crear desde cero el sistema.

El sistema fue desarrollado como un monolítico, el mismo permite:
A todos los usuarios:

- Loggearse con un mail y contraseña.
- Se pueden ver las distintas categorías.
- Se pueden ver las distintas regiones.
- Se pueden ver los puntos turísticos de una región y filtrar dicha búsqueda por categorías.
- Se pueden ver los distintos hospedajes (dado un punto turístico y que no tengan la capacidad completa para la cantidad de personas seleccionadas).
- Se puede crear una reserva.
- Se puede ver una reserva (consultar su estado actual dado el código de reserva).
- Se puede dejar una reseña.
- Acceden a distintos descuentos en los hospedajes según la edad.

A los administradores que se encuentren loggeados:

- Desloguearse.
- Crear y eliminar otros administradores.
- Actualizar a los administradores (cambiar su contraseña y/o nombre).
- Dar de alta un punto turístico (dada una región existente).
- Dar de alta un hospedaje (dado un punto turístico existente).
- Borrar un hospedaje.
- Actualizar un hospedaje (modificar su capacidad actual).
- Actualizar una reserva (cambiar el estado actual).
- **Realizar reportes sobre los Hospedajes.**
- **Importar Hospedajes (y sus Puntos Turísticos, en caso de no estar registrados) desde un archivo.**

1.1. Errores conocidos y funcionalidades no implementadas

Algo que faltó implementar, dada la limitante del tiempo, fue hacer una validación de los datos en todas las capas de la aplicación, se realizó únicamente en los modelos y a nivel de interfaz, pero podrían haberse validado en los servicios y a nivel de base de datos. Esto nos permitiría asegurarnos que los datos serán consistentes e independientes a los cambios en los distintos componentes del proyecto.

Si bien las funcionalidades nuevas en el backend las implementamos siguiendo TDD, una función privada que se encuentra dentro de uno de los Modelos no se está contando como parte de la cobertura aunque se este testeando su funcionalidad, investigando porque podría estar pasando eso nos encontramos con que es un problema de cómo el compilador traduce la función lambda que llama al método¹.

Se permite importar Hospedajes desde archivos externos, si ocurre un error durante el guardado de los datos, porque algún dato es incorrecto, no se realiza un aborto de todos los datos guardados previamente. Esto podría causar problemas si se vuelve a intentar importar con el mismo documento pueden quedar datos duplicados. Tampoco se le informa al usuario si el archivo que envió es inválido, le devuelve que están creados los hospedajes, haciendo que tenga una mala experiencia ya que no sabe si sus datos se guardaron realmente hasta que los consulte. Estos errores no se pudieron arreglar por falta de tiempo, pero se espera poder resolverlos lo más

¹Foro donde se explica la causa del error en la cobertura.

pronto posible.

Creamos la entidad Huésped con la intención de mostrar sus datos en el frontend, pero por falta de tiempo esto no se implemento, por lo que los datos de rango de la entidad no se utilizan. Se espera poder agregar esta información al frontend lo más pronto posible.

En la interfaz de usuario tenemos un error. Cuando la resolución excede los 992 pixels de ancho el contenido de la pagina se ve sobre la izquierda y muy angosto, se intentó resolver pero por falta de tiempo no se pudo encontrar la solución.

2. Arquitectura

2.1. Clean Architecture

Seguimos los principios y patrones de desarrollo Clean Architecture, basandonos en la siguiente arquitectura la cual es la recomendada por Microsoft para .NET. La misma tiene un repositorio¹ que ejemplifica el uso.

Esta Arquitectura Limpia propuesta por Microsoft es similar a la llamada "Cebolla", donde todas las capas exteriores dependen de las interiores, siendo las entidades (el negocio) el centro, no dependiendo de ninguna otra capa.

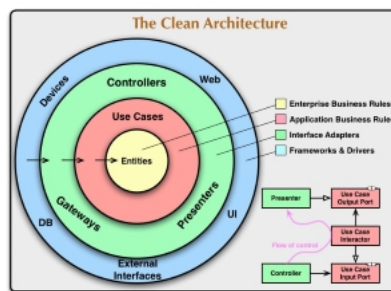


Figura 2.1: Representación de la arquitectura "Cebolla".

Si bien nos basamos en esta arquitectura; fuimos realizándole algunas adaptaciones que consideramos que se acoplaban de mejor forma a nuestra solución, tales como: separar las Entidades de ApplicationCore, evitando dependencias circulares entre proyectos y haciendo más limpia la arquitectura, por lo tanto más reutilizable y mantenible, y tener en proyectos separados las interfaces para mantener separadas implementaciones de interfaces, lo que permite que sea más fácil la reutilización o reemplazo de componentes. Todas estas adaptaciones permitieron seguir con el principio de responsabilidad única de SOLID.

2.2. Implementación

La implementación contempla algunas adaptaciones a la Arquitectura Oficial Propuesta por Microsoft, como se comentó en la sección anterior.

¹Repositorio que ejemplifica el uso de la arquitectura recomendada por Microsoft.

Separamos el contenido de la solución en dos directorios: src para todo lo referente al código fuente) y test (para los tests, ya sean unitarios o de integración).

Dentro de el directorio test nos encontramos con dos proyectos:

Uno que corresponde a las Pruebas de Integración (únicamente para la base de datos, levantando una en memoria), otro con los tests unitarios, dentro de estos podemos encontrar las distintas clases de test clasificadas en las mismas carpetas que las clases que están probando en src.

Dentro de src encontramos la siguiente estructura de paquetes:

- **InfrastructureInterface:** Contiene las interfaces para manejar los datos de las entidades. De esta forma generamos una mayor mantenibilidad, cambiar nuestra forma de implementar las persistencia en datos implicaría únicamente crear un nuevo proyecto que implemente las interfaces y cambiar las implementaciones que se van a usar en la clase Factory.

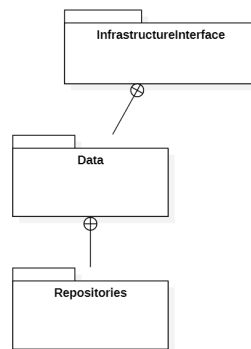


Figura 2.2: Diagrama de descomposición de namespaces de `InfrastructureInterface`.

- **Infrastructure:** Contiene las implementaciones de las clases definidas en `InfrastructureInterface`, es donde se encuentran los `Repositories`, las configuraciones de las entidades para EF Core, el Context como así también las migraciones ya que nos pareció que como esta relacionado a EF era correcto que estuviera todo contenido en un mismo paquete, para mantener nuestros proyectos clasificados y así hacer más fácil el uso (y entendimiento) de nuestro código.

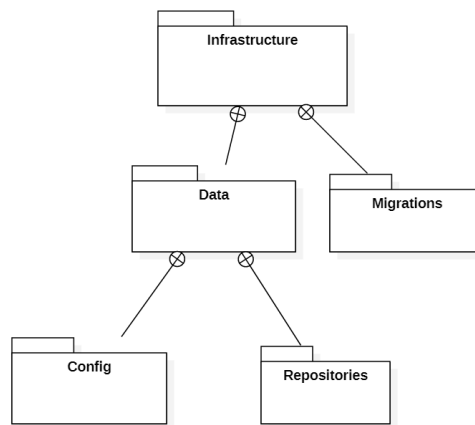


Figura 2.3: Diagrama de descomposición de namespaces de Infrastructure.

- **ApplicationCoreInterface:** Contiene las interfaces cuyas implementaciones contendrán la lógica de negocio. Así como hicimos con la infraestructura decidimos que nuestro código dependiera de abstracciones, generándole una mayor mantenibilidad; cambiar la implementación de cualquiera de los servicios solamente implicaría implementar la interfaz correspondiente y modificar la implementación utilizada en la clase Factory, sin que ningún otro paquete se vea afectado por esto.

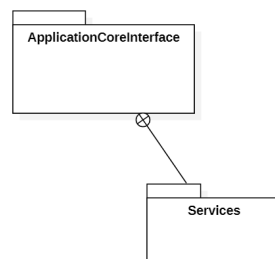


Figura 2.4: Diagrama de descomposición de namespaces de ApplicationCoreInterface.

- **ApplicationCore:** Este paquete contiene las implementaciones de ApplicationCoreInterface, los repositorios que utiliza son interfaces de InfrastructureInterface, de esta manera se genera un bajo acoplamiento, cabe aclarar que en este paquete también se encuentra la implementación de la interfaz ISessionService ya que esta se encuentra fuertemente acoplada a nuestro negocio, necesita acceso a repositorios, etc. Se pueden encontrar también servicios que no hacen uso de repositorios, se encargan de implementar cierta lógica que es utilizada o utiliza a otros servicios, siguiendo el Common Reuse Principle (donde las clases que se reutilizan juntas deberían estar agrupadas juntas) se decidió dejar el servicio encargado de calcular los precios de los hospedajes y el de manejar los hospedajes importados en este paquete.

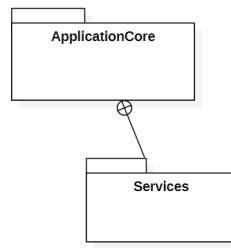


Figura 2.5: Diagrama de descomposición de namespaces de ApplicationCore.

- **SessionInterface:** En este paquete esta una interfaz utilizada para el log-in de los usuarios. Decidimos mantenerla separada del paquete ApplicationCoreInterface debido a que no estaba tan relacionada con el negocio y para permitir implementar nuevas formas de inicio de sesión (con SSO por ejemplo).



Figura 2.6: Diagrama de descomposición de namespaces de SessionInterface.

- **Importers:** Este paquete es que el permite que el sistema tenga extensibilidad para importar hospedajes desde diferentes formatos, sin la necesidad de recompilar el código. En él se encuentra la interfaz que el servicio encargado de manejar las importaciones utiliza para obtener los datos, también se encuentran aquí los modelos necesarios para saber qué formatos de datos se están esperando. Para implementar la lectura de un nuevo tipo de archivo basta con importar este paquete y crear su propia implementación. Este paquete fue creado siguiendo el Release Reuse Equivalency Principle, para que quien desee hacer su implementación se acople solamente a datos que le interesan. El criterio seguido para poner los modelos y la interfaz juntos en este paquete fue el Common Closure Principle, dejando juntos paquetes que cambiarán por las mismas razones.

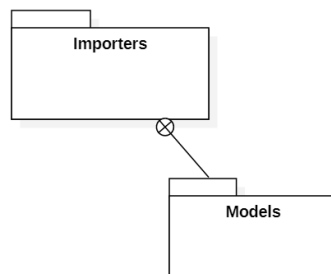


Figura 2.7: Diagrama de descomposición de namespaces de Importers.

- **Entities:** Aquí se encuentran las clases relacionadas con el negocio. Es donde se encuentran los módulos menos propensos al cambio, estos no dependen de nadie, mientras que el resto del sistema depende de ellos, esto se debe a que al ser parte del negocio son estables, es poco probable que el negocio cambie (luego de salir a producción), sin embargo la tecnología si es propensa a los cambios así que no se puede depender de ella.

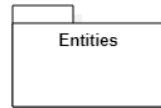


Figura 2.8: Diagrama de descomposición de namespaces de Entities.

- **Exceptions:** Contiene todas las excepciones creadas por los desarrolladores para el sistema. Lo mejor fue dejarlas en un paquete aparte ya que pueden ser utilizadas por distintos proyectos.

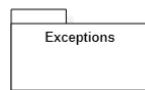


Figura 2.9: Diagrama de descomposición de namespaces de Exceptions.

- **Factory:** Este paquete fue creado con la finalidad de tener la clase Setup del proyecto Web más limpia y libre de dependencias, solo esta utiliza los métodos de la única clase del proyecto. Factory es el encargado de la inyección de dependencias, evita que tanto Web como otros proyectos dependan de las implementaciones, generando una mayor mantenibilidad, legibilidad del código y manteniendo a Web con la responsabilidad de ser la primer capa de comunicación con la api (cumpliendo el SRP).

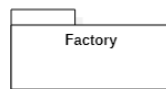


Figura 2.10: Diagrama de descomposición de namespaces de Factory.

- **Web:** Dentro de Web podemos encontrar un directorio con los distintos controllers, uno por cada entidad de las funciones solicitadas por el cliente. También tenemos un directorio con los DTOs llamado Models, dentro se encuentran en distintos directorios, según la entidad a la que corresponden, los distintos modelos que son utilizados por los controllers. También se encuentran los Fileters que se utilizan para hacer validaciones. como la autenticación del usuario que

hace la request antes de que llegue al controller, como atrapar las excepciones generando mensajes claros para el usuario luego de salir del controller.

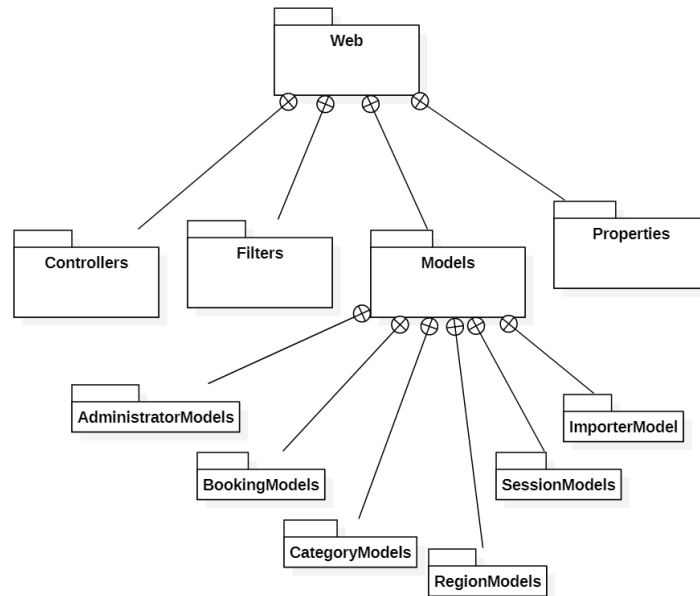


Figura 2.11: Diagrama de descomposición de namespaces de Web.

Finalmente, en la siguiente figura se puede ver en el diagrama de componentes como los paquetes (componentes) se relacionan unos con otros a través de interfaces o dependencias.

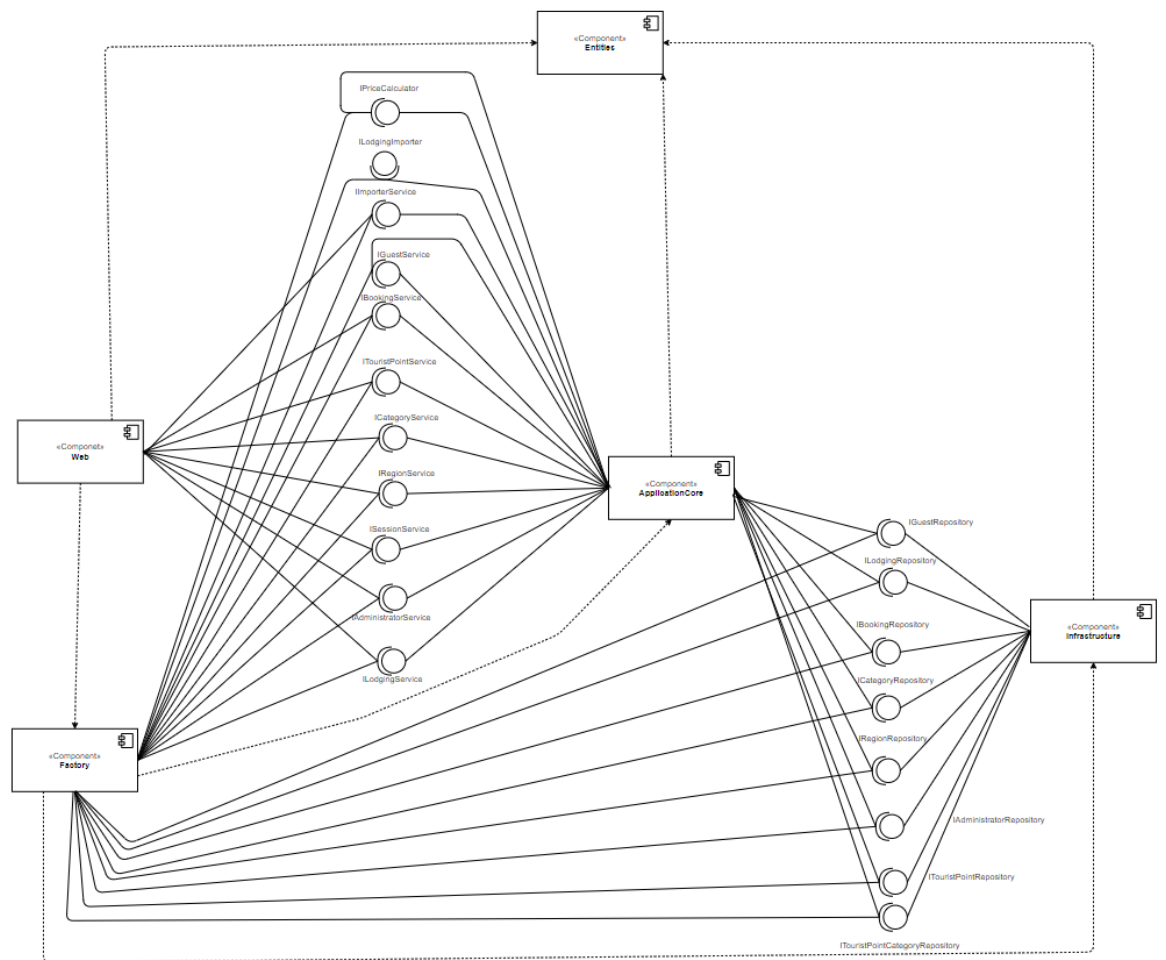


Figura 2.12: Diagrama de componentes.

3. Desarrollo guiado por pruebas

Es una metodología de desarrollo que se basa en crear primero las pruebas y luego escribir el software necesario para que estas pruebas pasen, seguido por una etapa de refactor del código implementado.

Consta de un ciclo formado por lo anteriormente mencionado: creación de test, implementación del código necesario para que el test pase y refactoring de el código:

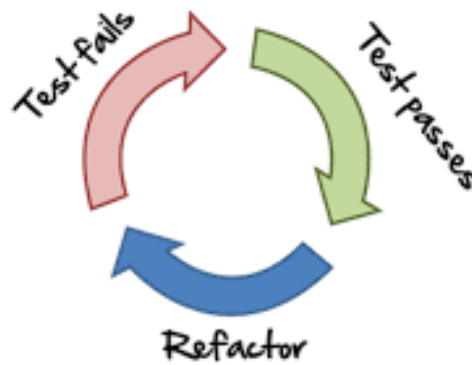


Figura 3.1: Ciclo de TDD.

Tiene múltiples ventajas, como:

- Modificar el código de forma segura, ya que las pruebas nos permiten saber si el cambio ha hecho que algo no funcione como se espera.
- Tener tests de nuestro código completos.
- Evitar código innecesario, haciendo más limpia nuestra solución.
- Los tests funcionan a modo de "documentación" de nuestra API, a través de ellos se puede saber qué hace nuestra solución.
- Durante el desarrollo nos permite ver claramente dónde estamos parados.

Nuestros Test se encuentran en un directorio separado del código fuente llamado "tests", dentro se podrán encontrar con dos proyectos; uno para los test unitarios y otro para los de integración. Decidimos separarlos así para hacer más clara la

arquitectura y a su vez mantener los proyectos con sus respectivas responsabilidades únicas. Ya que los test complementan a que el sistema sea más mantenible, lo mejor es mantenerlos ordenados. En las siguientes secciones hablaremos un poco más sobre cada proyecto.

3.1. Pruebas unitarias

Las pruebas unitarias prueban como su nombre indica funcionalidades unitarias, es decir, probaremos una funcionalidad pero si esta utiliza métodos de otra clase no corresponde probarlos, sino se convertiría en una prueba de integración. Nos limitaremos a probar aquello que la clase que está siendo testeada realiza, mockeando aquellas funciones ajenas a ella que utiliza.

Los Mocks nos permiten simular el comportamiento de aquellas funcionalidades que nos nos corresponde probar y la funcionalidad testeada utiliza. Para esto le indicamos que métodos esperamos que se llamen al ejecutar las funcionalidad testeada, que parámetros recibirá y que respuesta devolverá.

En el siguiente ejemplo, extraído de nuestra solución podemos ver un ejemplo del uso de mock. Se simula el objeto `IBookingRepository`, de él se utilizará el método `"GetAll()"` que no recibe parámetros y va a retornar la lista creada anteriormente. Ese mock se lo pasaremos, como objeto, por parámetro al constructor de la clase que se esta probando. Arriba del `Assert`, el método `"VerifyAll()"` nos permite asegurarnos de que estamos simulando solo el comportamiento que se utiliza y no más.

```
[TestMethod]
public void TestGetAllOk()
{
    var bookingsToReturn = new List<Booking>
    {
        new Booking
        {
            Code = _bookingCode1,
        },
        new Booking
        {
            Code = _bookingCode2,
        },
    };

    var mock = new Mock<IBookingRepository>(MockBehavior.Strict);
    mock.Setup(r => r.GetAll()).Returns(bookingsToReturn);
    var bookingService = new BookingService(mock.Object);

    IEnumerable<Booking> bookingsSaved = bookingService.GetAll();

    mock.VerifyAll();
    Assert.IsTrue(bookingsSaved.SequenceEqual(bookingsToReturn));
}
```

Figura 3.2: Ejemplo de uso de mock en test unitario, clase `BookingServiceTest`.

Pero no solamente se utilizan Mocks, si bien son sumamente importantes y útiles, existen otros como Fake, utilizamos un Fake en las pruebas de integración, con la base de datos local, sustituye el comportamiento de la base de datos externa pero no puede ser utilizado en producción.

Los Test Unitarios fueron realizados siguiendo la práctica de TDD, como se ve en el siguiente ejemplo:

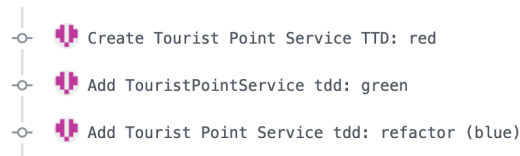


Figura 3.3: Ejemplo de uso de TDD con tests unitarios, github rama feature/createTouristPoint.

El proyecto donde se encuentran las Pruebas unitarias esta organizado de forma similar a como están los proyectos creados en nuestro directorio de código fuente: Web, Entities y ApplicationCore.

Dentro de Web seguimos encontrando dentro de directorios "paralelos" al proyecto Web de la carpeta src: Los Controllers y los Filters.

Dentro de ApplicationCore encontramos un directorio llamado "Services", nuevamente siguiendo la estructura utilizada en el proyecto ApplicationCore.

Esta organización permite visualizar claramente qué se esta probando y dónde encontrarlo, tanto si se viene desde el código fuente como si se empieza por los tests.

3.2. Pruebas de integración

Los test de integración, a diferencia de los unitarios, buscan probar una funcionalidad y sus dependencias, como su nombre lo dice prueban la integración de varias partes de nuestro sistema, ya no debemos mockear aquellos métodos pertenecientes a otras clases que las clase que estamos probando utiliza. Son muy útiles para detectar cuando un cambio afecta a una dependencia de la clase modificada, en nuestro caso, si se decide cambiar la implementación del repositorio, y no se implementa el nuevo de forma correcta, rápidamente los test nos informarán qué es lo que no esta funcionando como se espera.

En el siguiente fragmento de código podemos ver como se inicializa la base de datos local antes de cada test y como esta se limpia luego de terminado este.

```

[TestClass]
public class BookingRepositoryTest
{
    private IBookingRepository _bookingRepository;
    private DbContext _context;

    private readonly string _code = "test_code123";
    private readonly string _code2 = "test_code456";
    private readonly string _description = "This is a dummy description";

    [TestInitialize]
    public void Setup()
    {
        DbContextOptions<TourismContext> options = new DbContextOptionsBuilder<TourismContext>()
            .UseInMemoryDatabase(databaseName: "database_test")
            .Options;
        _context = new TourismContext(options);
        _bookingRepository = new BookingRepository(_context);
    }

    [TestCleanup]
    public void TestCleanup()
    {
        _context.Database.EnsureDeleted();
    }
}

```

Figura 3.4: Ejemplo de uso de una base de datos local en test de integración, clase BookingRepositoryTest.

Estos test fueron realizados durante el ejercicio de desarrollo siguiendo TDD, para las clases relacionadas con la persistencia de datos; llamados "Repositorios".

En la siguientes imagenes podemos ver como se realizaron ciclos de TDD para el desarrollo de los repositorios, creando los test de integración primero.

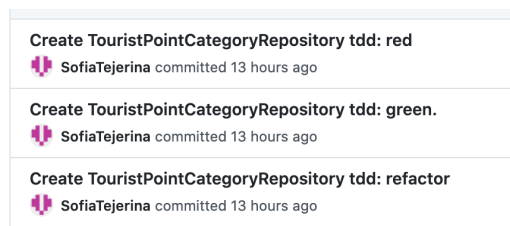


Figura 3.5: Ejemplo de seguimiento de TDD en Repositorios, github rama feature/-createTouristPoint.

Nuestros Test de integración, al igual que los unitarios, están organizados de tal forma que se sea fácil visualizar qué y dónde se encuentra lo que se está probando en cada clase, viendo desde el código fuente a los test o viceversa.

4. Código fuente

El código fuente se encuentra dentro del directorio de nombre "src" dentro de la solución. Divididos en determinados componentes (proyectos), podemos encontrar el código que conforma nuestra API. Estos proyectos se irán detallando en las siguientes secciones, explicando su responsabilidad y un poco más sobre su contenido.

Para dar una mirada más general antes de adentrarnos en cada proyecto mostraremos el siguiente diagrama de dependencias, que permite entender por qué se decidieron organizar de esta forma.

Como se puede ver en el diagrama se buscó dejar los componentes lo más independientes unos de otros, permitiendo seguir con el principio de responsabilidad única, tratando de lograr una alta cohesión y un bajo acoplamiento entre los proyectos, haciendo uso de patrones de diseño como Inyección de dependencias (que nos permite seguir el principio de Inversión de Dependencias), aplicación de Liskov Substitution Principle gracias al uso de interfaces. Tratar de mantener un bajo acoplamiento fue motivado también para cumplir con el principio de Dependency inversion cuanto fuera posible.

Todas estas buenas prácticas (junto con Clean code y estándares de codificación) buscan generar una solución de calidad, que sea fácil de entender, adaptar y modificar, haciendo que perdure en el tiempo sin disminuir la productividad del equipo.

Como se puede ver en el siguiente diagrama de paquetes seguimos principios como Acyclic Dependency Principle, no permitiendo que haya dependencias cíclicas entre los paquetes, también seguimos el Stable Dependency Principle; donde los paquetes mas inestables dependen de los mas estables, esto se verá más a fondo en el capítulo de métricas, pero un ejemplo concreto de esto es Web (que tiene la inestabilidad más alta) por lo que nadie depende de él y todos dependen de Entities (el paquete más estable del proyecto). Tratamos de seguir el principio de Stable Abstraction pero no se cumple estrictamente para casos como el nombrado anteriormente, todos dependen de Entities que no es abstracto, pero sí procuramos hacer que la mayoría de los paquetes dependieran de paquetes sumamente abstractos como SessionInterface o ApplicationCoreInterface, en lugar de los paquetes con clases concretas como ApplicationCore.

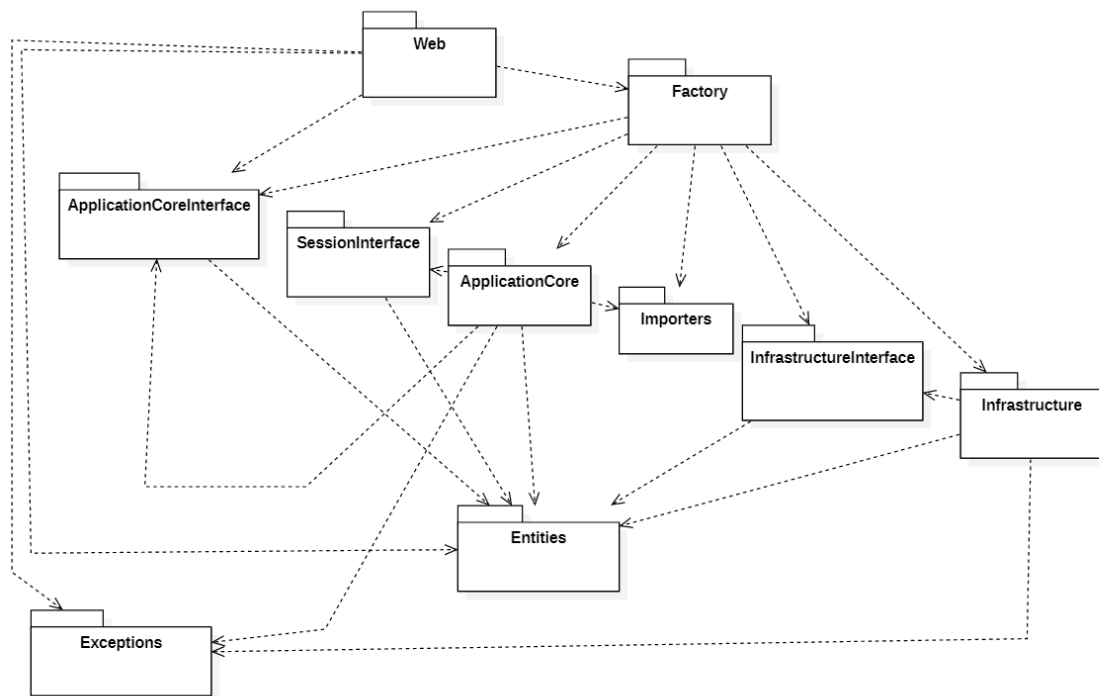


Figura 4.1: Diagrama de paquetes de la solución.

4.1. Entities

Dentro de la carpeta "src" el Proyecto Entities es el único que no depende de otro, es donde se almacenan las entidades relacionadas al dominio del negocio.

Decidimos mantenerlo como un archivo separado de los otros para que no se viera afectado su contenido si alguna otra capa cambiaba, ya que la única razón de cambio de las clases de este paquete debe ser consecuencia de cambios en el negocio.

Tuvimos que crear una entidad para permitir la relación N a N entre el Punto turístico y las categorías, debido a que Entity Framework con .Net Core no lo soporta. Con esta clase auxiliar "TouristPointCategory", la relación N a N pasa a ser dos relaciones 1 a N, entre la clase Punto turístico con la auxiliar, y la clase Categoría con la clase auxiliar.

La reserva puede estar en distintos estados, dado que suponemos que esto no cambiara (suposición hecha a partir de los requerimientos del cliente) decidimos implementarlo como un Enumerado, de esa forma ya quedaban los valores fijos que podía tomar el estado de una reserva, sin tener que preocuparse por agregarlos por fuera, como pasa con las Categorías y Regiones.

Si bien se sabe que para esta versión las Regiones y las Categorías son fijas, decidimos crearlas como entidades en lugar de con enumerados (como hicimos con

el estado de la reserva), esto se debe a que es probable que cambie (según los requerimientos del cliente), y de esa forma no tendríamos que hacer grandes cambios cuando ocurra.

Creamos una clase especial para los huéspedes, en lugar de hacerlo como se hicieron los tipos de estados, esta decisión de diseño surgió debido a la cantidad de características que distinguen a un tipo de huésped de otro (su nombre, el rango de edades y el porcentaje que pagan del total), de esa forma no se necesitaría acceder al código para quitar o agregar nuevos. Modificarlos agregándoles algún otro atributo también es muy simple, es decir que la solución hace que sea más mantenible.

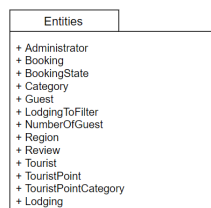


Figura 4.2: Paquete Entities, con sus clases y visibilidad.

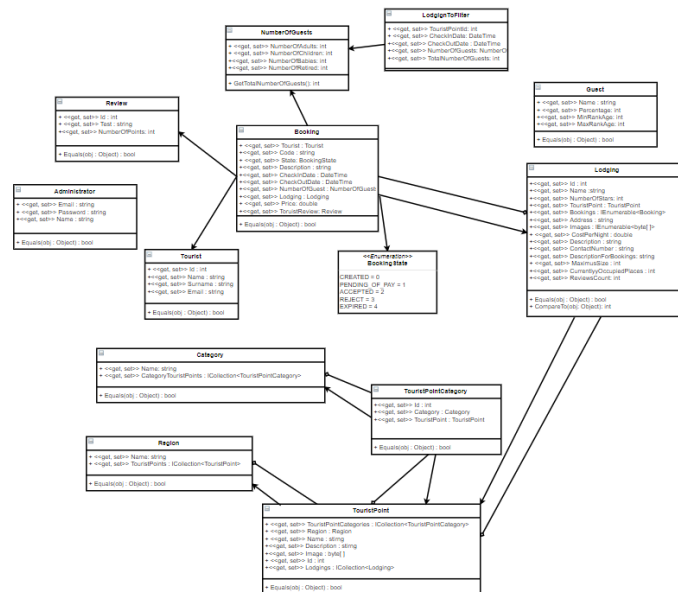


Figura 4.3: Diagrama de clases de Entity.

4.2. Infraestructure

Dentro de este proyecto se encuentra en el segundo nivel nuestro Contexto, decidimos dejarlo a este nivel ya que abarca todo aquello relacionado con la Data y la persistencia, podría dar confusión y no estaría representando bien lo que significa si

lo metiéramos dentro de otro nivel.

Al mismo nivel que el paquete Data podemos encontrar las migraciones, estas son las que nos permiten conservar e ir adaptando el estado de nuestra base de datos a medida que vamos realizándole modificaciones a la entidades. Guardan los estados y ven las diferencias cuando se crea una migración nueva para no perder todos los datos hasta el momento. Se encuentra en este paquete ya que esta íntimamente ligada al acceso a datos, no sería correcto dejarlo en un proyecto aparte.

Para la persistencia de los datos utilizamos el ORM de Microsoft Entity Framework, gracias a el pudimos generar la base de datos a partir de nuestro código fuente, lo que se conoce como "Code-First".

También se puede encontrar dentro de este paquete un archivo de configuración "appsettings.json", aquí se encuentra el String de conexión a la base de datos, si bien se puede usar directamente en el código, pero eso tendría ciertas desventajas; cada vez que se corra el código en una maquina diferente este string debería ser modificado, sin embargo con el archivo cada uno tiene su versión y no debe realizar ningún paso extra para poder hacer uso de su persistencia de datos.

Se podrá acceder a los datos mediante los Repositories, estos contienen objetos de EntityFramework tales como DbSet y DbContext, que guardan los datos y la conexión con la base de datos respectivamente. Estas clases corresponde que vayan dentro de un mismo namespace ya que están relacionadas en su responsabilidad: persistir las distintas entidades del negocio en la base de datos.

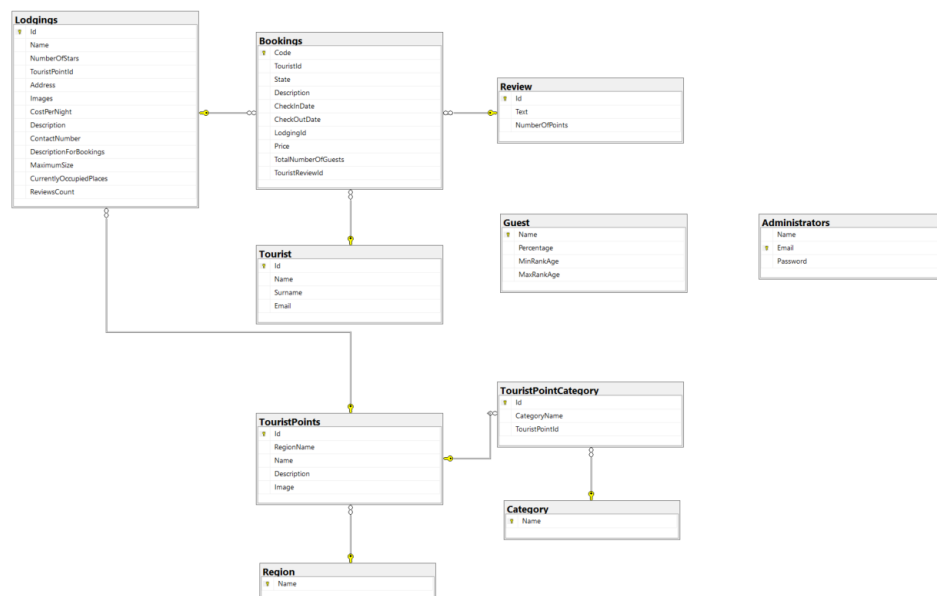


Figura 4.4: Modelo de tablas de nuestra base de datos.

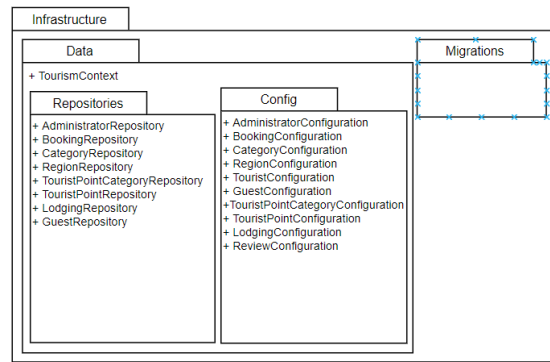


Figura 4.5: Paquete Intrastructure, con sus clases y visibilidad.

4.3. Application Core

Dentro del Paquete ApplicationCore se encuentran los servicios, estos son los encargados de interactuar entre los Repositorios (el acceso a datos), tienen la responsabilidad de brindarle a la API los datos que necesite y persistir los datos que llegan y deben guardarse. Permite evitar que se viole el principio de responsabilidad única, tener tanto el código como la arquitectura más limpias y claras, así como facilitar la reutilización o reemplazo de código. En el futuro se podría decidir realizar ciertas operaciones con los datos que llegan a la API, antes de ser guardados, basta con crear los Servicios que hereden de las interfaces de los Servicios y le agreguen la nueva implementación, sin que nunca se vea afectado el acceso a datos o la API, permite entonces un bajo acoplamiento entre ambas partes y mantiene alta su cohesión.

Dentro de este paquete tenemos clases encargadas de realizar operaciones sobre los datos antes de ser enviados a capas superiores (como Web) o inferiores (como Repositories), ejemplo de esto es 'PriceCalculatorService', el servicio encargado de hacer los cálculos de los descuentos de los distintos tipos de huéspedes y obtener así el precio real a pagar por el hospedaje. Con esta clase los cambios en los tipos de descuentos no afecta a otros servicios, como LodgingService o BookingService, haciendo más mantenible el sistema, además de quitar la responsabilidad de realizar los cálculos a los servicios que ya tienen sus propias responsabilidades (siguiendo el principio de responsabilidad única).

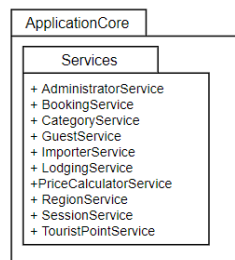


Figura 4.6: Paquete ApplicationCore, con sus clases y visibilidad.

4.4. Factory

Como nombramos en capítulos anteriores, este paquete fue creado con la intención de tener la clase Startup del proyecto Web más limpia. Es la encargada de las dependencias, como esta fuera del paquete de web permite reducir las dependencias entre Web y el resto de los componentes concretos.

Reducir las dependencias, permitir tener clases mas limpias y la inyección de dependencias son muy buenas practicas de codificación, ya que dejan un código mas limpio y fácil de entender y modificar, además de liberar de responsabilidades indebidas a las clases que utilizan otras (ya no deben crear instancias de objetos para utilizar una clase, la inyección de dependencias se encarga de esto por fuera).

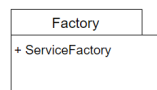


Figura 4.7: Paquete Factory, con sus clases y visibilidad.

4.5. Web

Este Proyecto es el punto de comunicación entre la api, nuestro sistema y el exterior.

En el primer nivel se encuentran las clases de Program y Startup (encargada de la inyección de dependencias, por lo que se comunica con Factory). Luego Podemos ver Clasificado en distintas carpetas los objetos según sus responsabilidades, los DTOs se encuentran en una carpeta y a su vez dentro están clasificados según la Entidad que representan. Esto permite mantener cierto orden dentro de las carpetas de este proyecto.

Utilizamos filters (que se encuentran en su carpeta), que permiten realizar validaciones antes y después de acceder al controller, libran de ciertas responsabilidades al controller (como crear los errores o validar la autorización), mantienen el código limpio y son fácilmente reutilizables.

Los controllers son los encargados únicamente de obtener los pedidos del cliente, a través de los DTOs y transferirlos a la capa de servicios, que se encargara de la lógica.

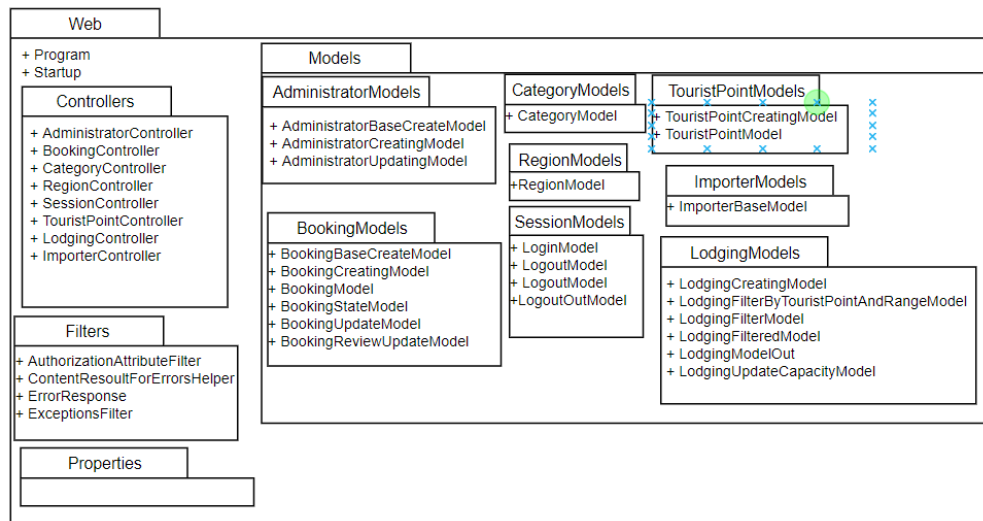


Figura 4.8: Paquete Web, con sus clases y visibilidad.

4.6. Importers

Este paquete es el que permite al sistema usar 'Reflection' una clase del paquete System de .Net que permite obtener información sobre los ensamblados cargados.

Dentro de él se encuentra la interfaz "ILodgingImporter", la cual deberán implementar aquellos que quieran crear su propia versión de Importador de Hospedajes, esta clase es utilizada por un servicio que se encuentra dentro del paquete "AplicacionCore" para enviar los datos importados al repositorio correspondiente.

En un directorio dentro del paquete se encuentran los Modelos (DTOs) que utilizará el importador para enviar los datos al servicio.

El paquete esta pensado para soportar cambios en el importador o nuevos tipos de importadores sin afectar otros paquetes, siguiendo el principio de paquetes: "Common Closure Principle". Gracias a que tanto los modelos como los importers se encuentran en él, también sigue el principio de "Release Reuse Equivalency", haciendo que las implementaciones solo tengan que acoplarse a clases que le interesan.

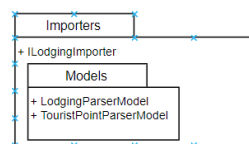


Figura 4.9: Paquete Importers, con sus clases y visibilidad.

5. Métricas

Las métricas son una medida cuantitativa que nos permite saber el grado en que nuestro sistema y sus componentes poseen un atributo de calidad dado. En esta sección vamos a trabajar con métricas a nivel de paquetes, tales como: Inestabilidad, Abstracción y Complejidad Ciclomática. Realizaremos un análisis de las métricas en base a principios de diseño relacionados con ellas.

La Complejidad Ciclomática es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. Es una de las métricas de software de mayor aceptación, ya que es independiente del lenguaje. Es el resultado de contar el número de ciclos diferentes que se siguen en un fragmento de código de un programa habiendo creado una rama imaginaria desde el nodo de salida al nodo de entrada de flujo correspondiente a este fragmento de código. Como se puede ver la imagen nuestro código en su mayoría mantiene una complejidad menor a 10 y siempre por debajo del 20, los valores comprendidos entre 1 y 10 significan que es un programa sin mucho riesgo y los valores entre 11 y 20 que tiene un riesgo moderado.

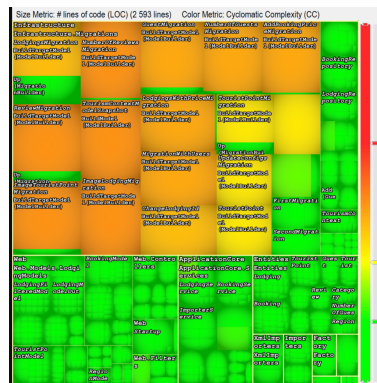


Figura 5.1: Complejidad Ciclomática

En la siguiente figura podemos ver la matriz de dependencia. Los cuadrados azules simbolizan dependencias entrantes (dependencias que van desde el paquete de las abscisa a la ordenada) y los verdes dependencias salientes (de la ordenada a la abscisa), el numero que se dentro es la cantidad de veces que se utilizan elementos del paquete correspondiente.

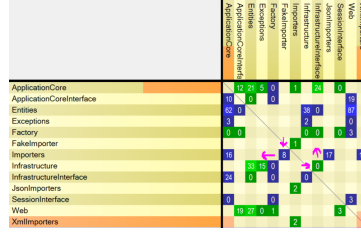


Figura 5.2: Matriz de dependencia

La matriz se analiza de la siguiente forma:

Realizaremos los cálculos para saber el nivel de estabilidad de nuestras clases. Para eso debemos conocer el acoplamiento aferente C_a (dependencias entrantes) y el acoplamiento eferente C_e (dependencias salientes), y calcularemos la inestabilidad de paquete con la siguiente fórmula $\frac{C_e}{C_a + C_e} = I$, que nos dará un valor entre 0 y 1 (0 significa mayor estabilidad y 1 mayor inestabilidad).

Las dependencias salientes las veremos en el siguiente ejemplo con FakeImporter e Importer. Nos fijaremos en los cuadrados de color verde:

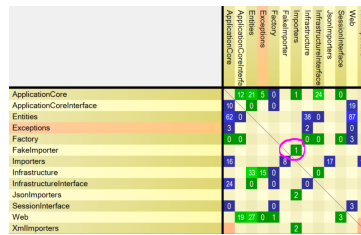


Figura 5.3: Matriz de dependencia, ejemplo dependencia saliente entre FakeImporter e Importer

Tenemos entonces que se utiliza solo una vez en el paquete FakeImporter el paquete Importers, igualmente para nuestros cálculos solo nos fijaremos en la cantidad de cuadrados verdes y azul, no en el numero que contienen.

Las dependencias entrantes las veremos en el siguiente ejemplo con Web y Factory, fijándonos en los cuadrados color azul:

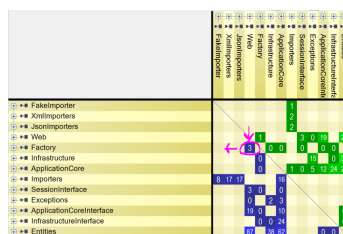


Figura 5.4: Matriz de dependencia, ejemplo dependencia entrante entre Web y Factory

Ahora vamos a calcular la estabilidad de todos los paquetes (excluyendo los paquetes de implementaciones de importers, ya que son de terceros):

- **ApplicationCore** $C_a = 1$, $C_e = 6$, $I = 0.86$, es Inestable
- **ApplicationCoreInterface** $C_a = 3$, $C_e = 1$, $I = 0.25$, es poco Estable
- **Entities** $C_a = 5$, $C_e = 0$, $I = 0$, es Estable
- **Exceptions** $C_a = 3$, $C_e = 0$, $I = 0$, es Estable
- **Factory** $C_a = 1$, $C_e = 5$, $I = 0.83$, es Inestable
- **Importers** $C_a = 4$, $C_e = 0$, $I = 0$, es Estable
- **Infrastructure** $C_a = 1$, $C_e = 3$, $I = 0.75$, es Inestable
- **InfrastructureInterface** $C_a = 3$, $C_e = 1$, $I = 0.25$, es poco Estable
- **SessionInterface** $C_a = 3$, $C_e = 0$, $I = 0$, es Estable

La estabilidad de nuestros paquetes está relacionada con la cantidad de trabajo necesario para realizar un cambio, en los paquetes estables sera fácil hacer un cambio mientras que en los inestables un cambio puede resultar en más trabajo del que se espera.

Ahora veremos la abstracción en nuestros paquetes, la cual esta dada por la cantidad de clases en nuestro paquete (N_c) en función de la cantidad de clases abstractas dentro de él (N_a): $\frac{N_a}{N_c} = A$.

- **ApplicationCore** $N_a = 0$, $N_c = 10$, $A = 0$, es Concreto
- **ApplicationCoreInterface** $N_a = 9$, $N_c = 9$, $A = 1$, es AbstractO
- **Entities** $N_a = 0$, $N_c = 13$, $A = 1$, es Concreto
- **Exceptions** $N_a = 0$, $N_c = 3$, $A = 0$, es Concreto
- **Factory** $N_a = 0$, $N_c = 1$, $A = 1$, es Concreto
- **Importers** $N_a = 1$, $N_c = 3$, $A = 0.30$, es poco Abstracto
- **Infrastructure** $N_a = 0$, $N_c = 19$, $A = 1$, es Concreto
- **InfrastructureInterface** $N_a = 8$, $N_c = 8$, $A = 1$, es Abstracto
- **SessionInterface** $N_a = 1$, $N_c = 1$, $A = 1$, es Abstracto

Saber que tan abstracto es nuestro paquete nos ayuda a saber que tan fácilmente puede ser extendido, los paquetes estables y abstractos son fáciles de extender, cosa que se puede realizar para hacer modificaciones en el código siguiendo el Open Closed Principle.

Viendo al siguiente gráfica podemos corroborar lo que las métricas nos indican; la mayoría de nuestros paquetes se encuentran en la zona verde o secuencia principal, no son ni muy abstractos ni muy inestables, tenemos casos como el de SessionInterface que es muy estable y abstracto (algo que queríamos lograr creando ese paquete, así se podía reutilizar en caso de que se decidiera manejar sesiones con otras tecnologías como cuentas de Google, entre otras.) o Web que es muy inestable y nada Abstracto, razón por la cual él depende de otros componentes pero ninguno de él (siguiendo el patrón de diseño de paquetes Estables o SDP).

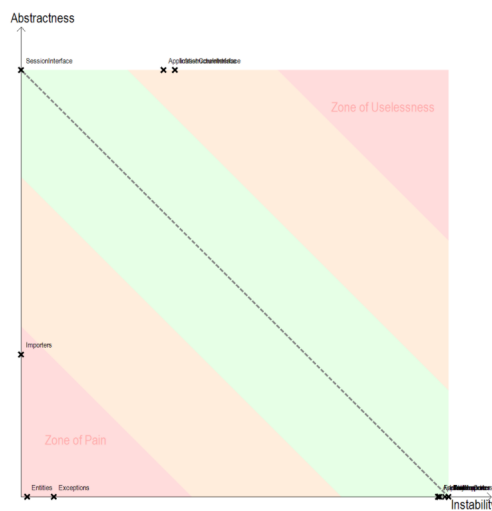


Figura 5.5: Gráfica de Abstracción vs Estabilidad de nuestros paquetes

Tenemos resultados esperados en la zona de peligro (paquetes que no son extensibles y si cambian impactan en otros), como Entidades; el cual es muy estable (no depende de nadie y todos de él) y no es abstracto (porque es el paquete más acoplado al negocio). Importers se encuentra en esta zona por la decisión de mantener la clase interfaz, que importarán quienes quieran realizar sus propios importadores, junto con los Modelos, esta decisión de diseño fue realizada siguiendo el Common Closure Principle, pero dado el resultado obtenidos se deberían hacer modificaciones en las decisiones elegidas para esta implementación, separando los modelos de la interfaz. Un resultado que no se esperaba era el de Exceptions y sobre el cual se deberá hacer una investigación más detallada.

6. Anexos

6.1. Clean Code y TDD

6.1.1. Estándares

Clean Code y estándares de C#

Seguimos el estándar oficial de Microsoft para C# .NET. También usamos como referencia un repositorio que explica otros aspectos.

A lo largo de toda la solución se mantuvieron los estándares de C#, por ejemplo: los métodos públicos utilizan PascalCase, las variables dentro de un método o los parámetros utilizan camelCase, entre otros.

Algo en lo que se hizo énfasis es en que el código pueda ser entendido y mantenido por una persona que nunca antes lo haya visto, por ende se siguieron prácticas recomendadas por Clean Code tales como: nombres de variables mnemotécnicos, código en inglés, nombres plurales para enumerados, etc.

Otra buena práctica que se siguió es la de colocar las constantes que se utilicen en la parte superior de la clase, evitando de esta forma la definición de valores que no queden claros, por ejemplo Magic Numbers.

```

public class BookingService : IBookingService
{
    private readonly IBookingRepository _repository;
    private const string InvalidDateErrorMessage = "Error, the Check-out date must be greater than the Check-in date.";

    public BookingService(IBookingRepository repository)
    {
        _repository = repository;
    }

    public Booking Add(Booking booking)
    {
        validateDates(booking.CheckInDate, booking.CheckOutDate);
        booking.Code = Guid.NewGuid().ToString();
        return _repository.Add(booking);
    }

    public IEnumerable<Booking> GetAll()
    {
        return _repository.GetAll();
    }

    public Booking Get(string bookingCode)
    {
        return _repository.Get(bookingCode);
    }

    public void Update(Booking booking)
    {
        _repository.Update(booking);
    }

    private void validateDates(DateTime checkIn, DateTime checkOut)
    {
        if (checkIn >= checkOut) {
            throw new InvalidAttributeValuesException(InvalidDateErrorMessage);
        }
    }
}

```

Figura 6.1: Código extraído de Booking Service (src/ApplicationCore/Services/BookingService).

En la figura 6.35 se puede ver un fragmento de código que evidencia la aplicación de buenas prácticas, estándares y convenciones que fueron seguidos tales cómo:

- Clases son PascalCase.
- Interfaces son PascalCase comenzando con una “I”.
- Variables privadas de una clase comienzan con guión bajo.
- Constantes son PascalCase.
- Definimos constantes para mensajes de error evitando así strings hardcodeados en el código.
- Métodos públicos utilizan PascalCase.
- Parámetros de los métodos utilizan camelCase

- Se intenta utilizar abstracciones y no clases concretas para lograr así bajo acoplamiento ej: IBookingRepository o IEnumerable.
- Se separa código en funciones auxiliares, evitando de esta forma un posible comentario y no rompiendo el SRP.
- Métodos que reciben como máximo dos parámetros (uso de monads and dyads), exceptuando los constructores.
- Métodos no tiene mas de 2 niveles de anidación.
- Nombres que revelan su intención, permitiendo evitar comentarios de “aclaraciones”. Estos nombres son pronunciables, fáciles de buscar y no requieren que quien lea el código haga mapas mentales para poder seguir la lógica.
- Nombres de clases significativos correspondientes con el dominio. Se evitó el uso de verbos.
- Métodos cortos.
- Código en inglés.
- Nombres que revelan la intención.
- Se evita utilizar nombres que referencien a la implementación ej: bookingList porque en un futuro podría no ser una lista.
- Evitar prefijos tales como strName para un string o nList para una lista de nombres.
- No se utilizaron comentarios, el código fue lo suficientemente claro y limpio como para ser entendido sin el uso de ninguno, lo que podía no quedar claro se separó a una función con un nombre descriptivo.
- Se utilizó el estilo de formateo de código de C#, dejando los “{.en la siguiente línea en la creación de Clases, métodos, etc.
- Apertura vertical entre conceptos: Se separan con líneas en blanco funcionalidades.

6.1.2. TDD

Es una metodología de desarrollo que se basa en crear primero las pruebas y luego escribir el software necesario para que estas pruebas pasen, seguido por una etapa de refactor del código implementado.

Lo seguimos realizando los test desde el centro de nuestra arquitectura hacia afuera, de esa forma siempre íbamos teniendo lo que necesitábamos para el siguiente ciclo de TDD. Comenzamos con los test para crear las entidades de negocio, una vez ese ciclo termina pasamos a los test de integración para crear el repository, de este volvíamos a los test unitarios y con ayuda de herramientas tales como Mock

realizábamos los test para el desarrollo de los servicios, por último realizábamos el controller.

Debido a una confusión tras una pregunta en el foro, no realizamos commits de cada paso del ciclo de desarrollo mientras realizábamos TDD, más tarde nuestro profesor nos aclaró que si se debía hacer para ciertas funcionalidades, así que desde ese momento en adelante fuimos haciendo commits de cada etapa.

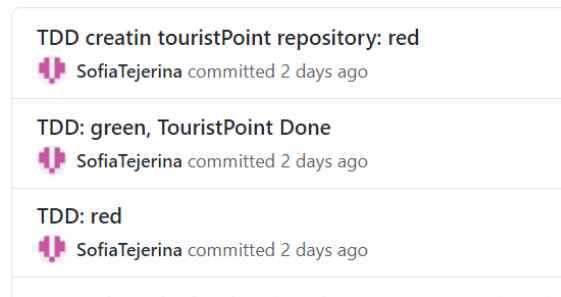


Figura 6.2: Evidencia de TDD, Punto Turistico.

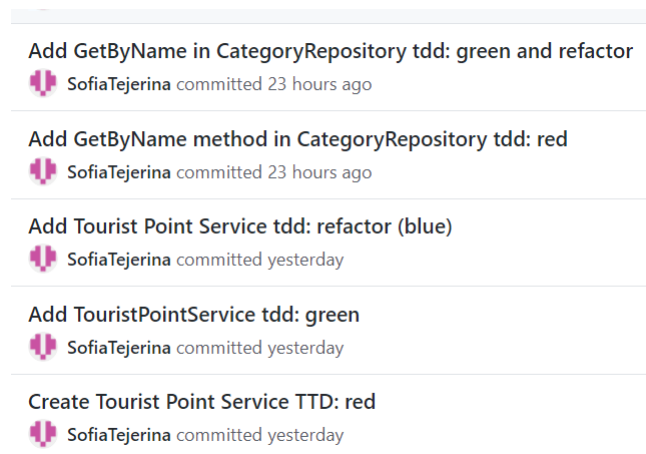


Figura 6.3: Evidencia de TDD, Servicio de Punto Turistico.

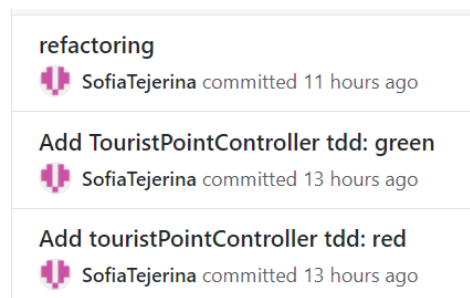


Figura 6.4: Evidencia de TDD, Controller de Punto Turistico.









TDD Red Lodging
 santiagotscanini committed 11 hours ago
TDD Green Lodging
 santiagotscanini committed 10 hours ago
Refactoring
 santiagotscanini committed 9 hours ago
Add lodging in Booking tdd: red
 santiagotscanini committed 9 hours ago
Add Lodging in booking tdd: green
 santiagotscanini committed 9 hours ago
Add Lodging repository tdd: red
 santiagotscanini committed 9 hours ago
Add Lodging Repository tdd : green
 santiagotscanini committed 8 hours ago
Add logging service tdd: red
 santiagotscanini committed 7 hours ago
Add tourist service tdd: green

Figura 6.5: Evidencia de TDD, Controller de Hospedaje.






▼  Total	97%	98/3092
>  tests	98%	27/1591
▼  src	95%	71/1501
>  Entities	100%	0/216
>  Exceptions	100%	0/9
>  ApplicationCore	99%	1/358
>  Infrastructure	99%	3/314
>  Web	90%	58/574
>  Importers	70%	9/30

Figura 6.6: COVERAGE DE LA APLICACIÓN POR USO DE TDD

6.2. Descripción de API

6.2.1. REST

Estándares

REST o Representational State Transfer es un estilo de arquitectura basada en HTTP, a la hora de realizar una comunicacion entre cliente y servidor, algunos criterios y buenas prácticas que se siguieron son:

- Usar sustantivos y no verbos.

- Plural ante singular.
- Todo en minúscula.
- Nombres concretos ante abstractos.
- No más de 2 niveles de profundidad en la API.
- Versionado en la URL.
- Hacer uso de “?” para ocultar complejidad.
- Verbos fuera de la URI ej: no utilizar v1/get-bookings.
- Uso de kebab-case en lugar de camelCase para los parámetros que se envíen¹.

Mecanismo de Autenticación Para autenticar a los administradores en ciertas funcionalidades reservadas para los mismos se creo un filter, el mismo corre antes de que la request llegue al controller y funciona de la siguiente forma:

1. El administrador solicita loggearse mediante un POST enviando por el body su email y password (nunca por query params ya que quedaría guardada su password en el navegador y cualquier persona con acceso a su navegador podría verla).
2. Se verifica que esa combinación email-password exista en la base de datos, es decir que pertenecen a un administrador existente.
 - a) En el caso de ser incorrecta se devuelve 400 (Bad Request) y un mensaje que dice que la contraseña o el mail es incorrecto, nunca se dice cual porque un posible atacante podría probar a fuerza bruta y cuando consiguiera un mail válido intenta probar con distintas contraseñas.
 - b) En el caso de ser correcto se generará un token, luego lo agregamos a un diccionario que guarda token y el e-mail de la persona que se esta loggando (esto se podría utilizar para conseguir quien es la persona de tal token, a modo de auditoría por ejemplo), luego se devuelve el token que se utilizará posteriormente en cada request validando que el administrador este loggeado.
3. Luego que se tiene el token se tendrá que enviar en todas las requests para funcionalidades reservadas, el Header tiene como key “Authorization”.
 - a) En el caso de no enviarse ningún token se devuelve 401 (Unauthorized) ya que no se encuentra autorizado a realizar dicha consulta.
 - b) En el caso de que el token no pertenezca a un administrador se devuelve 403 (Forbidden) ya que no se autenticó² con el perfil correspondiente para realizar dicha consulta.

¹Estudio que demuestra que snake.case es 20% más fácil de leer que camelCase.

²Diferencias entre Autenticar y Autorizar

- c) Si el token es correcto la request continua y sigue el funcionamiento esperado.
4. Finalmente, tenemos un método para desloguearnos, este mismo es un POST ya que si fuera un GET podríamos tener problemas ³

Cabe destacar que si se consigue un token ya generado se podrá enviar requests con este mismo (siempre y cuando se encuentre loggeado).

6.2.2. Descripción de los códigos devueltos

200 Para las solicitudes exitosas.

201 Cuando se crea un recurso correctamente.

204 Cuando se realizan modificaciones.

400 Cuando el usuario mando datos incorrectos.

401 El usuario no se encuentra autorizado a realizar la consulta.

403 El usuario no se autentico con el perfil correspondiente para realizar la consulta.

404 Cuando se quiere acceder, borrar o modificar un elemento que no existe.

500 Error en el lado del servidor.

6.3. Resources de la API

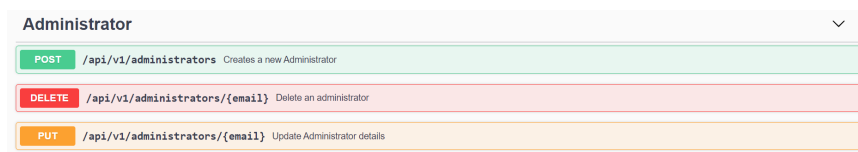


Figura 6.7: Administrator

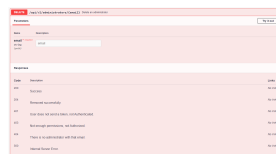


Figura 6.8: Delete Administrator

³Tweet que ejemplifica que problemas se pueden tener por el prefetching de los navegadores.

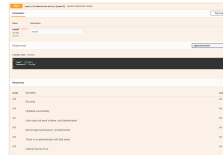


Figura 6.9: Put Administrator

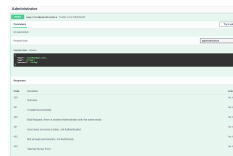


Figura 6.10: Post Administrator

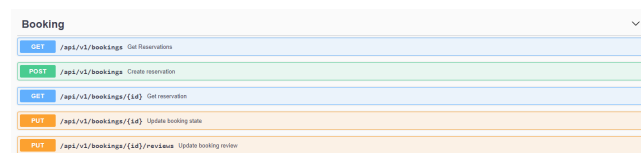


Figura 6.11: Booking

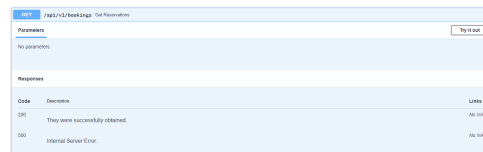


Figura 6.12: Get Booking

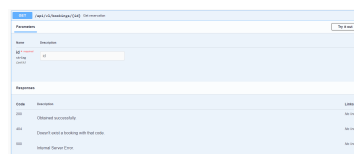


Figura 6.13: Get by id Booking



Figura 6.14: Put Booking

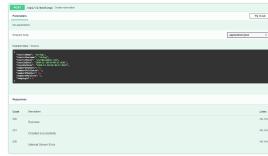


Figura 6.15: Post Booking



Figura 6.16: Put Review Booking

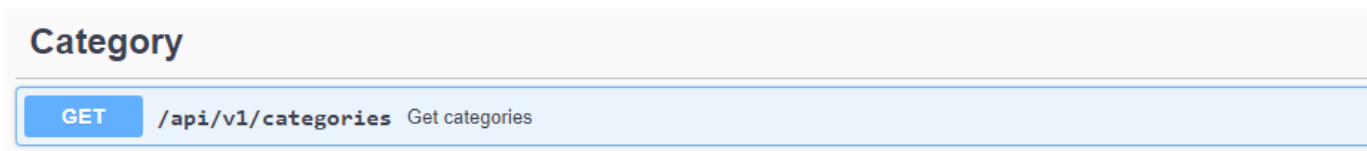


Figura 6.17: Category

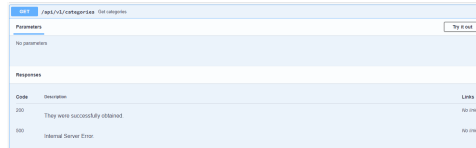


Figura 6.18: Get Category

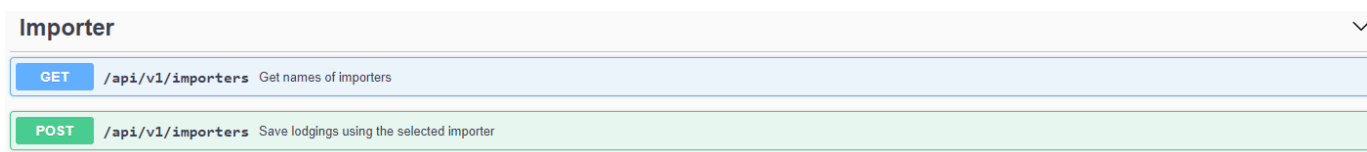


Figura 6.19: Importer

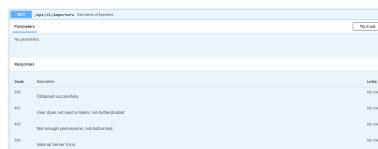


Figura 6.20: Get Importer

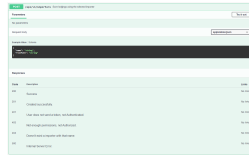


Figura 6.21: Post Importer

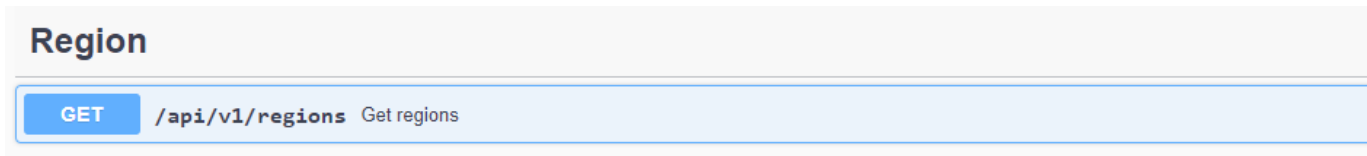


Figura 6.22: Region

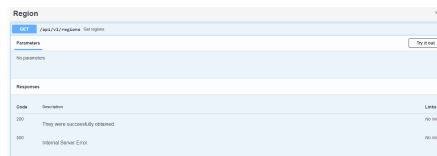


Figura 6.23: Get Region

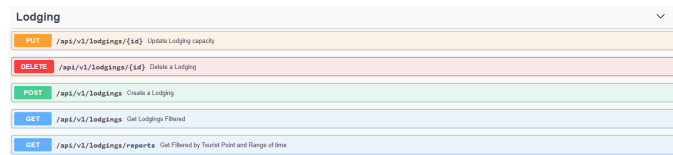


Figura 6.24: Lodging

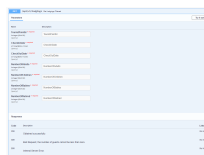


Figura 6.25: Get Lodging



Figura 6.26: Put Lodging

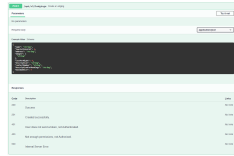


Figura 6.27: Post Lodging



Figura 6.28: Delete Lodging

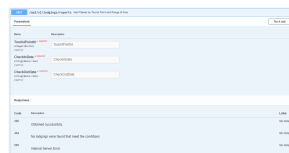


Figura 6.29: Get report Lodging

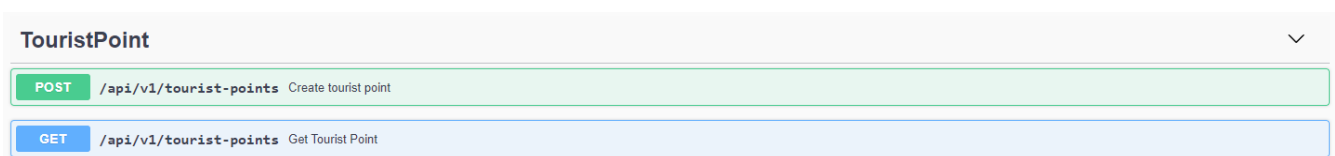


Figura 6.30: TouristPoint

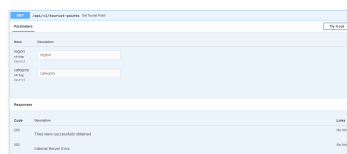


Figura 6.31: Get TouristPoint

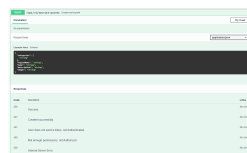


Figura 6.32: Post TouristPoint

Session	
POST	/api/v1/sessions Login
POST	/api/v1/sessions/logout Logout

Figura 6.33: Session

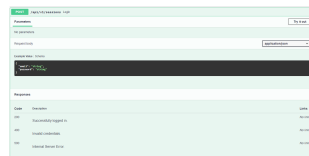


Figura 6.34: Post Login

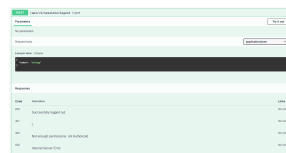


Figura 6.35: Post Logout

6.4. Implementación de importer

Para utilizar el importador que consigue Puntos Turisticos de distintos archivos, así como poder conseguir los tipos de formatos aceptados para mostrarse debemos de conseguir las ddls del proyecto que contenga el formateo, en nuestro caso implementamos uno para JSON y otro para XML. Estas ddls deben de ser colocadas en `src\Web\bin\Release\netcoreapp3.1`

Bibliografía

- [1] Robert Martin. Clean Code. [Online]. Available: <https://github.com/SaikrishnaReddy1919/MyBooks/blob/master/%5BPROGRAMMING%5D%5BClean%20Code%20by%20Robert%20C%20Martin%5D.pdf>
- [2] Paul D. Sheriff, Ken Getz. ASP.NET Developer's JumpStart. [Online]. Available: https://books.google.com.uy/books?id=2NpMDIV_BmQC&lpg=PA599&dq=btn%2C%20pnl%2C%20txt&hl=es&pg=PA600#v=onepage&q&f=false
- [3] Entity Framework Microsoft Documentation. [Online]. Available: <https://docs.microsoft.com/en-us/ef/>
- [4] Entity Framework. [Online]. Available: <https://www.entityframeworktutorial.net/>
- [5] Reflection. [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection>
- [6] Complejidad Ciclomática. [Online]. Available: https://es.wikipedia.org/wiki/Complejidad_ciclom%C3%A1tica