Universidad ORT Uruguay
Facultad de Ingeniería

Diseño de Aplicaciones 2
Obligatorio 2

Entrega requisito de la materia Diseño de aplicaciones 2

Link al repositorio en Github:
https://github.com/IngSoft-DA2/Font_Latorre_Mileo

Francisco Latorre (282414), Manuel Font(323003), Nahuel Mileo(303552)

20 de noviembre de 2025

1. Descripción del diseño

1.1. Suposiciones y decisiones de diseño

A continuación se listan las suposiciones tomadas a la hora de interpretar la letra y las respuestas del foro, como a su vez ciertas decisiones de diseño que fueron tomadas para el correcto funcionamiento de la solución:

1- Consideramos el Mantenimiento Preventivo como un tipo de incidencia, ya que su comportamiento es muy similar salvo que se especifica la fecha de inicio y de fin del mantenimiento. Además el Administrador los registra en vez del Operador

2- El mantenimiento preventivo se activa cuando es la fechaHora de inicio y se desactiva cuando es la fechaHora de fin. Mientras el mantenimiento está activo esa atracción no puede ser utilizada. El mantenimiento no se elimina para poder llevar un historial de los pasados mantenimientos.

3- Se permiten crear varios Relojes porque aunque solo se utiliza en el sistema un único DateTime FechaHora, es posible que se quiera ver el historial de cómo fueron cambiando las FechaHora del sistema para llevar un registro. El sistema solo toma en cuenta para su funcionamiento la FechaHora registrada más reciente.

4- Se decidió que una atracción puede ser parte de varios eventos, ya que podría coincidir por ejemplo Halloween con la semana del aniversario del Parque. A su vez un evento puede estar ligado a muchas atracciones a la vez.

1.2. Descripción general del diseño

Para la resolución del obligatorio se desarrolló una aplicación de gestión para un Parque Temático. Para ello se expone una Web Api que es utilizada por la interfaz web, desde donde se pueden realizar de manera amigable todas las operaciones de gestión requeridas por el cliente.

El sistema tiene 3 tipos de usuarios o roles, cada uno con limitaciones y funcionalidades distintas.

El Administrador tiene acceso a todas las operaciones CRUD (Create, Read, Update, Delete) de las Atracciones, Eventos, Recompensas, Usuarios y Mantenimientos Preventivos. Además puede escoger la estrategia de puntuación, ver un reporte de uso de atracciones, ver el ranking diario y modificar la fecha/hora actual del sistema.

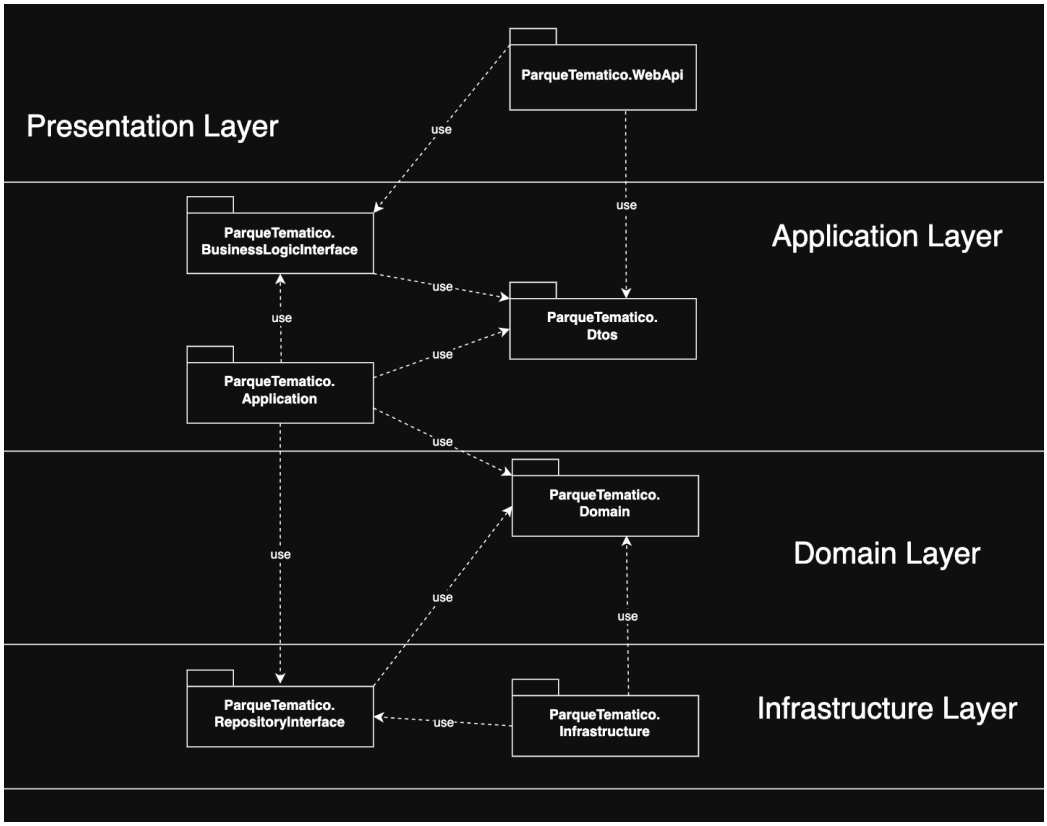
El operador tiene acceso a todas las operaciones CRUD de Incidencias. Además puede crear un egreso (cuando un visitante se va de una atracción), visualizar el aforo de una atracción y ver su capacidad restante.

Por último, los visitantes son capaces de crear su propio perfil, sin necesidad de un Administrador u Operador. Además es capaz de comprar Tickets para atracciones para lo cual debe indicar para qué fecha se compra el ticket. Puede también canjear Recompensas si cumple con los requisitos para ello y ver su historial de puntuación.

Para implementar estos requerimientos la solución contiene un número de *assemblies* y cada uno de ellos implementa un paquete lógico, con el objetivo de que al modificar el código fuente se minimice el impacto del cambio en los componentes físicos de la solución. Es por esto que distintos paquetes interactúan entre sí mediante interfaces expuestas y no utilizando implementaciones concretas. A lo largo de la documentación se detallarán las responsabilidades y dependencias de cada paquete.

1.3. Diagrama general de paquetes

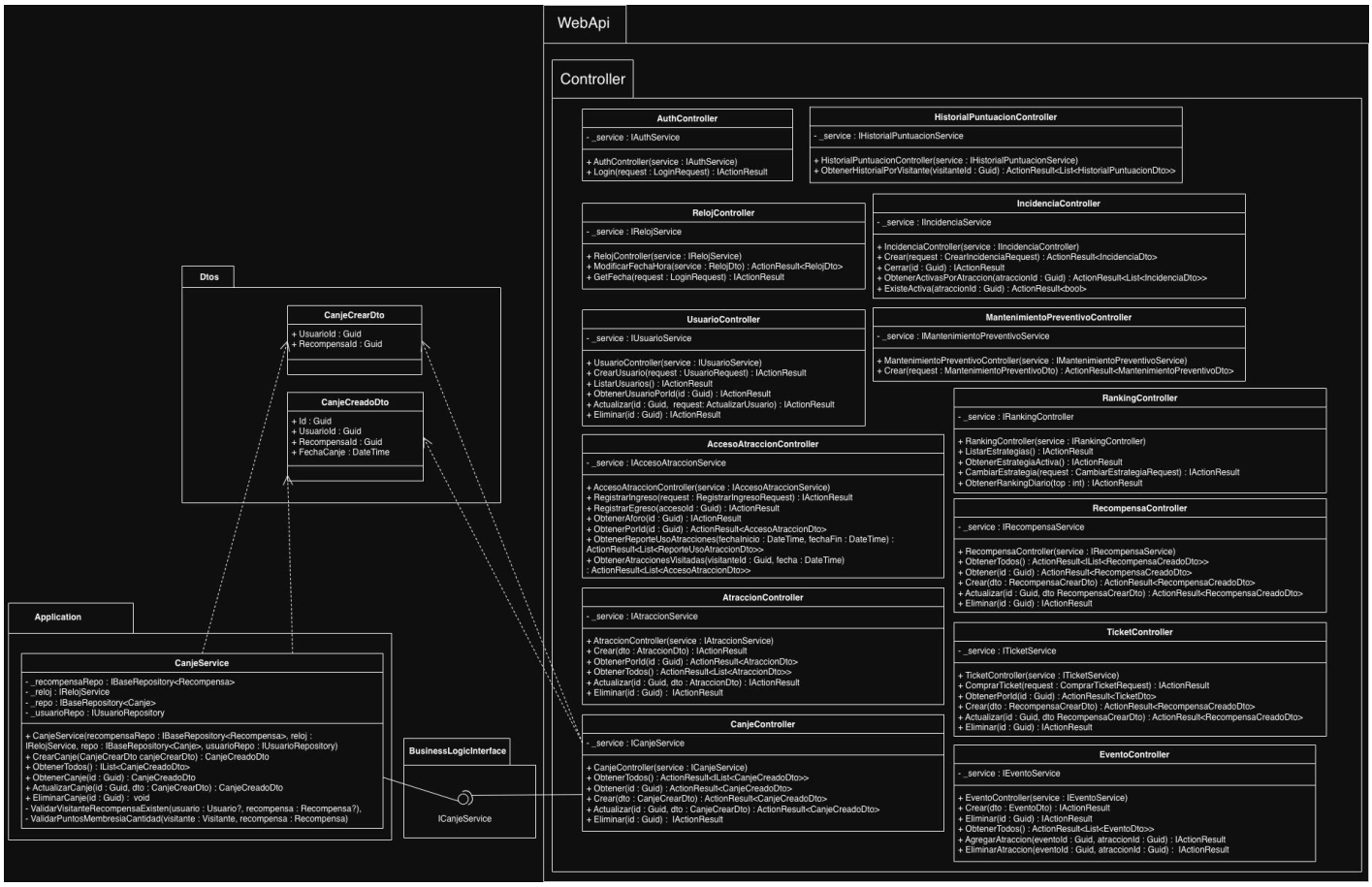
Para este diagrama se ignoró el paquete Test para facilitar la visualización del mismo, ya que este paquete depende de todos los paquetes del backend y diagramar todas esas dependencias únicamente dificultaría el entendimiento del diagrama. Se escogió dividir la solución en estos paquetes para que cada paquete tenga una responsabilidad única. Además se crearon paquetes de interfaz, para segregar paquetes y aumentar la mantenibilidad, cumpliendo con DIP y ISP.



1.4. Paquete WebApi

Define los endpoints y maneja las requests y las responses. Dentro de este paquete se hallan tres carpetas: Controllers, Filtros y Converters. Los controllers definen todos los endpoints de la WebApi y utilizan la interfaz de servicio para procesar el Dto del request y retornar el Dto response adecuado. Los filtros procesan la request antes que el controller para asegurar que esta sea válida. Hay filtros para "atrapar" excepciones y para asegurar que el usuario que hace la request esté autenticado. Por último los converters modifican los formatos de los DateTimes para que correspondan con el formato solicitado por la letra.

Para representar este paquete se utilizó un diagrama de clases que muestra la estructura interna del paquete y las relaciones relevantes.
Se incluyen todos los controllers y, además, se muestra en detalle cómo el CanjeController se relaciona con los Dtos de Canje y con la interfaz ICanjeService.
No se detalla las dependencias de cada controller individualmente, ya que todos presentan el mismo patrón de dependencias; con este ejemplo es suficiente para visualizar cómo se relacionan las clases.



Criterios REST

Nuestra aplicación cliente-servidor busca desacoplar el cliente del servidor con el fin de que escalen de manera independiente. Se corre cierto código del lado del cliente, donde se hacen ciertas validaciones y luego al enviar la solicitud se procesa del lado del servidor y le llega al cliente el status code adecuado.

El servidor maneja las responses y requests de forma RESTful. Las solicitudes son stateless lo que quiere decir que ninguna solicitud depende de otra. La ejecución de una solicitud no depende del historial de solicitudes previas del cliente, lo cual mejora la escalabilidad. Gracias a la restricción de Sistema de Capas, el cliente no tiene que saber si está conectado directamente al servidor final o si existen intermediarios (proxies, load balancers, etc.) en la comunicación. Esto permite añadir capas de seguridad o rendimiento (caché) sin impactar al cliente.

Uno de los aspectos más importantes de REST es su Interfaz Uniforme. Los recursos se identifican de manera consistente mediante el formato URI, como por ejemplo: http://api.ejemplo.com/api/users/231. La operación a ejecutar sobre este recurso se define utilizando el Método HTTP (GET, POST, PUT, DELETE) apropiado.

Por último en el body se incluirán los datos necesarios para ejecutar la operación si es que son requeridos, por ejemplo en un POST o PUT y en el Header se deberá colocar la autenticación del usuario que ejecuta la operación. También se deben incluir metadatos esenciales, como el tipo de contenido enviado Content-Type: application/json) para que el servidor sepa cómo procesar los datos del Body.

La seguridad de la operación se garantiza incluyendo el token de autenticación del usuario en el encabezado Authorization. En nuestro caso se utiliza el esquema Bearer Token, con el siguiente formato: Authorization: Bearer [Token JWT]. Se optó por utilizar Bearer Token JWT porque cumple con Stateless, es muy seguro y de alto rendimiento. Además utiliza un formato basado en JSON, que es ampliamente legible y soportado por prácticamente todos los lenguajes de programación y plataformas (frontend y backend).

EndPoint
Los endpoints representan los recursos expuestos por la WebApi. Para cada recurso se define su URL y los métodos HTTP permitidos, siguiendo las convenciones REST. Para cada endpoint se documentan: URL, método HTTP, parámetros (path y query), headers relevantes, body de la request cuando aplica, y los posibles status codes de respuesta. Para no extendernos en la descripción se incluyeron todos los endpoints en el anexo.

1.5. Domain

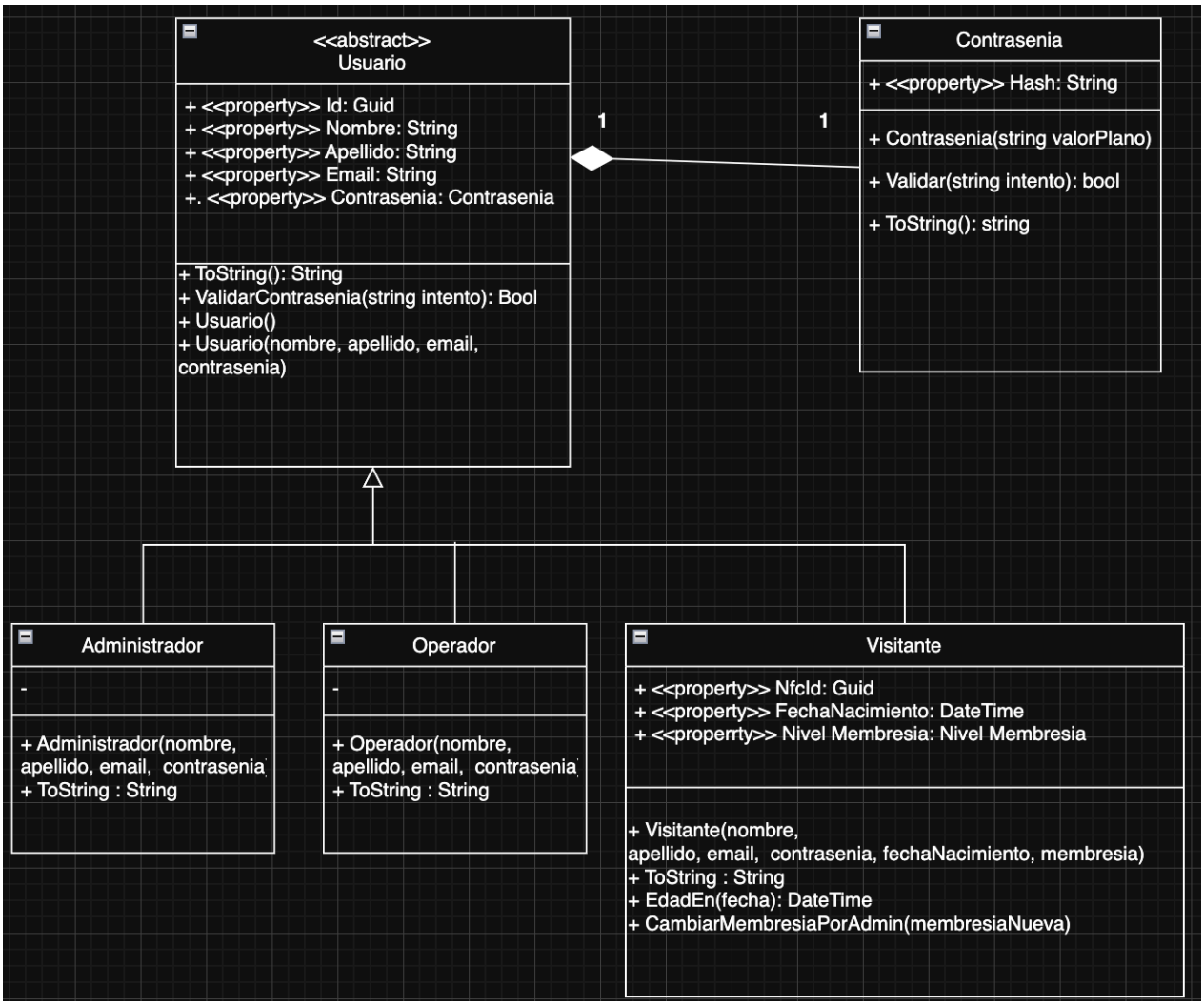
Se buscó seguir para esta solución un Domain-Driven Design, con el cual se tiene un enfoque de diseño que organiza la arquitectura alrededor del dominio de negocio y su lógica y no alrededor de la base de datos los frameworks o la tecnología. Este paquete contiene las entidades y value objects que se detallan en la letra, además de excepciones creadas para lanzarse cuando se incumple una regla del dominio al intentar crear o modificar una entidad.

Usuario
La clase Usuario y sus herencias son un aspecto muy importante del dominio, ya que todas las operaciones que se hagan en el sistema van a estar atadas a un usuario que las

ejecuta. Se optó por hacer 3 clases distintas que hereden de Usuario para dividir las operaciones que puede hacer cada uno.

Cada clase utiliza el método override ToString() que indica cual es el tipo del usuario. De esta forma el UsuarioService puede saber fácilmente qué tipo de usuario es obteniendolo de este string y este mismo string será utilizado por AuthService para generar un Claim tipoUsuario, el cual será utilizado por el filtro TipoUsuarioFilter para saber si el usuario es del tipo que corresponde y puede realizar la operación.

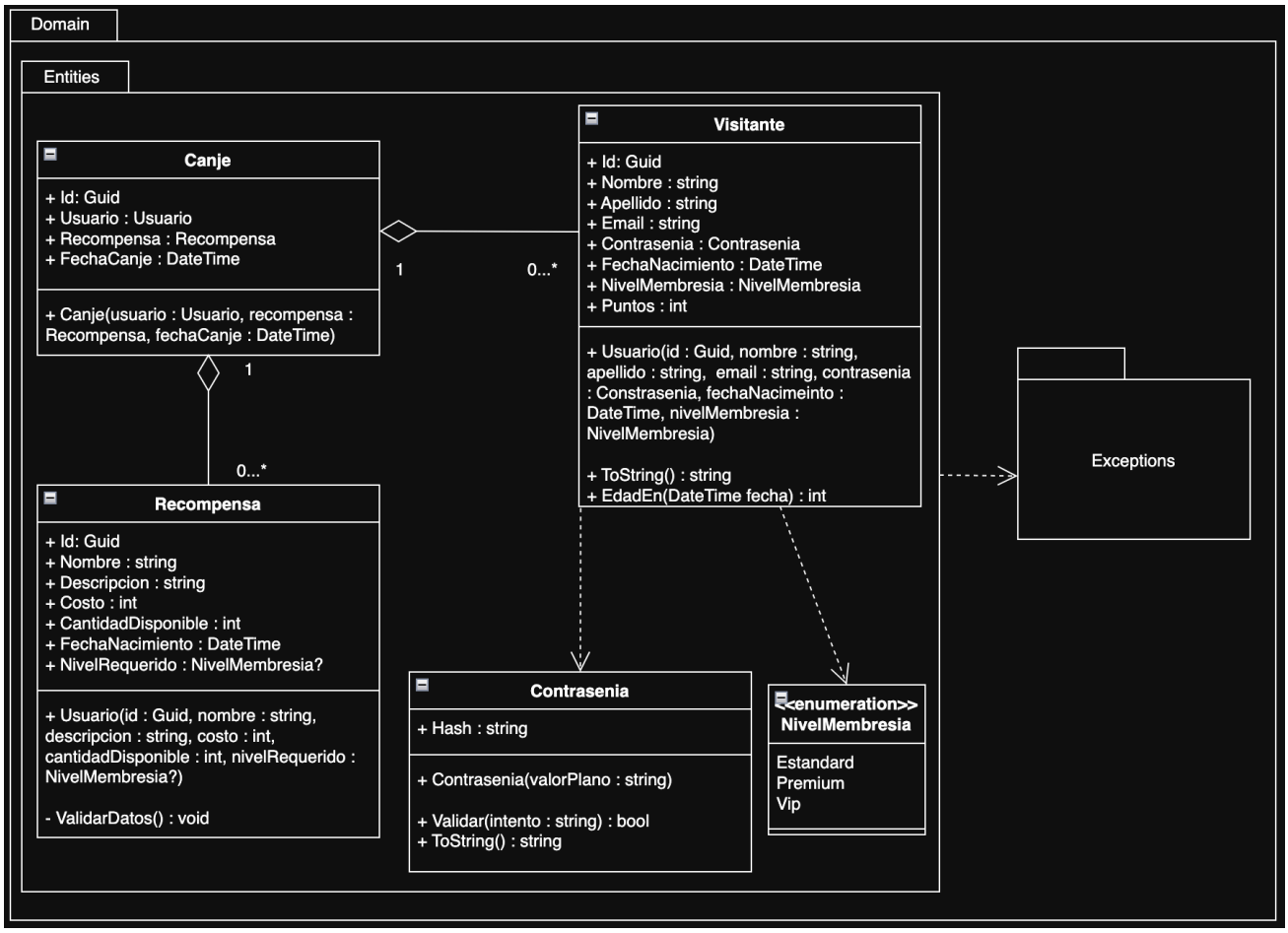
Esto aplica el principio Open/Closed ya que los servicios no necesitan ser modificados si creamos un nuevo rol o tipo de usuario. Cada usuario sabe cómo comportarse dependiendo de qué tipo es y los servicios saben manejarlo. Como también aplicamos el principio de Polimorfismo por el cual utilizamos herencia para ahorrar muchos if y switch que manejarían distintos casos según el tipo de usuario.



Se cumple el Principio de Sustitución de Liskov porque cualquier clase hija de Usuario puede utilizarse en lugar de Usuario sin alterar el comportamiento esperado. Las clases derivadas respetan el contrato definido por el padre, y no introducen restricciones adicionales que rompan su uso.

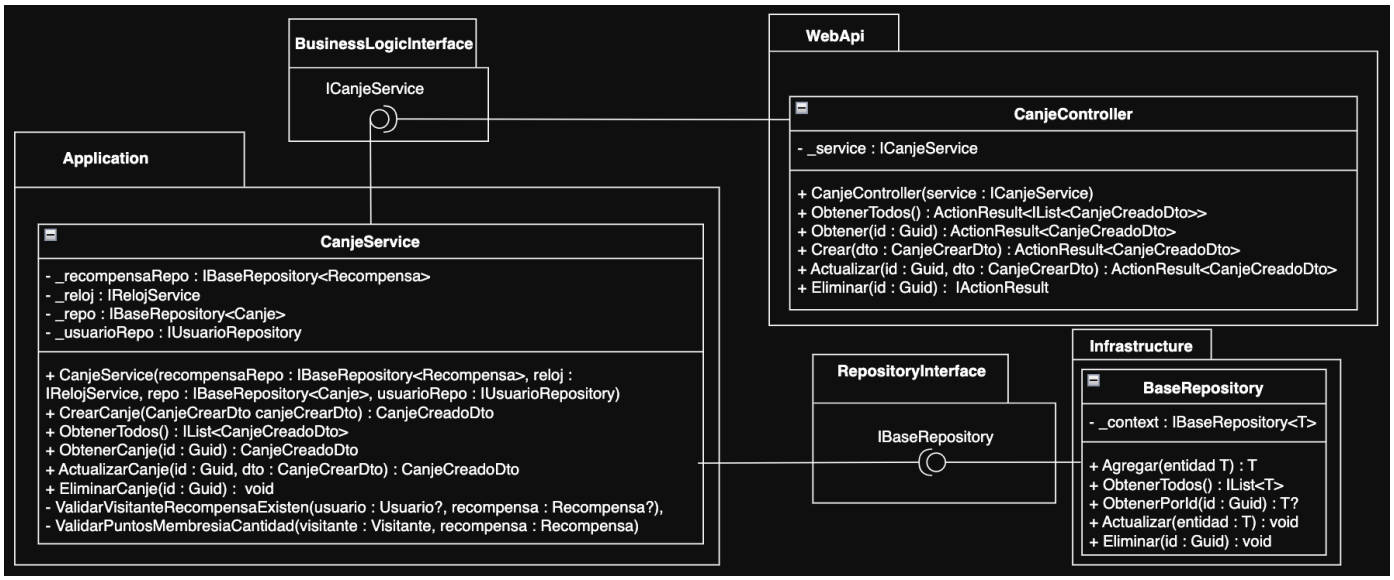
Canje de recompensas

Se optó por mostrar del dominio la inclusión de las clases nuevas Recompensa y Canje. Estas clases fueron creadas para cumplir con el requerimiento del canje de recompensas por parte del visitante. A su vez se incluyó en el diagrama la carpeta exceptions ya que todas las clases del dominio utilizan excepciones personalizadas. Consideramos que no aporta valor adicionar representar la relación de cada clase con su clase exception, se sobreentiende que UserException es utilizada por User, etc.



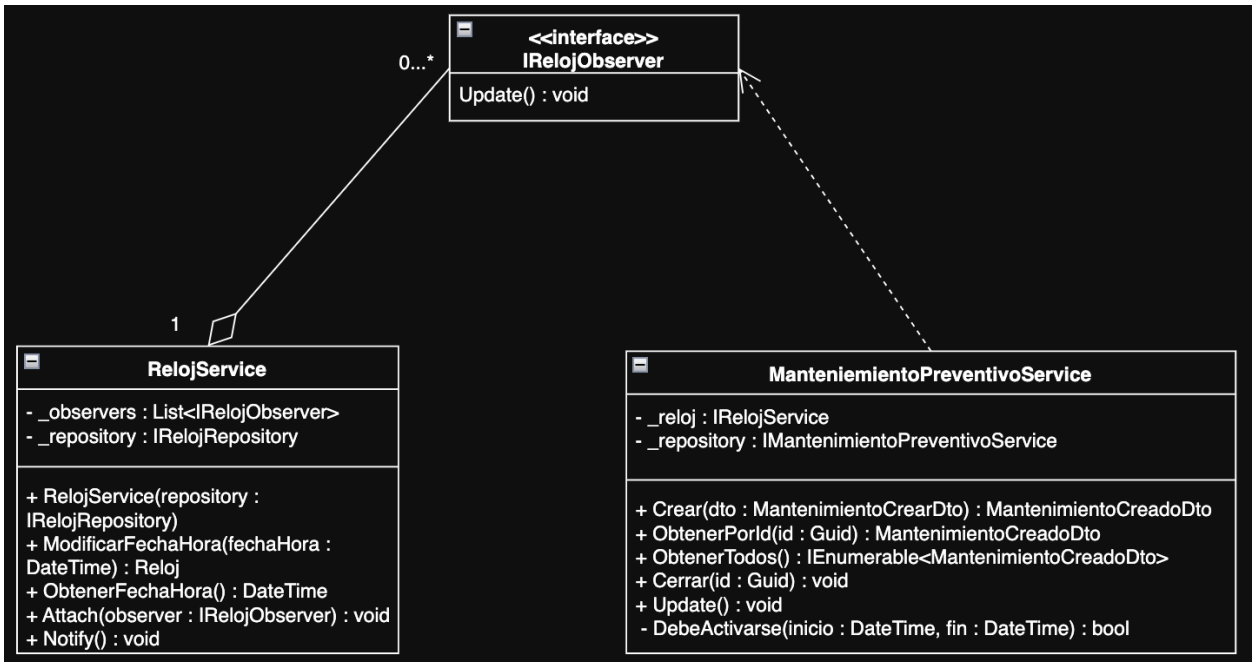
1.6. Application

Contiene principalmente los servicios, los cuales operan sobre los repositorios y exponen los datos que tienen guardados mediante el recibimiento y retorno de Dtos. A continuación se puede ver un ejemplo de cómo un servicio utiliza un repositorio mediante interfaces y expone las operaciones para que las pueda utilizar la WebApi. El motivo principal de utilizar interfaces para los servicios y los repositorios es disminuir el acoplamiento, cumplir con DIP y cumplir con OCP haciendo la arquitectura más flexible y fácil de modificar o cambiar de tecnología en un futuro. Además podemos hacer Unit Testing creando doubles de las interfaces para no tener que crear objetos reales en todos los Test. Si no usamos doubles el fallo en un objeto puede conducir al fallo en la clase que estamos testeando, pero en Unit Testing deberíamos estar testeando únicamente una cosa a la vez. Se optó por mostrar un ejemplo concreto en vez de representar todas las clases de Application para facilitar la visualización del diagrama. De este ejemplo se puede deducir el comportamiento del resto de servicios.



Dentro de Application se utilizó el patrón comportamental Observer que se utiliza para definir una dependencia uno a muchos entre objetos, de modo que cuando un objeto (sujeto) cambia de estado, todas sus dependencias (observadores) son notificadas y actualizadas automáticamente. Este patrón disminuye el acoplamiento y aumenta la cohesión, cumpliendo con SRP ya que el RelojService sólo conoce la interfaz genérica del observador (IObserver) y no necesita saber quiénes dependen de él ni qué hacen cuando son notificados, haciendo que cada uno se concentre en su propia responsabilidad. Además el código es reutilizable ya que los observadores son independientes, se pueden añadir o eliminar nuevos observadores al sujeto en tiempo de ejecución sin necesidad de modificar el código del RelojService.

De esta forma además de cumplir SRP, cumplimos con OCP porque se pueden incluir nuevos observers simplemente haciendo que hereden de IRelojObserver. También se cumple DIP, porque RelojService y MantenimientoPreventivoService no dependen de ninguna clase concreta adicional, sino de una interfaz.



BusinessLogicInterface
Aquí se hallan las interfaces de todos los servicios, las cuales son implementadas por los servicios y utilizadas por la WebApi para cumplir con Dependency Inversion Principle.

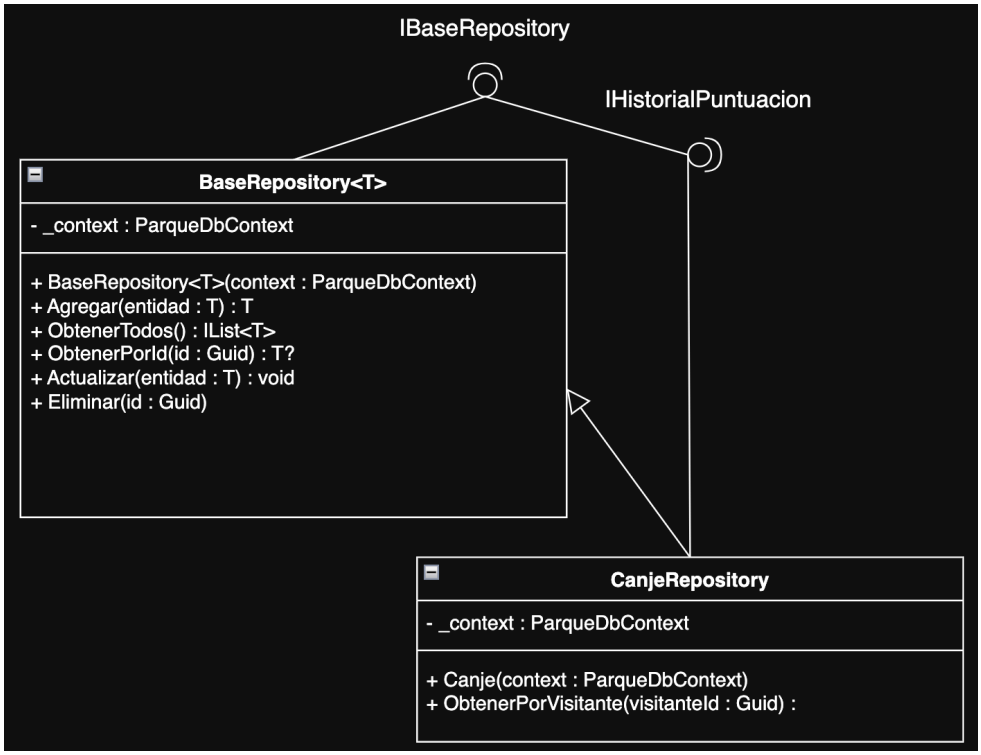
1.7. Infrastructure

Tiene todos los repositorios mediante los cuales se guardan las entidades reales en la base de datos utilizando .Net. También se encuentra el contexto de la base de datos, su configuración y las migraciones. Se buscó que en los repositorios no haya lógica, sino que esta sea sea manejada por los servicios. Los repositorios tienen como única responsabilidad el CRUD básico de las entidades.

Para este paquete se decidió mostrar la implementación de BaseRepository. Este repositorio fue creado principalmente con la mantenibilidad en mente, encapsular las operaciones más comunes de los repositorios, en vez de tener lógica muy similar en todos los repositorios. Ahora se cumple el principio OCP y se abstraen estas operaciones en vez de repetirlas en varias clases (DRY: Don't Repeat Yourself).

Todas las interfaces de repositorios deben implementar IBaseRepository y todos los repositorios deben heredar de BaseRepository. Cuando se requiere agregar más funcionalidades simplemente se crea una nueva interfaz con los métodos adicionales y el

nuevo repositorio implementará todos los métodos de BaseRepository más los métodos de su interfaz propia.



Modelado de la base de datos

Se utilizó .Net con metodología Code First para generar la base de datos SQL configurada en ParqueTematicoContext. Al final de la sección se puede apreciar un diagrama con todas las tablas y las relaciones entre sí. Cabe destacar algunas desiciones de diseño tomadas para ciertas tablas.

Comenzando por la tabla usuario, se decidió utilizar una única tabla para los tres tipos de usuarios, ya que los tres heredan de la clase Usuario. Esto es un enfoque TPH (Table Per Heritage). Para las columnas que son exclusivas de por ejemplo visitante, sus valores permanecerán en Null cuando el usuario no sea de tipo visitante.

Este enfoque TPH fue el que prevaleció en nuestra solución porque cuando son pocas las columnas que pueden ser Null este enfoque tiene un rendimiento más alto. Siguiendo esta misma lógica se le agregó a la tabla Incidencias las columnas fechaInicio y fechaFin, para que MantenimientoPreventivo el cual hereda de Incidencia pueda existir en la misma tabla, simplemente utilizando estas dos columnas nuevas.

Es por este enfoque TPH que es importante en la tabla usuario la columna TipoUsuario y en Incidencia la columna Discriminator. Estas columnas distinguen fácilmente con qué tipo de entidad se está tratando.

Cómo se puede observar .Net generó una tabla EventoAtracciones, ya que existe una relación many-to-many no explícita entre Eventos y Atracciones. Es decir que muchas atracciones pueden ser parte del mismo evento y una atracción puede tener varios eventos. Al no tener Atracción una propiedad Evento en nuestro dominio .Net asume una relación many-to-many y crea esta tabla relacional.

A diferencia del caso anterior las tablas Canje y AccesoAtraccion son tablas explícitas creadas por nosotros. AccesoAtracción define una relación many-to-one y Canje define una relación many-to-many. Estas tablas las creamos explícitamente para sumarle columnas necesarias a estas relaciones. Para AccesoAtracción se necesitaba saber los PuntosObtenidos, las Fechas, etc. y para Canje se requería saber la FechaCanje. Estas columnas eran necesarias y por lo tanto se configuraron estas tablas explícitamente.

RepositoryInterface

Estas son las interfaces de todos los repositorios. Los servicios las utilizan para cumplir con Dependency Inversion Principle y no conocer a los repositorios concretos reales.

Dtos

Aquí se encuentran los Dtos, los cuales tienen cómo rol transferir datos primitivos entre capas del sistema, es decir que no transfieren entidades del dominio. Esto es útil para que la WebApi no conozca el dominio y se comunique con las InterfaceServices mediante estos objetos que encapsulan solo los datos necesarios.

1.8. Paquete Test

En este paquete hay una carpeta por cada otro paquete de la solución y dentro de la carpeta se encuentran las clases de MSTest que testean una clase en concreto. Se utilizó el framework moq para realizar pruebas unitarias simulando comportamientos con Mocks. Se respetó TDD el cuál se va a evidenciar a continuación y se siguió inside-out.

Inside-out es una forma de hacer TDD por la cual se comienza asegurando (con la creación de tests) el correcto funcionamiento de las capas internas de la solución, cómo lo es el dominio y de a poco se van programando los tests hacia afuera, cómo los tests de repositorios y WebApi. De esta forma se asegura el control de los bloques fundacionales de nuestra solución, es decir el código del que dependen los demás paquetes.

Se utilizó [TestInitialize] y se siguió una convención para nombrar los [TestMethod] para simplificar el entendimiento de las pruebas. La convención utilizada es NombreDelMétodoATestear_Condiciones_ComportamientoEsperado.

```
[TestMethod]
public void GetById_IdNoExiste_RetornaNull()
{
```

Cobertura

Se buscó alcanzar la máxima cobertura posible en los tests, aunque no siempre es posible alcanzar el 100%. Vale aclarar que el Frontend no se testea y que el paquete Test no se testea porque no se testean los tests por lo que ignoramos estos paquetes.

La cobertura de infraestructura parece baja, pero en verdad lo que sucede es que hay muchas líneas que son de configuración y de migraciones. Si entramos podemos ver la

cobertura real de los repositorios.

Lo mismo sucede con WebApi en donde si ignoramos las líneas de configuraciones y otros archivos la cobertura aumenta.

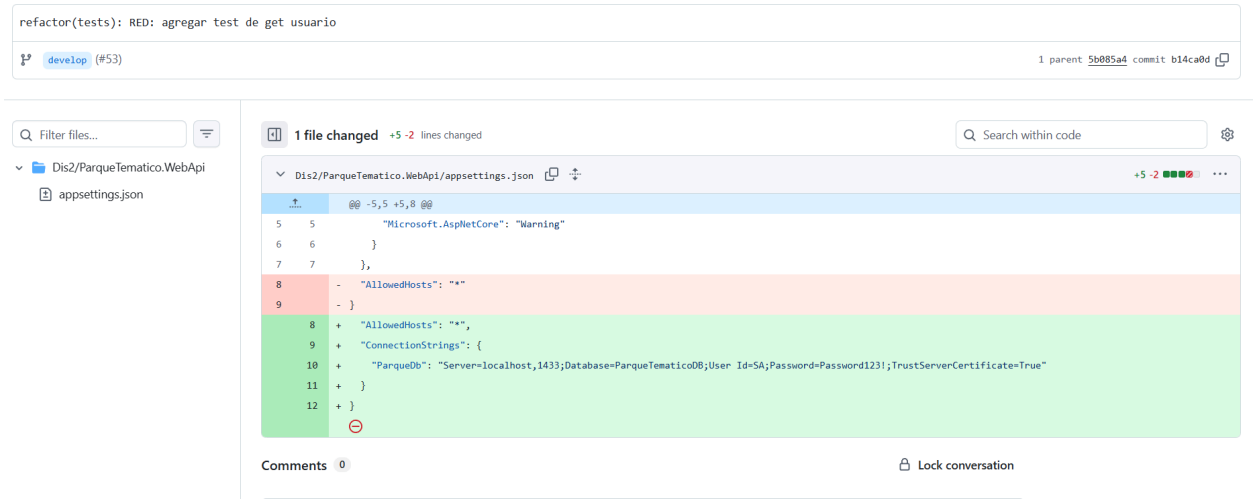
No se testeó tampoco program.cs y ParqueTematicoContext no se testeó porque sus líneas de código siempre están siendo cubiertas al ejecutar los métodos de los repositorios.

TDD

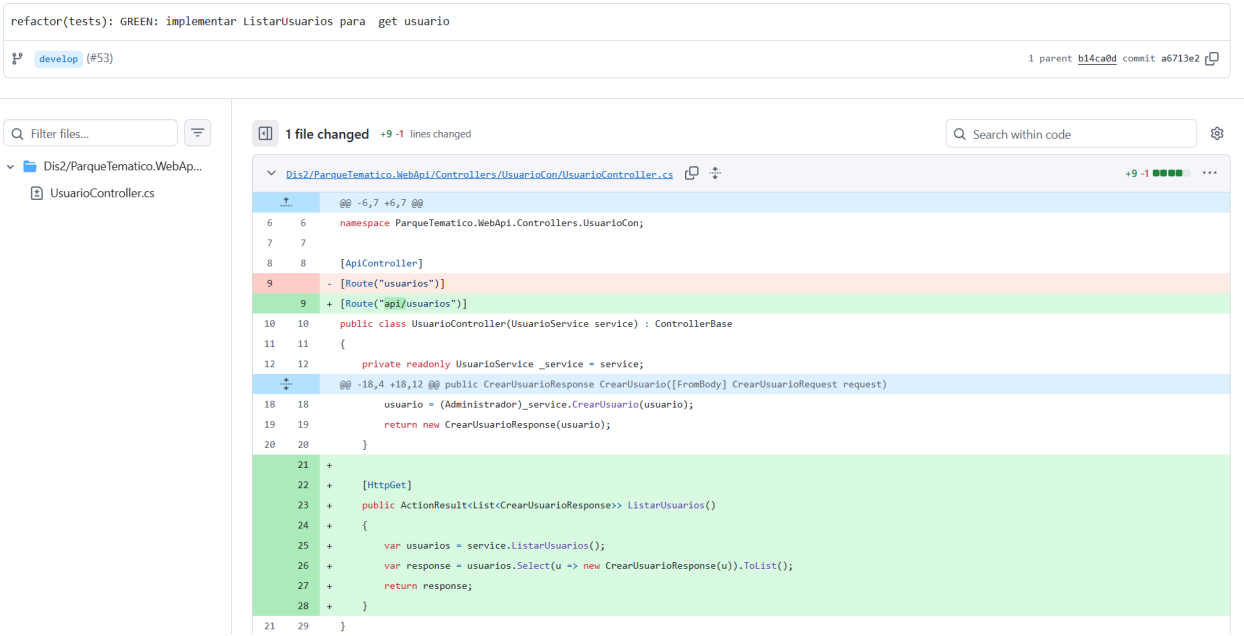
En el proyecto aplicamos TDD siguiendo el ciclo Red-Green-Refactor. Siempre se escribieron las pruebas antes de implementar la funcionalidad, también para asegurarnos que no pasen y que las funcionalidades eran lo que las hacían pasar, hacer esto sigue la filosofía TDD de que las pruebas van antes que el código y que las pruebas funcionan cómo contrato sobre lo el código debe cumplir, poder hacer y las reglas que se deben cumplir.

A continuación evidenciamos algunos commits con el ciclo Red-Green-Refactor.

Red: Agregar test de get usuario



Green: Implementar ListaUsuarios para get usuario



F.I.R.S.T

Se consiguió que los tests sean rápidos (Fast), pudiéndose ejecutar en su totalidad en menos de un minuto. Los tests son independientes (Independent), esto se logra utilizando doubles para simular el comportamiento adecuado de ciertos objetos y asegurar que cuando falla un test el fallo está en las líneas de código que testea, esto facilita encontrar donde está el error e ir a solucionarlo. Los test son repetibles (Respetable) ya que pueden ejecutarse todas las veces que queramos y siempre darán el mismo resultado. Los test son Self-Validating, es decir que el test o falla y muestra rojo o pasa y muestra verde. De esta forma nos damos cuenta al instante si falló o no, no es necesario interpretar un mensaje de log ni nada por el estilo. Además deben ser Timely, o sea cumplir TDD y escribir los tests antes que el código.

Arrange-Act-Assert

Se siguió la secuencia Arrange-Act-Assert para escribir los tests. Primero se prepara el caso que queremos testear por lo que creamos los objetos, ponemos los valores que queremos probar y hacemos SetUps en los Mocks de la librería Moq. Luego ejecutamos la operación que queremos testear realmente. Por último utilizamos los métodos Assert de MsTest que serán true o false. A continuación se ejemplifica. (Los comentarios son utilizados solo en esta imágen, no se utilizan comentarios innecesarios en el código)

```
[TestMethod]
public void InicializarAdministrador_YaExiste_NoSeInicializa()
{
    //Arrange
    var service = new UsuarioService(_repository);

    //Act
    service.InicializarAdministrador();
    service.InicializarAdministrador();

    //Assert
    Assert.AreEqual(1, _repository.ObtenerTodos().Count);
}
```

Doubles

Los test doubles son objetos que reemplazan a otros a los efectos de realizar pruebas y simular su comportamiento. Se utilizan doubles para testear unitariamente solo las clases que queremos y los objetos de las clases de las cuales depende pueden ser simulados.

Principalmente se utilizaron en nuestra solución mocks, los cuales son preprogramados para cumplir con expectativas que tenemos sobre ellos. Estos pueden tirar excepciones si reciben llamadas inesperadas y se puede chequear si fueron llamados la cantidad de veces que esperábamos.

En el siguiente ejemplo se utiliza Dummy ya que simplemente se utiliza para satisfacer los requisitos de un constructor o método, pero su estado o comportamiento no son relevantes para el test específico que se está ejecutando en este momento, porque no se espera que se llame a ninguno de sus métodos.

```
private CanjeController _controller = null!;
private Mock<ICanjeService> _service = null!;

[TestInitialize]
public void SetUp()
{
    _service = new Mock<ICanjeService>();
    _controller = new CanjeController(_service.Object);
}
```

En el siguiente ejemplo se utiliza un Stub ya que se le define al objeto un valor determinado, pero no nos interesa verificar cómo se está llamando a ese objeto.

```
[TestMethod]
public void Obtener_RetornaRecurso()
{
    var dto = new CanjeCreadoDto
    {
        Id = Guid.NewGuid(),
        UsuarioId = Guid.NewGuid(),
        RecompensaId = Guid.NewGuid(),
        FechaCanje = DateTime.UtcNow
    };
    _service.Setup(s :ICanjeService => s.ObtenerCanje(dto.Id)).Returns(dto);

    ActionResult<CanjeCreadoDto> result = _controller.Obtener(dto.Id);

    var okResult = result.Result as OkObjectResult;
    Assert.IsNotNull(okResult);
    Assert.AreEqual(200, okResult.StatusCode);
    Assert.AreEqual(dto, okResult.Value);
}
```

En el siguiente ejemplo se utiliza un Mock ya que el objeto no solo retorna el valor predefinido que precisamos, sino que además se utiliza Verify para asegurar que tienen todas las llamadas que se esperaban.

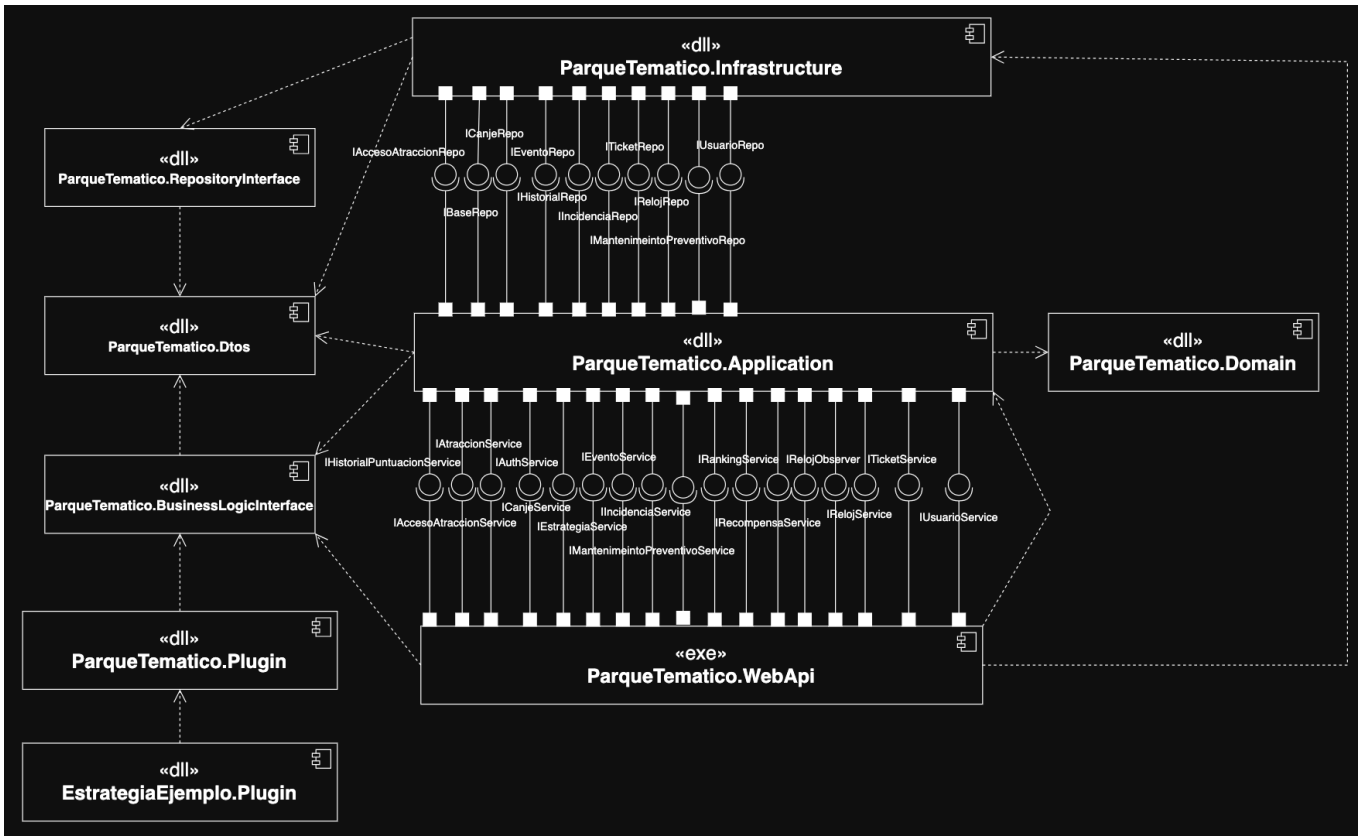
```
[TestMethod]
public void Eliminar_DeberiaRetornarNoContent()
{
    var id = Guid.NewGuid();
    _service.Setup(s :ICanjeService => s.EliminarCanje(id));

    IActionResult result = _controller.Eliminar(id);

    var noContent = result as NoContentResult;
    Assert.IsNotNull(noContent);
    Assert.AreEqual(204, noContent.StatusCode);
    _service.Verify(s :ICanjeService => s.EliminarCanje(id), Times.Once);
}
```

1.9. Vista de Diseño (Lógica)

Para representar esta vista se fueron mostrando varios diagramas de clases y de paquetes a lo largo del documento. Además se definieron las capas de la solución y se describieron las responsabilidades de las clases, paquetes e interfaces. También se especificó el uso de patrones de diseño y su razón de ser. Teniendo esto en cuenta no es útil repetir información o volver a mostrar diagramas.

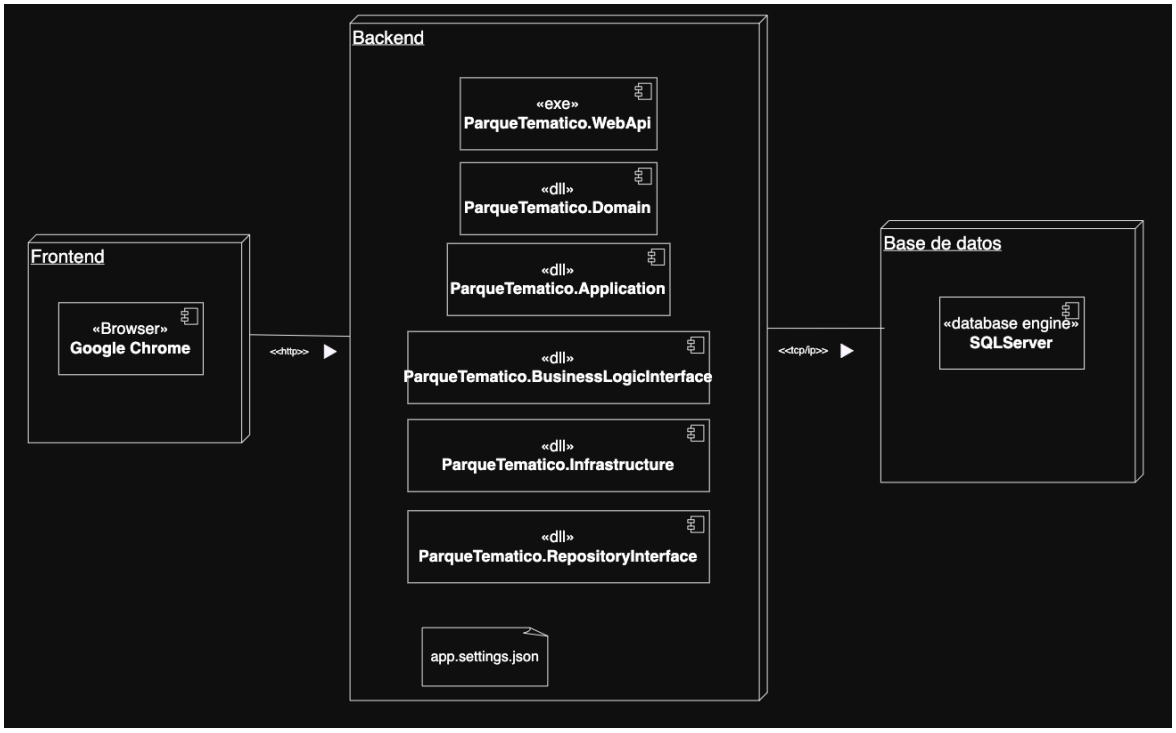
1.10. Vista de Componentes

Esta vista muestra la organización estática del código fuente en el ambiente de desarrollo. Demuestra la estructura de los archivos, carpetas, librerías y componentes del software tal como son manejados.



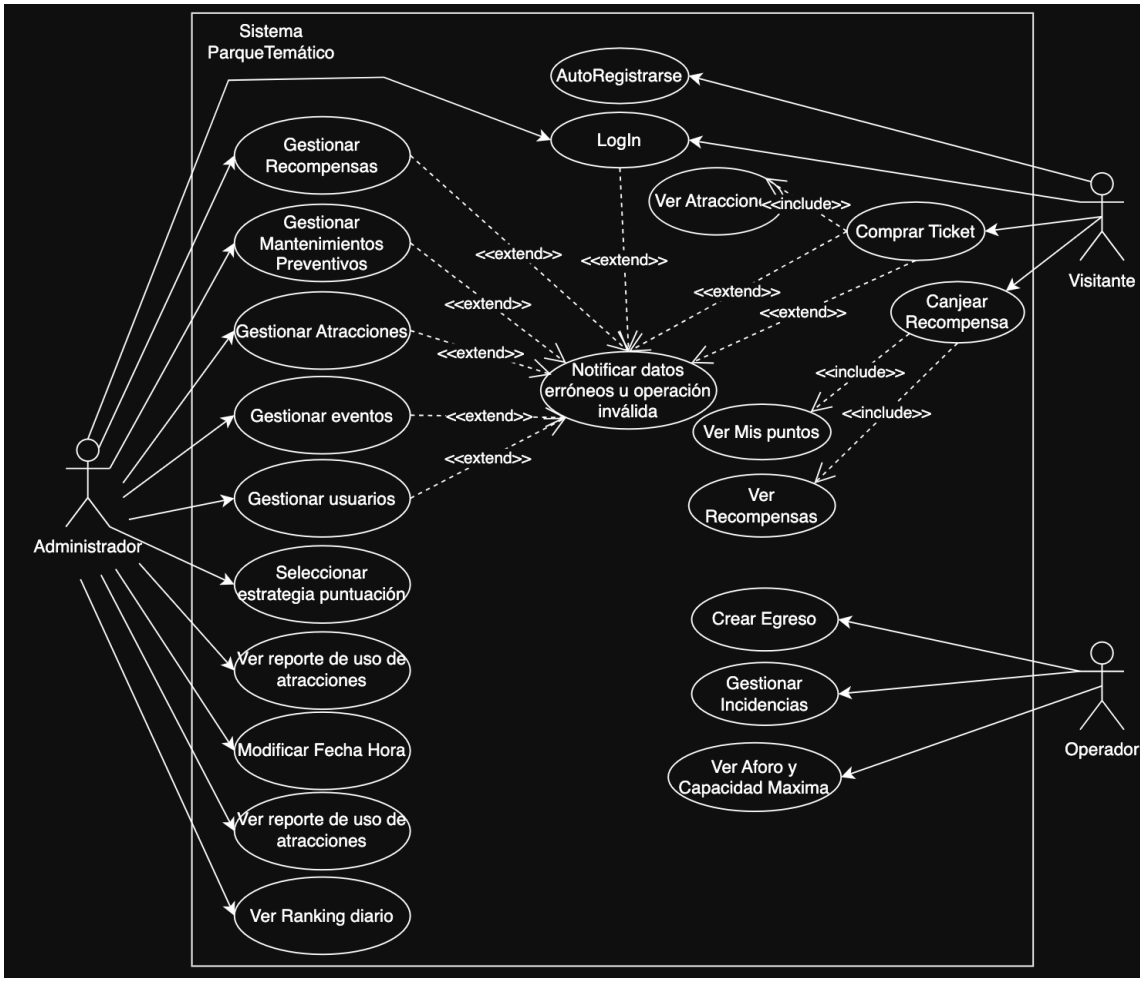
1.11. Vista de Despliegue

Esta vista se centra en la distribución física del software en el hardware. Muestra dónde residen los componentes de software (de la Vista de Componentes) en los nodos de hardware (servidores, bases de datos, dispositivos del cliente).



1.12. Vista Casos de Uso

Para la vista casos de uso de UML 4+1 realizamos un diagrama casos de uso en donde se pueden apreciar las distintas operaciones que puede realizar cada uno de los tipos de usuarios. Se utilizaron las etiquetas adecuadas <<extend>> y <<include>> para relacionar unos casos con otros.



1.13. Reflection

Incorporación de Nuevas Estrategias de Puntaje mediante Plugins (DLL)
El sistema incorpora un mecanismo de extensibilidad diseñado para permitir que terceros puedan agregar nuevas estrategias de puntaje de forma dinámica, sin necesidad de recompilar la aplicación ni modificar su código fuente. Este mecanismo está basado en la utilización de Reflection en C# para cargar clases externas contenidas en archivos DLL ubicados dentro de una carpeta especial de plugins.

El sistema fue diseñado bajo el concepto de plugin architecture aplicando principios SOLID especialmente el OCP, de forma que la lógica de cálculo de puntaje pueda extenderse sin modificar las clases internas.

Para ello, la aplicación define una interfaz pública y estable, llamada IPuntuacion. Cualquier desarrollador que desee crear una nueva estrategia debe simplemente: Crear una clase que implemente esta interfaz, compilar dicha clase dentro de una biblioteca de clases .dll, copiar el archivo dll a la carpeta Plugins y reiniciar la aplicación o ejecutar el ciclo de carga de plugins.

Al iniciar, el sistema examina todos los ensamblados (assemblies) almacenados en la carpeta /Plugins, identificando aquellos tipos que implementan la interfaz IPuntuacion. Si el tipo cumple los requisitos mínimos, la clase es registrada como una estrategia válida y queda disponible en el sistema.

Todo desarrollador que desee incorporar una nueva estrategia debe implementar la siguiente interfaz:

```
public interface IPuntuacion
{
   string Nombre { get; }
   string Descripcion { get; }
   int CalcularPuntos(AccesoAtraccion acceso, IEnumerable<AccesoAtraccion>
accesosDelDia);
}
```

Esta interfaz constituye el contrato oficial del sistema. Las reglas son las siguientes: la clase debe ser pública, debe implementar todas las propiedades y métodos definidos arriba, debe poseer un constructor público sin parámetros, no debe requerir dependencias externas obligatorias no incluidas dentro del plugin, si una clase no cumple alguno de estos requisitos, se va a descartar automáticamente durante la carga.

Cuando la aplicación se inicia, el componente PluginManager activa el proceso de carga dinámica. El flujo es el siguiente: se buscan todos los archivos con extensión .dll, cada DLL se carga en memoria mediante Assembly.LoadFrom, el sistema obtiene todos los tipos contenidos en la DLL a través de Reflection, el sistema almacena el tipo en un diccionario interno, usando su nombre como clave y al completar la carga, la interfaz gráfica consulta la lista de estrategias registradas y las presenta al usuario final como opciones disponibles.

Todo este flujo se ejecuta sin necesidad de recompilar el sistema ni alterar su código, permitiendo que nuevos comportamientos sean incorporados en forma modular y segura.

Un desarrollador externo puede agregar una nueva estrategia siguiendo los pasos que se detallan a continuación.

dotnet new classlib -n MiNuevaEstrategia

<ProjectReference Include="..\ParqueTematico.Domain\ParqueTematico.Domain.csproj" />

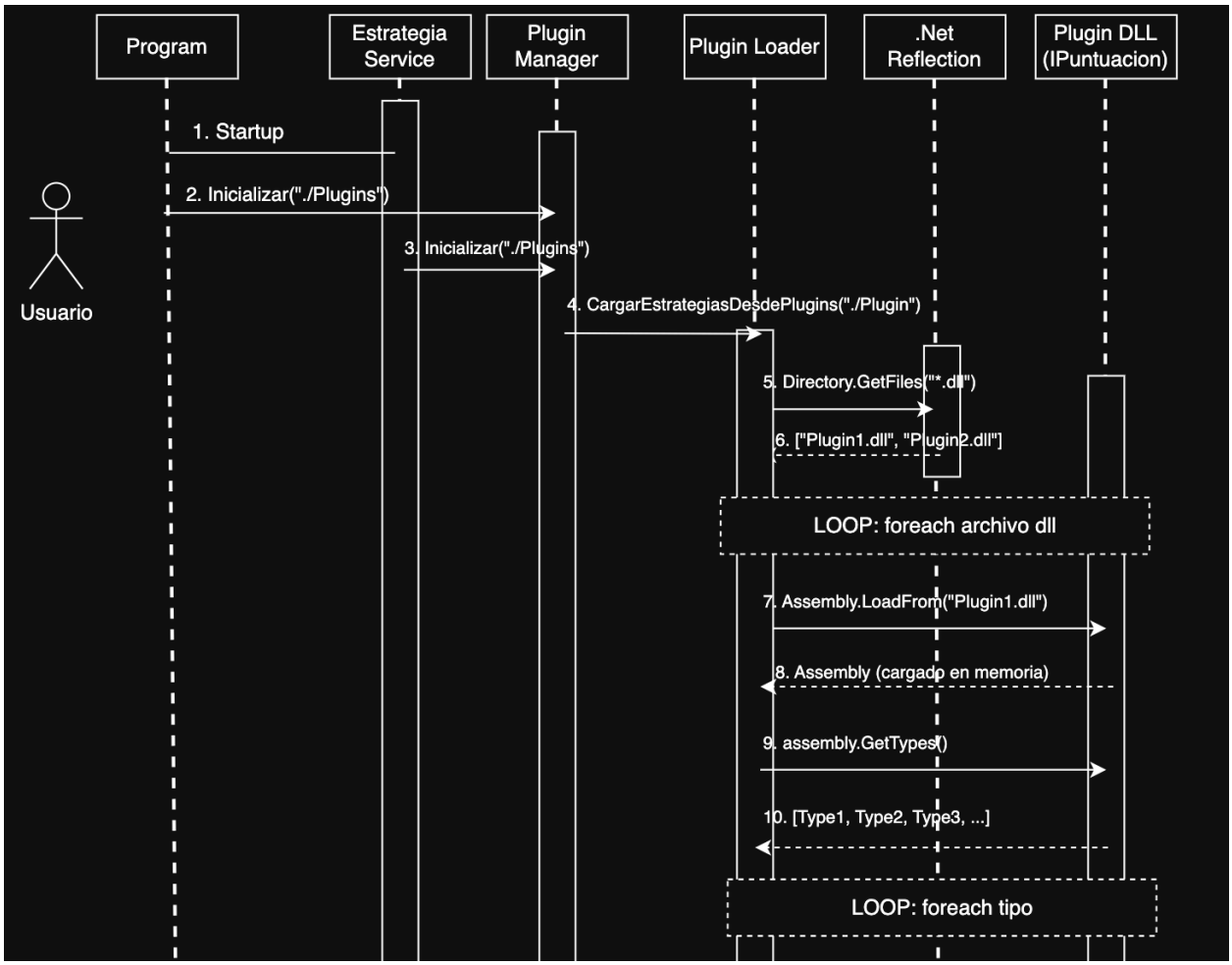
```
using Dominio.Entities.Puntuacion;

public class PuntuacionPorHora : IPuntuacion
{
   public string Nombre => "PuntuacionPorHora";
   public string Descripcion => "Multiplica puntos durante ciertas horas del día";

   public int CalcularPuntos(AccesoAtraccion acceso, IEnumerable<AccesoAtraccion> accesosDelDia)
   {
      int puntosBase = 50;
      int hora = acceso.FechaHoraIngreso.Hour;

      if (hora >= 18 && hora < 22)
         return puntosBase * 2;

      return puntosBase;
   }
}
```

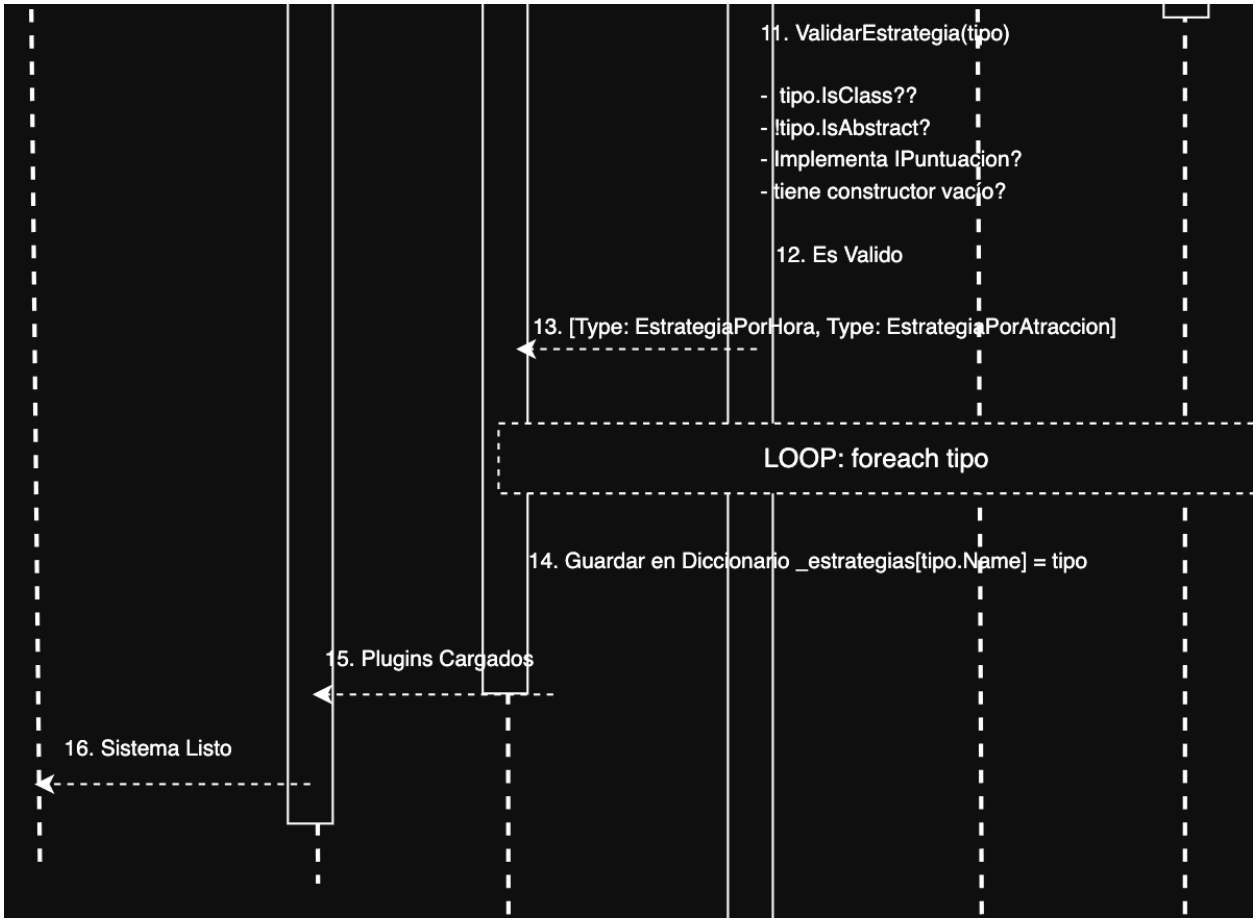
dotnet build
El archivo DLL estará disponible dentro de la carpeta bin dentro de la carpeta del proyecto.

Copiar el DLL a
ParqueTematico.WebApi/Plugins
Al reiniciar la aplicación, el sistema detectará automáticamente la nueva estrategia y la mostrará en la interfaz gráfica.

Como parte de la documentación y del requerimiento académico, se entrega un plugin ejemplo ya compilado que implementa la lógica solicitada: multiplicar el puntaje base durante un rango horario específico.
Este plugin va a sirvir como referencia para que otros desarrolladores entiendan cómo estructurar sus propias DLLs y cómo cumplir con el contrato definido por la interfaz.

A continuación se muestra un diagrama de secuencia detallando este proceso.





1.14. Mejoras en comparación con la primera entrega

Mejora basada en LSP
Antes de la refactorización, los repositorios no compartían una estructura común, lo que generaba duplicación de métodos, impedía el polimorfismo y hacía imposible reutilizar lógica genérica. Para resolverlo se creó la interfaz IBaseRepository, que define de forma uniforme las operaciones CRUD básicas. Con esta abstracción, todos los repositorios específicos pasan a heredar la misma interfaz base.

Mejora basada en ISP y DIP
La interfaz original IAccesoAtraccionService mezclaba responsabilidades de lectura y escritura, obligando a los consumidores a depender de métodos que no necesitaban. Para corregirlo, se aplicó el principio ISP dividiendo la interfaz en dos contratos independientes: IComandoAccesoAtraccion (solo escritura) e IConsultaAccesoAtraccion.

Además se consiguió desacoplar los paquetes creando nuevos paquetes BusinessLogicInterface y RepositoriesInterfaces. Esto hace que se dependan de abstracciones en vez de clases concretas y se desacoplaron las distintas capas del diseño.

Mejora basada en OCP
El filtro de excepciones estaba implementado con un gran switch, lo que implicaba modificar el código cada vez que se agregaba una nueva excepción. Para cumplir con OCP, este mecanismo fue reemplazado por el patrón Strategy mediante la creación de la interfaz IExceptionHandler.

Mejora basada en SRP Single Responsibility Principle
AccesoAtraccionService concentraba múltiples responsabilidades (validación, registro, cálculo de puntos y reportes), formando una clase difícil de mantener y testear. Se aplicó SRP dividiendo esta lógica en cuatro servicios internos: ValidacionAccesoService, RegistroAccesoService, GestionPuntosAccesoService y ReporteAccesoService, quedando AccesoAtraccionService como una fachada que coordina acciones entre ellos.

Mejora basada en REST
Se mejoraron los status code response de la WebApi para que retornen los mensajes adecuados. Se utiliza un ExceptionFilter para evitar dar el status code 500 y siempre poder dar un mensaje adecuado. Se buscó dar el mensaje adecuado y la información relevante. Por ejemplo al crear un objeto retornar los datos de ese objeto creado o cuando hay un error 400 aclarar que field es el que falta o está incorrecto.

Mejora basada en Unit Test
Se buscó utilizar doubles para testear una unidad de código a la vez y poder simular el comportamiento de las clases que no estamos testeando. Esto nos lo permitió la creación de las nuevas interfaces.

Mejora basada en OCP en Infrastructure
Se creó la clase BaseRepository que contiene los métodos base que todos los repositorios deben utilizar. Todos los repositorios heredan de esta clase o si no es necesario ampliarla se utiliza BaseRepository<T>, siendo T el tipo de entidad del dominio que queremos guardar en base de datos. Esto nos permite no tener que volver a crear estos métodos en todos los repositorios, haciendo posible agregar nuevos repositorios sin tener que agregar código nuevo y haciendo el proyecto más extensible.

2. Anexo

1.1. Errores conocidos y oportunidades de mejora

1- No se puede modificar la hora del sistema ya que no se utilizó para tener en cuenta la fecha y hora. Aún así el reloj está implementado en el backend por lo que sería sencillo hacer un refactoreo para que administrador pueda modificar la fecha y hora desde el frontend.
2- Algunos diagramas no representan al 100% el código ya que se fue haciendo la documentación a la vez que se terminaba de refactorear el código
3- No todos los repositorios heredan de BaseRepository, porque fue agregado como parte de un refactoreo tardío.

1.2. Descripción de los endpoints

Para todos los endpoints se deberá poner el header:
Content-Type | application/json

Login
Desde este endpoint pueden generar su token todos los usuarios, para luego poder hacer el resto de acciones estando autenticado.

URL: http://localhost:5203/api/auth/login
Verbos HTTP: POST

Headers: No requiere autenticación

Ejemplo Request: {
  "Email": "admin@admin.com",
  "Contrasenia": "Admin123456798!"
}

Ejemplo Response: {

"token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImFkbWluQGFkbWluLmNvbSIsInRpcG9Vc3VhcmlvIjoiQWRtaW5pc3RyYWRvciIsInVuaXF1ZV9uYW1lIjoiYWRtaW5AYWRtaW4uY29tIiwiaWQiOiI1NjYwMjgwYi1jNWE0LTQwYmMtYjA5Ny0yNjMyZTZkOTlkZWIiLCJuYmYiOjE3NjM2NTczMDAsImV4cCI6MTc2MzY2MDkwMCwiaWF0IjoxNzYzNjU3MzAwfQ.KZ3Spm1MhShBQeB7rre9AAi70bn5Fj9qXXVEiG8sG20",
"expiraEn": "2025-11-20T17:48"
}

Status Codes:
Login éxitoso → 200 OK
Datos incorrectos → 401 Unauthorized ("Credenciales erróneas")
Datos faltantes → 400 Bad Request ("Se requiere campo contraseña")

Registrar ingreso a una atracción

Permite registrar el ingreso de un visitante a una atracción, generando un nuevo acceso.

URL: http://localhost:5203/api/accesos/ingreso
Verbos HTTP: POST

Headers: Requiere autenticación con rol Operador

Ejemplo Request:

```
{
  "TicketId": "guid-ticket",
  "AtraccionId": "guid-atraccion"
}
```

Ejemplo Response:

```
{
  "accesoId": "guid-acceso-generado"
}
```

Status Codes:
Registro exitoso → 201 Created
Ticket o atracción no válidos → 400 Bad Request
Usuario sin permisos → 403 Forbidden

Registrar egreso de una atracción

Registra el egreso de un visitante previamente ingresado.

URL: http://localhost:5203/api/accesos/egreso/{accesoId}
Verbos HTTP: PUT

Headers: Requiere autenticación con rol Operador

Ejemplo Request:
No requiere body.

Ejemplo Response:
No devuelve contenido.

Status Codes:
Egreso registrado → 204 No Content
Acceso inexistente → 404 Not Found
Usuario sin permisos → 403 Forbidden

Obtener aforo actual de una atracción

Devuelve la cantidad de personas actualmente dentro de una atracción.

URL: http://localhost:5203/api/accesos/aforo/{atraccionId}
Verbos HTTP: GET

Headers: Requiere autenticación con rol Operador

Ejemplo Request:

No requiere body.

Ejemplo Response:

```
{
   "aforoActual": 125
}
```

Status Codes:
Consulta exitosa → 200 OK
Atracción no encontrada → 404 Not Found
Usuario sin permisos → 403 Forbidden

Obtener acceso por ID

Devuelve la información de un acceso registrado.

URL: http://localhost:5203/api/accesos/{id}
Verbos HTTP: GET

Headers: Requiere autenticación con rol Operador

Ejemplo Request:
No requiere body.

Ejemplo Response:

```
{
   "Id": "guid",
   "TicketId": "guid",
   "AtraccionId": "guid",
   "FechaIngreso": "2025-11-20T14:25",
   "FechaEgreso": "2025-11-20T15:10"
}
```

Status Codes:
Acceso encontrado → 200 OK
Acceso inexistente → 404 Not Found
Usuario sin permisos → 403 Forbidden

Obtener reporte de uso de atracciones

Devuelve un listado con las estadísticas de uso de todas las atracciones en un rango de fechas.

URL: http://localhost:5203/api/accesos/reporte-uso
Verbos HTTP: GET

Headers: Requiere autenticación con rol Operador

Query Params:

fechaInicio

fechaFin

Ejemplo Request:
Sin body, solo query params.

Ejemplo Response:

```
[
  {
    "AtraccionId": "guid",
    "CantidadIngresos": 350,
    "CantidadEgresos": 342
  }
]
```

Status Codes:
Consulta exitosa → 200 OK

Usuario sin permisos → 403 Forbidden

Obtener mi historial de accesos (Visitante)

Devuelve el listado de accesos del visitante autenticado en una fecha determinada.

URL: http://localhost:5203/api/accesos/mi-historial
Verbos HTTP: GET

Headers: Requiere autenticación con rol Visitante

Query Params:

fecha

Ejemplo Request:
Sin body.

Ejemplo Response:

```
[
   {
      "Id": "guid",
      "TicketId": "guid",
      "AtraccionId": "guid",
      "FechaIngreso": "2025-11-20T14:25",
      "FechaEgreso": "2025-11-20T15:10"
   }
]
```

Status Codes:
Consulta exitosa → 200 OK
Usuario sin permisos → 403 Forbidden

Obtener atracciones visitadas por un visitante específico

Devuelve todos los accesos de un visitante y fecha dada.

URL: http://localhost:5203/api/accesos/visitante/{visitanteId}
Verbos HTTP: GET

Headers: Requiere autenticación con rol Operador

Query Params:

fecha

Ejemplo Request:
Sin body.

Ejemplo Response:

```
[
   {
      "Id": "guid",
      "AtraccionId": "guid",
      "FechaIngreso": "2025-11-20T14:25",
      "FechaEgreso": "2025-11-20T15:10"
   }
]
```

Status Codes:
Consulta exitosa → 200 OK
Visitante inexistente → 404 Not Found
Usuario sin permisos → 403 Forbidden

Crear atracción

Crea una nueva atracción en el sistema.

URL: http://localhost:5203/api/atracciones
Verbos HTTP: POST

Headers: Requiere autenticación con rol Administrador

Ejemplo Request:

```
{
   "Nombre": "Montaña Rusa",
   "Tipo": "Intensa",
   "EdadMinima": 12,
   "CapacidadMaxima": 24,
   "Descripcion": "Atracción de alta velocidad",
   "Disponible": true
}
```

Ejemplo Response:

```
{}
```

Status Codes:
Creación exitosa → 200 OK
Datos inválidos → 400 Bad Request
Usuario sin permisos → 403 Forbidden

Obtener atracción por ID

Devuelve la información de una atracción específica.

URL: http://localhost:5203/api/atracciones/{id}
Verbos HTTP: GET

Headers: Requiere autenticación con rol Administrador

Ejemplo Request:
Sin body.

Ejemplo Response:


```
{
   "Id": "guid",
   "Nombre": "Montaña Rusa",
   "Tipo": "Intensa",
   "EdadMinima": 12,
   "CapacidadMaxima": 24,
   "Descripcion": "Atracción de alta velocidad",
   "Disponible": true
}
```


Status Codes:
Consulta exitosa → 200 OK
Atracción inexistente → 404 Not Found
Usuario sin permisos → 403 Forbidden

Obtener todas las atracciones

Devuelve la lista completa de atracciones registradas.

URL: http://localhost:5203/api/atracciones
Verbos HTTP: GET

Headers: Requiere autenticación con rol Administrador

Ejemplo Request:
Sin body.

Ejemplo Response:


```
[
   {
      "Id": "guid",
      "Nombre": "Montaña Rusa",
      "Tipo": "Intensa",
```

```
    "EdadMinima": 12,
    "CapacidadMaxima": 24,
    "Descripcion": "Atracción de alta velocidad",
    "Disponible": true
  }
]
```


Status Codes:
Consulta exitosa → 200 OK
Usuario sin permisos → 403 Forbidden

Actualizar atracción

Actualiza los datos de una atracción existente.

URL: http://localhost:5203/api/atracciones/{id}
Verbos HTTP: PUT

Headers: Requiere autenticación con rol Administrador

Ejemplo Request:


```
{
  "Nombre": "Montaña Rusa Extrema",
  "Tipo": "Intensa",
  "EdadMinima": 14,
  "CapacidadMaxima": 28,
  "Descripcion": "Actualizada",
  "Disponible": true
}
```


Ejemplo Response:


```
{}
```

Status Codes:
Actualización exitosa → 200 OK
Atracción inexistente → 404 Not Found
Datos inválidos → 400 Bad Request
Usuario sin permisos → 403 Forbidden

Eliminar atracción

Elimina una atracción del sistema.

URL: http://localhost:5203/api/atracciones/{id}
Verbos HTTP: DELETE

Headers: Requiere autenticación con rol Administrador

Ejemplo Request:
Sin body.

Ejemplo Response:

{}

Status Codes:
Eliminación exitosa → 200 OK
Atracción inexistente → 404 Not Found
Usuario sin permisos → 403 Forbidden

Obtener mis canjes

Devuelve todos los canjes realizados por el visitante autenticado.

URL: http://localhost:5203/api/canje
Verbos HTTP: GET

Headers: Requiere autenticación con rol Visitante

Ejemplo Request:
Sin body.

Ejemplo Response:


```
[
   {
      "Id": "guid",
      "RecompensaId": "guid",
      "UsuarioId": "guid",
      "FechaCanje": "2025-11-20T14:00"
   }
]
```


Status Codes:
Consulta exitosa → 200 OK
Usuario sin permisos → 403 Forbidden

Obtener canje por ID

Devuelve la información de un canje específico.

URL: http://localhost:5203/api/canje/{id}
Verbos HTTP: GET

Headers: No requiere autenticación

Ejemplo Request:
Sin body.

Ejemplo Response:


```
{
   "Id": "guid",
   "RecompensaId": "guid",
   "UsuarioId": "guid",
```

```
  "FechaCanje": "2025-11-20T14:00"
}
```


Status Codes:
Consulta exitosa → 200 OK
Canje inexistente → 404 Not Found

Crear canje

Permite que un visitante canjee una recompensa.

URL: http://localhost:5203/api/canje
Verbos HTTP: POST

Headers: Requiere autenticación con rol Visitante

Ejemplo Request:


```
{
  "RecompensaId": "guid"
}
```


Ejemplo Response:


```
{
  "Id": "guid",
  "RecompensaId": "guid",
  "UsuarioId": "guid",
  "FechaCanje": "2025-11-20T14:00"
}
```


Status Codes:
Canje creado → 201 Created
Datos incorrectos → 400 Bad Request

Usuario sin permisos → 403 Forbidden

Actualizar canje

Actualiza un canje existente.

URL: http://localhost:5203/api/canje/{id}
Verbos HTTP: PUT

Headers: No requiere autenticación

Ejemplo Request:

```
{
   "UsuarioId": "guid",
   "RecompensaId": "guid"
}
```

Ejemplo Response:

```
{
   "Id": "guid",
   "RecompensaId": "guid",
   "UsuarioId": "guid",
   "FechaCanje": "2025-11-20T14:00"
}
```

Status Codes:
Actualización exitosa → 200 OK
Canje inexistente → 404 Not Found
Datos incorrectos → 400 Bad Request

Eliminar canje

Elimina un canje del sistema.

URL: http://localhost:5203/api/canje/{id}
Verbos HTTP: DELETE

Headers: No requiere autenticación

Ejemplo Request:
Sin body.

Ejemplo Response:
Sin contenido.

Status Codes:
Eliminación exitosa → 204 No Content
Canje inexistente → 404 Not Found

Recurso: Estrategias
Path: /api/estrategias
Método HTTP: GET
Headers: ninguno
Body: ninguno
Status Codes:

200 OK: devuelve IEnumerable<EstrategiaDto>

Recurso: Estrategia activa
Path: /api/estrategias/activa
Método HTTP: GET
Headers: ninguno
Body: ninguno
Status Codes:

200 OK: devuelve EstrategiaDto

Recurso: Cambiar estrategia activa
Path: /api/estrategias/activa
Método HTTP: PUT

Headers: ninguno
Body:

CambiarEstrategiaRequest

```
{
  "NombreEstrategia": "string"
}
```

Status Codes:

204 No Content: estrategia cambiada con éxito

400 Bad Request: la estrategia indicada no existe

1. Crear evento
Path: /api/eventos
Método HTTP: POST
Headers: requiere rol Administrador
Body:

CrearEventoRequest

```
{
  "Nombre": "string",
  "Fecha": "2025-01-01",
  "Hora": "14:00",
  "Aforo": 100,
  "CostoAdicional": 50
}
```

Status Codes:

201 Created: evento creado

400 Bad Request: datos inválidos

2. Obtener evento por id
Path: /api/eventos/{id}
Método HTTP: GET
Headers: ninguno
Body: ninguno
Status Codes:

200 OK: devuelve EventoDto

404 Not Found: no existe el evento

3. Eliminar evento
Path: /api/eventos/{id}
Método HTTP: DELETE
Headers: requiere rol Administrador
Body: ninguno
Status Codes:

204 No Content: eliminado

404 Not Found: no existe el evento

4. Obtener todos los eventos
Path: /api/eventos
Método HTTP: GET
Headers: ninguno
Body: ninguno
Status Codes:

200 OK: devuelve List<EventoDto>

5. Agregar atracción a evento
Path: /api/eventos/{eventoId}/atracciones/{atraccionId}
Método HTTP: POST
Headers: requiere rol Administrador
Body: ninguno
Status Codes:

200 OK: atracción agregada

404 Not Found: evento o atracción inexistentes

6. Eliminar atracción de evento
Path: /api/eventos/{eventoId}/atracciones/{atraccionId}
Método HTTP: DELETE
Headers: requiere rol Administrador
Body: ninguno
Status Codes:

204 No Content: atracción eliminada

404 Not Found: evento o atracción inexistentes

1. Obtener historial de un visitante específico
Path: /api/historial-puntuacion/visitante/{visitanteId}
Método HTTP: GET
Headers: requiere rol Visitante
Body: ninguno
Status Codes:

200 OK: devuelve List<HistorialPuntuacionDto>

404 Not Found: el visitante no tiene historial (si aplica según lógica del servicio)

2. Obtener mi historial
Path: /api/historial-puntuacion/mi-historial
Método HTTP: GET
Headers: requiere rol Visitante
Body: ninguno
Status Codes:

200 OK: devuelve List<HistorialPuntuacionDto>
1. Crear incidencia
Path: /api/incidencias

Método HTTP: POST
Headers: requiere rol Administrador
Body (JSON):

```
{
  "AtraccionId": "guid",
  "TipoIncidencia": "string",
  "Descripcion": "string"
}
```

Response (201 Created): IncidenciaDto
Status Codes:

201 Created: incidencia creada

400 Bad Request: datos inválidos

2. Cerrar incidencia
Path: /api/incidencias/{id}/cerrar
Método HTTP: PUT
Headers: requiere rol Administrador
Body: ninguno
Status Codes:

204 No Content: incidencia cerrada

404 Not Found: incidencia inexistente

3. Obtener incidencias activas por atracción
Path: /api/incidencias
Método HTTP: GET
Headers: requiere rol Administrador
Query Params:

atraccionId (Guid)

activas (bool, default: false)

Body: ninguno
Responses:

200 OK: lista de IncidenciaDto

si activas = true, devuelve incidencias activas

si activas = false, devuelve lista vacía

Status Codes:

200 OK

4. Consultar si existe incidencia activa
Path: /api/incidencias/existe-activa
Método HTTP: GET
Headers: requiere rol Administrador
Query Params:

atraccionId (Guid)

Body: ninguno
Response (200 OK):

true o false

Status Codes:

200 OK

1. Crear mantenimiento preventivo
Path: /api/mantenimientos
Método HTTP: POST
Headers: requiere rol Administrador
Body (JSON):
{
  "AtraccionId": "guid",

```
  "Descripcion": "string",
  "FechaInicio": "2025-01-01T00:00:00",
  "FechaFin": "2025-01-02T00:00:00"
}
```

Response (201 Created): MantenimientoPreventivoDto
Status Codes:

201 Created: mantenimiento creado

400 Bad Request: datos inválidos

2. Obtener mantenimiento por ID
Path: /api/mantenimientos/{id}
Método HTTP: GET
Headers: requiere rol Administrador
Body: ninguno
Response (200 OK): MantenimientoPreventivoDto
Status Codes:

200 OK: devuelve el mantenimiento

404 Not Found: no existe mantenimiento con ese ID

3. Obtener todos los mantenimientos
Path: /api/mantenimientos
Método HTTP: GET
Headers: requiere rol Administrador
Body: ninguno
Response (200 OK): lista de MantenimientoPreventivoDto
Status Codes:

200 OK

4. Finalizar mantenimiento
Path: /api/mantenimientos/{id}/finalizar
Método HTTP: PUT

Headers: requiere rol Administrador
Body: ninguno
Status Codes:

204 No Content: finalizado correctamente

404 Not Found: no existe mantenimiento con ese ID

1. Listar estrategias disponibles
Path: /api/ranking/estrategias
Método HTTP: GET
Headers: requiere rol Administrador
Body: ninguno
Response (200 OK):

Lista de nombres de estrategias.

Status Codes:

200 OK

2. Obtener estrategia activa
Path: /api/ranking/estrategia-activa
Método HTTP: GET
Headers: requiere rol Administrador
Body: ninguno
Response (200 OK):

```
{
  "estrategiaActiva": "NombreEstrategia"
}
```

Status Codes:

200 OK

3. Cambiar estrategia activa
Path: /api/ranking/estrategia-activa

Método HTTP: PUT
Headers: requiere rol Administrador
Body (JSON):

```
{
  "NombreEstrategia": "string"
}
```

Response (200 OK):

```
{
  "mensaje": "Estrategia actualizada correctamente",
  "estrategiaActiva": "NuevaEstrategia"
}
```

Status Codes:

200 OK: estrategia cambiada

400 Bad Request: estrategia inválida o inexistente

4. Obtener ranking diario
Path: /api/ranking/diario
Método HTTP: GET
Headers: requiere rol Administrador
Body: ninguno
Response (200 OK): lista de posiciones

Ejemplo:

```
[
  {
    "posicion": 1,
    "visitanteId": "guid",
    "nombreCompleto": "Juan Pérez",
    "puntosTotales": 1500
  }
]
```

Status Codes:

200 OK

1. Obtener todas las recompensas
Path: /api/recompensa
Método HTTP: GET
Headers: no requiere autenticación especial
Body: ninguno
Response (200 OK):

Lista de recompensas creadas.

Status Codes:

200 OK

2. Obtener recompensa por ID
Path: /api/recompensa/{id}
Método HTTP: GET
Headers: no requiere autenticación especial
Body: ninguno
Response (200 OK):

```
{
  "id": "guid",
  "nombre": "string",
  "descripcion": "string",
  "costoPuntos": 100
}
```

Status Codes:

200 OK

404 Not Found: si no existe

3. Crear recompensa

Path: /api/recompensa
Método HTTP: POST
Headers: requiere rol Administrador
Body (JSON):

```
{
  "nombre": "string",
  "descripcion": "string",
  "costoPuntos": 100
}
```

Response (201 Created):

```
{
  "id": "guid",
  "nombre": "string",
  "descripcion": "string",
  "costoPuntos": 100
}
```

Status Codes:

201 Created

400 Bad Request: datos incompletos o inválidos

4. Actualizar recompensa
Path: /api/recompensa/{id}
Método HTTP: PUT
Headers: requiere rol Administrador
Body (JSON):

```
{
  "nombre": "string",
  "descripcion": "string",
  "costoPuntos": 100
}
```

Response (200 OK):

Recompensa actualizada.

Status Codes:

200 OK

400 Bad Request

404 Not Found

5. Eliminar recompensa
Path: /api/recompensa/{id}
Método HTTP: DELETE
Headers: requiere rol Administrador
Body: ninguno
Response (204 No Content)
Status Codes:

204 No Content

404 Not Found

1. Obtener fecha y hora del reloj
Path: /api/relojes
Método HTTP: GET
Headers: ninguno
Body: ninguno
Response (200 OK): RelojDto

Ejemplo Response:

{
  "FechaHora": "2025-11-20T15:30:00"
}

Status Codes:

200 OK

2. Modificar fecha y hora del reloj
Path: /api/relojes
Método HTTP: PUT
Headers: requiere rol Administrador
Body (JSON): RelojDto
{
  "FechaHora": "2025-11-21T09:00:00"
}

Response (200 OK): RelojDto

Ejemplo Response:

{
  "FechaHora": "2025-11-21T09:00:00"
}

Status Codes:

200 OK: modificación exitosa

400 Bad Request: datos inválidos

403 Forbidden: usuario sin permisos administrador

1. Comprar ticket
Path: /api/tickets
Método HTTP: POST
Headers: requiere rol Visitante
Body (JSON): ComprarTicketRequest
{
  "fechaVisita": "2025-12-01T10:00:00",
  "tipoEntrada": "General",
  "eventoId": "f3c1a6d5-0fb5-4d11-baaa-0e9a8bb9c123"
}

Response (201 Created):

Body:


```
{
  "ticketId": "c2e7d36c-32d1-4f1d-a5f2-3f7e0a971111"
}
```


Location apunta a /api/tickets/{id}

Status Codes:

201 Created: ticket creado correctamente

400 Bad Request: datos inválidos

401 Unauthorized: usuario no autenticado

403 Forbidden: usuario sin rol visitante

2. Obtener ticket por ID
Path: /api/tickets/{id}
Método HTTP: GET
Headers: requiere rol Operador
Body: ninguno
Response (200 OK): TicketDto

Ejemplo:


```
{
  "id": "c2e7d36c-32d1-4f1d-a5f2-3f7e0a971111",
  "fechaVisita": "2025-12-01T10:00:00",
  "tipoEntrada": "General",
  "visitanteId": "1a2b3c4d-1234-5678-9999-aabbccddeeff"
}
```

Status Codes:

200 OK

404 Not Found: no existe ticket con ese ID

401 Unauthorized

403 Forbidden

3. Obtener tickets por visitante
Path: /api/tickets/visitante/{visitanteId}
Método HTTP: GET
Headers: requiere rol Operador
Body: ninguno
Response (200 OK): List<TicketDto>

Ejemplo:


```
[
  {
    "id": "11111111-2222-3333-4444-555555555555",
    "fechaVisita": "2025-12-01T10:00:00",
    "tipoEntrada": "General",
    "visitanteId": "1a2b3c4d-1234-5678-9999-aabbccddeeff"
  },
  {
    "id": "66666666-7777-8888-9999-aaaaaaaaaaaa",
    "fechaVisita": "2025-12-05T15:00:00",
    "tipoEntrada": "VIP",
    "visitanteId": "1a2b3c4d-1234-5678-9999-aabbccddeeff"
  }
]
```


Status Codes:

200 OK

401 Unauthorized

403 Forbidden

1. Crear usuario
Path: /api/usuarios
Método HTTP: POST
Headers: ninguno (endpoint público)
Body (JSON): CrearUsuarioRequest

Ejemplo:

```
{
  "nombre": "Juan",
  "apellido": "Pérez",
  "email": "juan@example.com",
  "password": "Password123",
  "rol": "Visitante"
}
```

Response (201 Created):

Body:

```
{
  "id": "d4b05c0b-3ae8-4bb5-8992-9a08bfa1d111",
  "nombre": "Juan",
  "apellido": "Pérez",
  "email": "juan@example.com",
  "rol": "Visitante"
}
```

Location → /api/usuarios/{id}

Status Codes:

201 Created

400 Bad Request

2. Listar usuarios
Path: /api/usuarios
Método HTTP: GET
Headers: requiere autenticación
Body: ninguno
Response (200 OK): lista de usuarios

Ejemplo:

```
[
  {
    "id": "bc1f0a22-0a55-472d-b7bd-9fb15f502222",
    "nombre": "Ana",
    "apellido": "García",
    "email": "ana@example.com",
    "rol": "Operador"
  }
]
```

Status Codes:

200 OK

401 Unauthorized

3. Obtener usuario por ID
Path: /api/usuarios/{id}
Método HTTP: GET
Headers: requiere autenticación
Body: ninguno
Response (200 OK):

{
 "id": "bc1f0a22-0a55-472d-b7bd-9fb15f503333",
 "nombre": "Ana",
 "apellido": "García",
 "email": "ana@example.com",
 "rol": "Operador"
}

Status Codes:

200 OK

404 Not Found

401 Unauthorized

4. Obtener mi información
Path: /api/usuarios/mi-informacion
Método HTTP: GET
Headers: requiere rol Visitante
Body: ninguno
Response (200 OK): usuario obtenido a partir del JWT

Ejemplo:

{
 "id": "99999999-aaaa-bbbb-cccc-111111111111",
 "nombre": "Luis",
 "apellido": "Piñeyro",
 "email": "luis@example.com",
 "rol": "Visitante"
}

Status Codes:

200 OK

404 Not Found

401 Unauthorized

403 Forbidden

5. Actualizar usuario
Path: /api/usuarios/{id}
Método HTTP: PUT
Headers: requiere autenticación
Body (JSON): ActualizarUsuarioRequest

Ejemplo:

```
{
  "nombre": "Juan Francisco",
  "apellido": "Pérez López"
}
```

Response (204 No Content): sin body
Status Codes:

204 No Content

400 Bad Request

404 Not Found

401 Unauthorized

6. Eliminar usuario
Path: /api/usuarios/{id}
Método HTTP: DELETE
Headers: requiere autenticación
Body: ninguno
Response (204 No Content)
Status Codes:

204 No Content

404 Not Found

401 Unauthorized