UNIVERSIDAD ORT
Uruguay

Universidad ORT Uruguay
Facultad de Ingeniería

Diseño de Aplicaciones 2

Primer Obligatorio

Cristian Palma 208443

Federico Alonso 182999

Descripción del diseño

Grupo N6A

Repositorio: https://github.com/ORT-DA2/182999_208443

Índice

1. Descripción del diseño

1.1 Descripción general del trabajo

Para la solución del obligatorio se implementa una WEB API como backend, la cual define las funcionalidades necesarias para administrar los incidentes en los proyectos de software cumpliendo con todos los requerimientos exigidos en la letra del obligatorio.

El sistema tiene como funcionalidad principal la gestión de Incidentes asociados a un proyecto, contemplando roles como administrador, desarrollador y tester.

Los administradores del sistema pueden realizar altas de **usuarios** registrando su nombre, apellido, nombre de usuario, contraseña, dirección de correo electrónico y su rol. También pueden crear **proyectos** los cuales tienen nombre del proyecto , testers y desarrolladores asignados e incidentes.

El sistema cuenta con un módulo que permite importar incidentes en formato XML y en archivo de texto.

Los administradores tienen acceso a los reportes de cantidad de bugs por proyecto y cantidad de bugs resueltos por un desarrollador.

Los usuarios con el rol tester, pueden ver todos los incidentes de todos los proyectos a los cuales pertenecen y también pueden crear, modificar y eliminar incidentes.

Los usuarios con rol desarrollador pueden ver los proyectos a los cuales pertenecen y visualizar los bugs de estos modificando su estado en (activo/resuelto).

1.2 Errores conocidos

- Por falta de tiempo para esta versión del producto no se utilizó la clase mapper y se hicieron los mapeos manualmente.
- Falto realizar la prueba en los filtros de autorización y de errores, las mismas quedaron comentadas a fin de implementar para la próxima entrega. El ejecutarlas provoca un error al intentar instanciar el servicio ILogicaAutorización desde el contexto empleado.

```
public void OnAuthorization(AuthorizationFilterContext context)
{
    _autorizacion = (ILogicaAutorizacion)context.HttpContext.RequestServices.GetService(typeof(ILogicaAutorizacion));

    string token = context.HttpContext.Request.Headers["autorizacion"];
    if (token == null)
    {
        context.Result = new ContentResult()
```

- De manera similar a la anterior, en la capa de datos existe un método de prueba comentado en el cual se prueba la relación N a N entre los proyectos y los usuarios asignados. El mismo funciona correctamente con la aplicación corriendo y genera error al ejecutar las pruebas con InMemory. Dicho error se debe a que en el método se debe cerrar el contexto de la base de datos y al intentar vaciar la misma provoca el lanzamiento de una excepción.

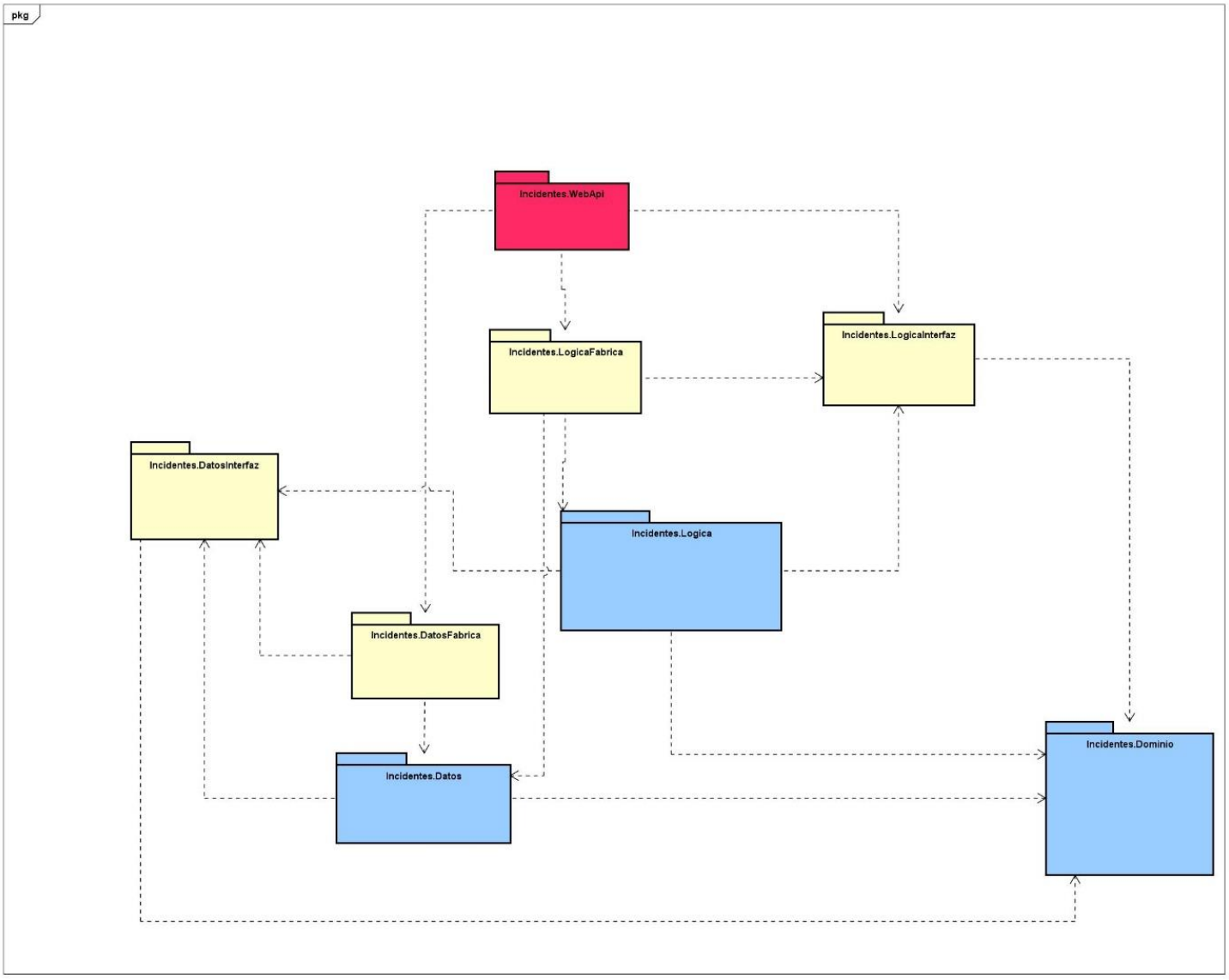
```
[Test]
public void se_pueden_pasar_la_autorizacion()
{
    /*DefaultHttpContext httpContext = new DefaultHttpContext();
    httpContext.Request.Headers["autorizacion"] = "token";
    AuthorizationFilterContext actionContext = new AuthorizationFilterContext(
        new ActionContext()
        {
            HttpContext = httpContext,
            ActionDescriptor = new ActionDescriptor(),
            RouteData = new RouteData()
        },
        new List<IFilterMetadata>()
    );

    _logicaA.Setup(c => c.TokenValido(It.IsAny<string>(), It.IsAny<string[]>())).Returns(true);

    _aFilter.OnAuthorization(actionContext);*/
    Assert.Pass();
}
```

Se deja el método comentado para evidenciar que se trabajó con él hasta ese momento y se procurará solucionarlo para la siguiente entrega.

1.3 Diagrama general de paquetes



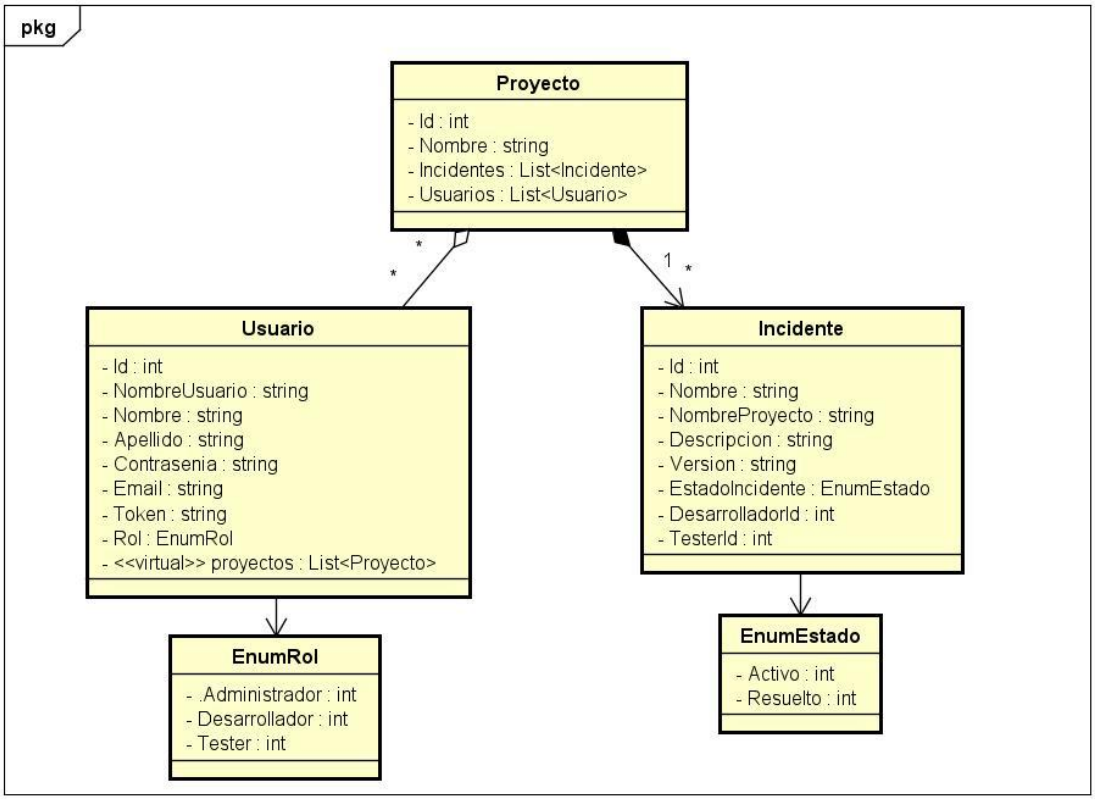
En el presente diagrama podemos observar el bajo acoplamiento que existe entre las capas principales (las cuales están de color) del backend:

- WebApi
- Logica
- Datos
- Dominio

Para ellos se utilizaron paquetes de interfaces y fabricas que en el presente documento vamos a detallar.

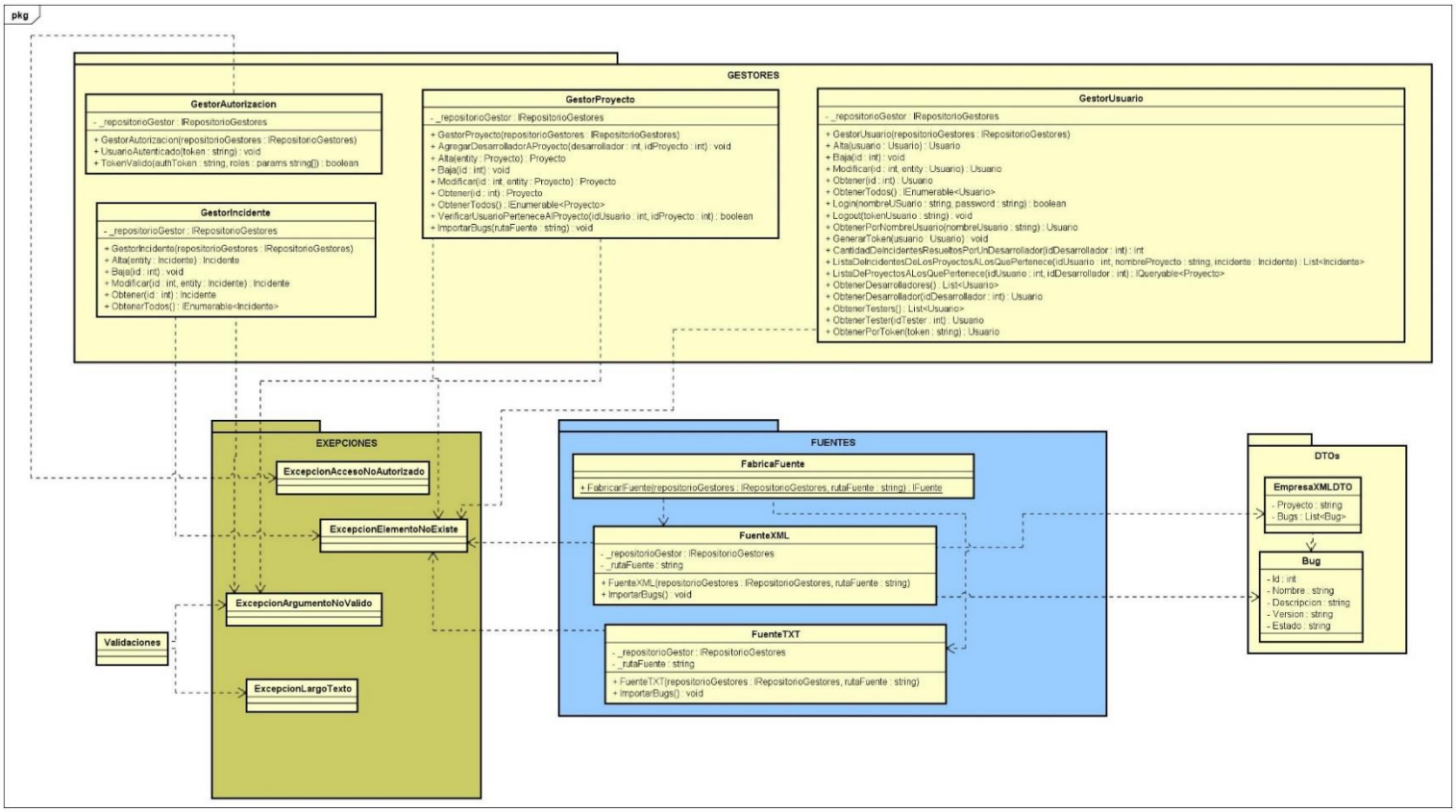
1.3.1 Dominio

Contiene las entidades más básicas del proyecto que se identificaron del problema a resolver.



1.3.2 Lógica

En este paquete se encuentran los gestores de cada repositorio, la implementación de las interfaces de la capa ILogicaInterfaz y las excepciones. También se encuentra la carpeta fuentes, que contiene las clases necesarias para poder importar Incidentes. Se trata de implementaciones de la interfaz IFuente, donde las clases FuenteXML y FuenteTXT desarrollan la lógica para ejecutar las importaciones tanto de archivos TXT o XML. Además, a efecto de mantenerse abiertos al cambio y para prever futuros nuevos formatos, se crea una fábrica de IFuente, siendo ésta quien implementa la creación de las clases necesarias por la lógica.



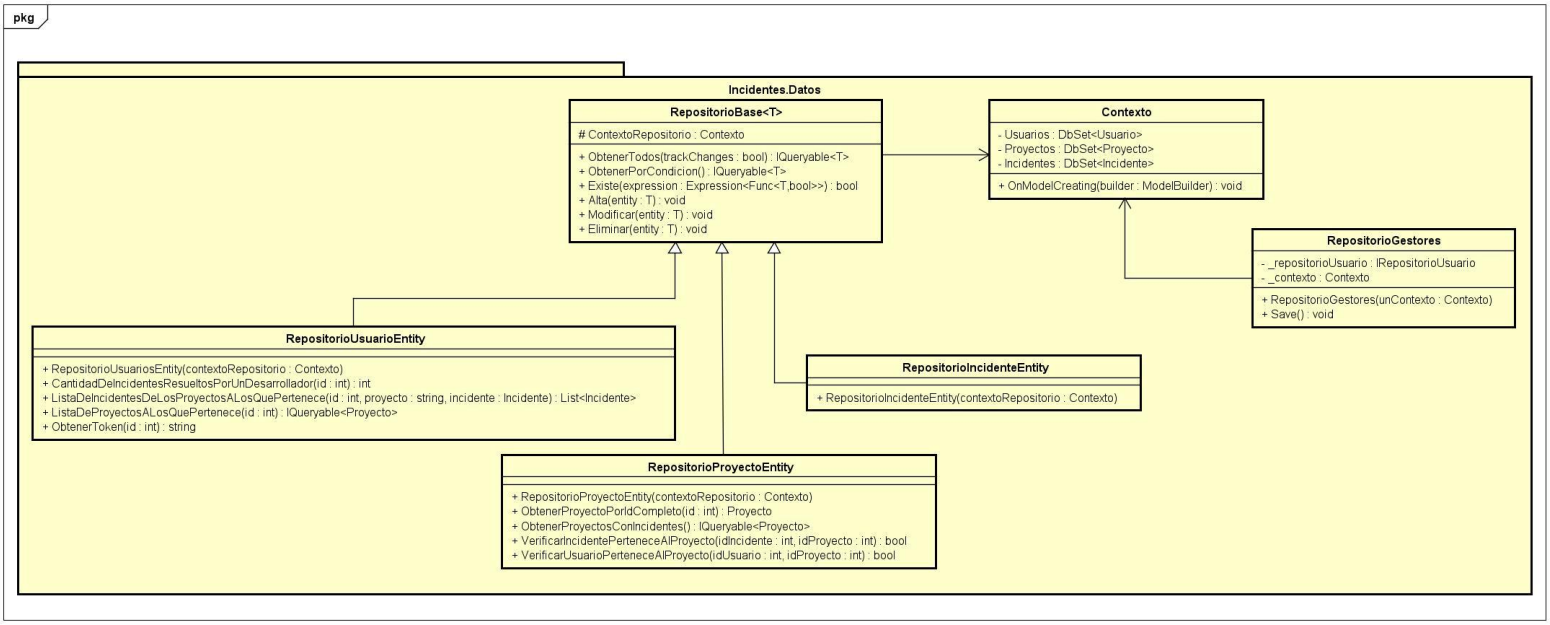
1.3.3 LogicaFabrica y LogicaInterfaz

El paquete lógica interfaz puntualmente provee un conjunto de interfaces que serán implementadas por clases del paquete de lógica, y utilizadas a más alto nivel con el objetivo de evitar acoplamiento e invertir dependencias.

El paquete LogicaFabrica tiene como responsabilidad hacer la inyección de dependencia entre la capa de lógica y la WEB API así de esta manera logramos aplicar el principio de inversión de dependencias haciendo que un módulo de alto nivel NO dependa de un módulo de más bajo nivel.

1.3.4 Datos

Este paquete tiene la responsabilidad de implementar las funcionalidades para interactuar con la BD, acoplándose con EntityFrameworkCore. Como podemos ver en el siguiente diagrama, tenemos los repositorios necesarios para trabajar con la base de datos:



1.3.5 DatosFabrica y DatosInterfaz

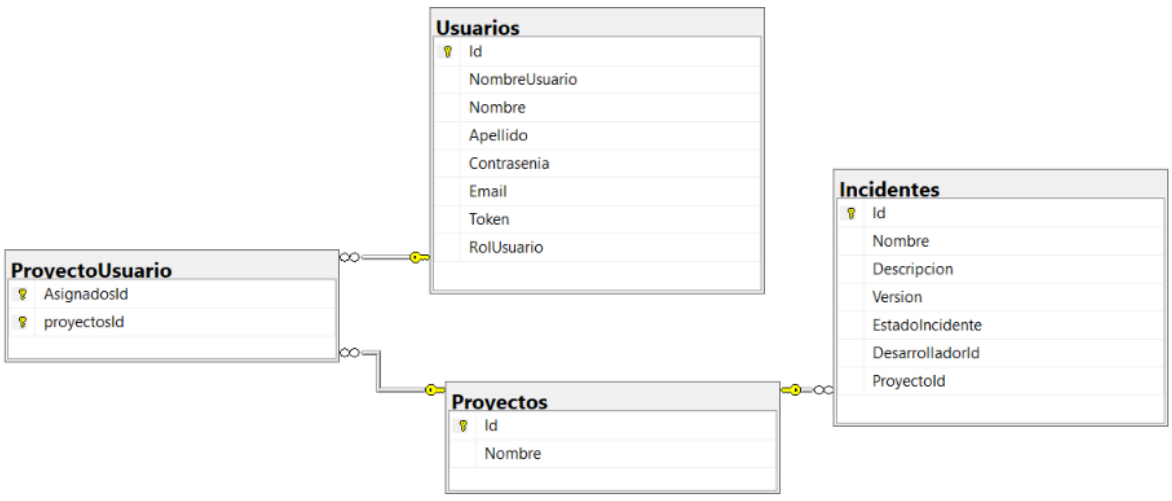
El paquete DatosInterfaz provee una serie de interfaces de repositorios, las cuales serán implementadas en el paquete de Datos.

El paquete DatosFabrica tiene una clase llamada FabricaServiciosDatos, que tiene como responsabilidad hacer la inyección de dependencias antes mencionada, y además tener un método para obtener la cadena de conexión sql a la base de datos.

Estos paquetes tienen como responsabilidad hacer la inyección de dependencia entre la web API y la capa de acceso a datos logrando que el módulo de la web API no tenga dependencias con la capa de acceso a datos ya que es aquí donde hacemos agregamos el contexto y la cadena de conexión, quitando esta responsabilidad del Startup de la web API y así pudiendo ser extensible a cualquier otra plataforma.

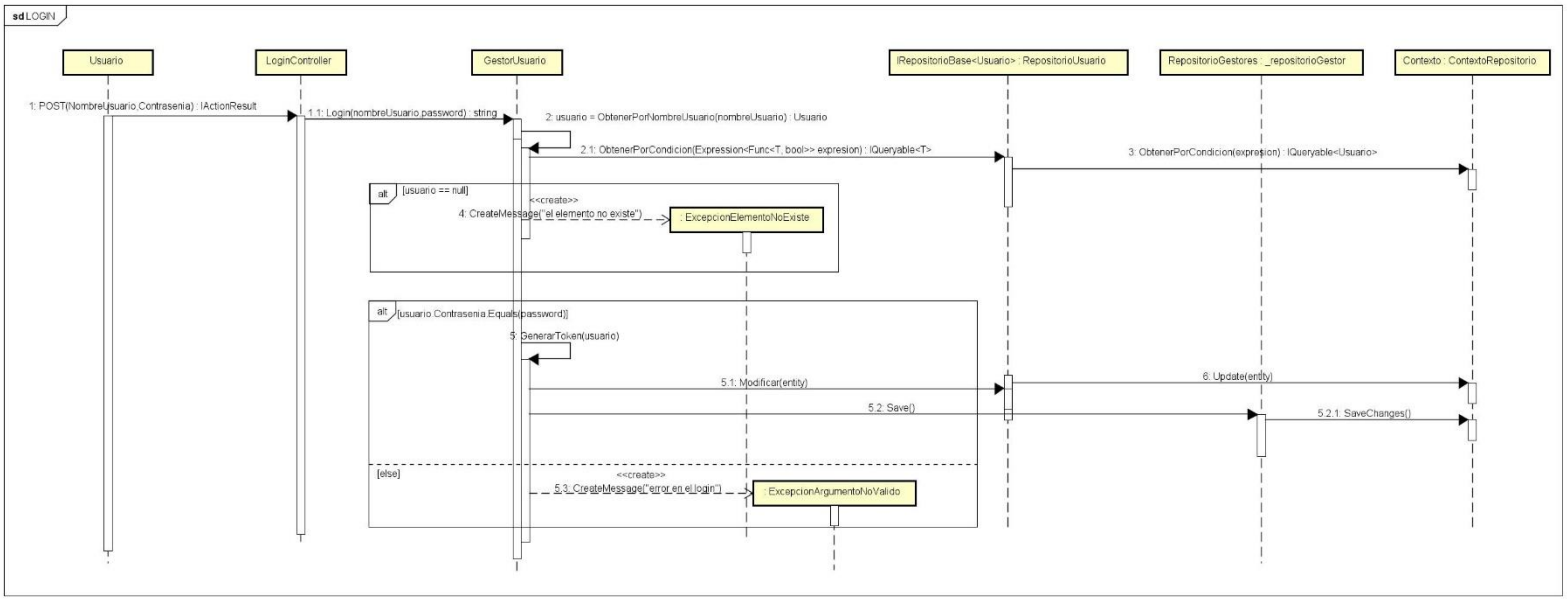
1.4 Modelo de tablas de la estructura de la base de datos.

La base de datos representa los objetos del domino del sistema, la relación N a N entre usuarios y proyectos y la relación 1 a N entre incidentes y proyectos.

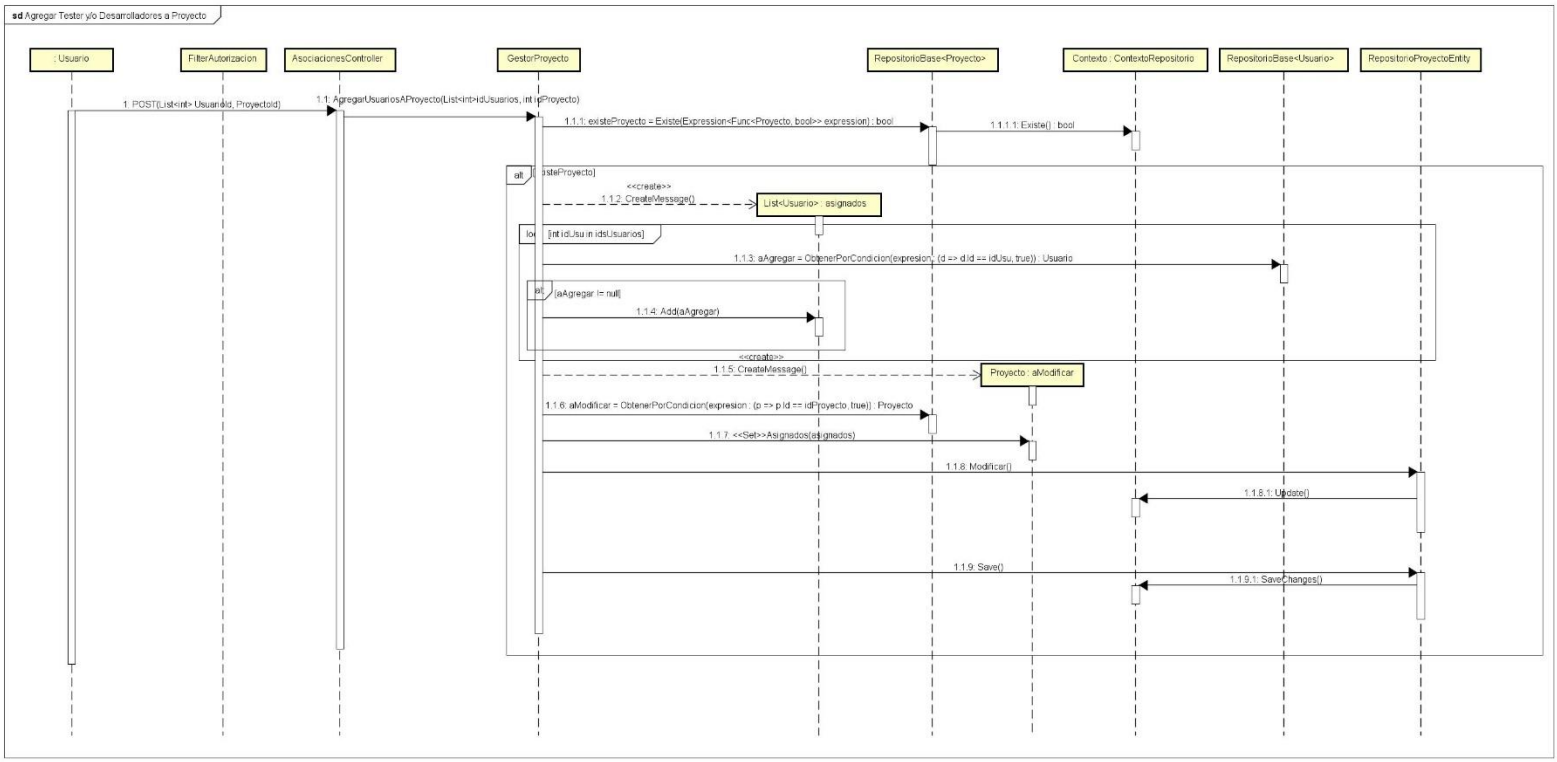


1.5 Diagrama de interacción relevantes

LOGIN



AGREGAR DESARROLLADORES Y TESTER A PROYECTO



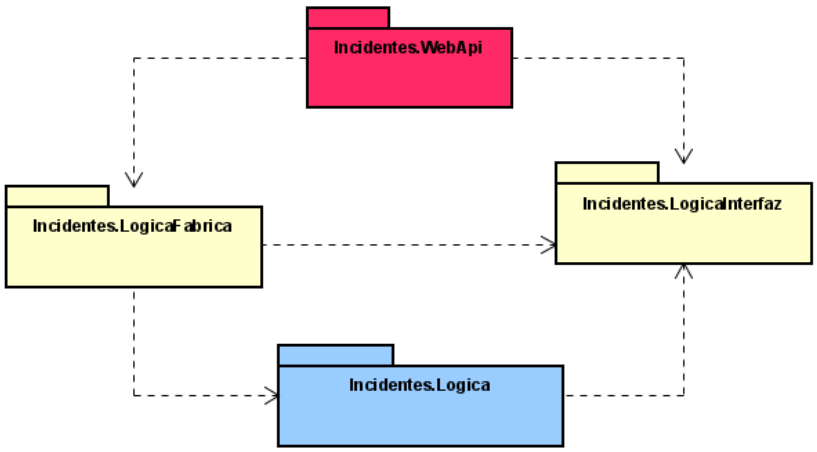
1.6 Justificación del diseño

1.6.1 Mecanismos de inyección de dependencias, fábricas, patrones y principios de diseño.

En los puntos 1.3.3 y 1.3.5 se mencionaron las inyecciones de dependencia que fueron utilizadas entre los paquetes WebApi, Lógica y la capa de Datos. Además, se crean las fabricas correspondientes para que haya total independencia entre estos módulos principales.

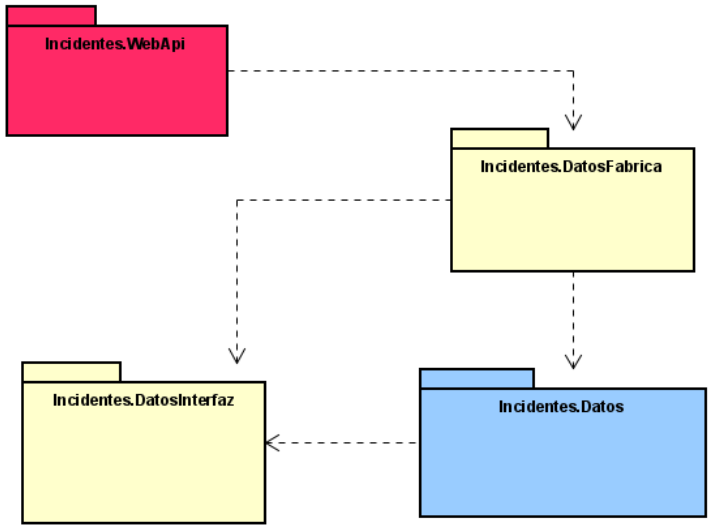
De esta manera logramos que un componente reciba sus dependencias en lugar de instanciarlas para evitar violar SRP, el acoplamiento se da entre interfaces evitando violar OCP y al no revertir las dependencias evitamos violar DIP.

Inyección de dependencia y fabricas para desacoplar de lógica y WebAPI




```
namespace Incidentes.LogicaFabrica
{
    public class FabricaServicios
    {
        private readonly IServiceCollection services;
        public FabricaServicios(IServiceCollection services)
        {
            this.services = services;
        }
        public void AgregarServicios()
        {
            services.AddScoped<ILogicaUsuario, GestorUsuario>();
            services.AddScoped<ILogicaProyecto, GestorProyecto>();
            services.AddScoped<ILogicaIncidente, GestorIncidente>();
            services.AddScoped<ILogicaAutorizacion, GestorAutorizacion>();
        }
    }
}
```

Inyección de dependencia y fabricas para desacoplar de datos y WebAPI




```
namespace Incidentes.DatosFabrica
{
    public class FabricaServiciosDatos
    {
        private readonly IServiceCollection services;
        public FabricaServiciosDatos(IServiceCollection services)
        {
            this.services = services;
        }
        public void AgregarServicios()
        {
            services.AddScoped<IRepositorioGestores, RepositorioGestores>();

        }
        public void AgregarContextoDatos()
        {
            string directory = Directory.GetCurrentDirectory();
            IConfigurationRoot configuration = new ConfigurationBuilder()
            .SetBasePath(directory)
            .AddJsonFile("appsettings.json")
            .Build();
            var connectionString = configuration.GetConnectionString(@"sqlConnection");

            services.AddDbContext<Contexto>(opts =>
            opts.UseSqlServer(connectionString,
            b => b.MigrationsAssembly("Incidentes.Datos")));
        }
    }
}
```

Se utilizó SINGLETON para la creación de los repositorios:

- RepositorioUsuario
- RepositorioProyecto
- RepositorioIncidentes

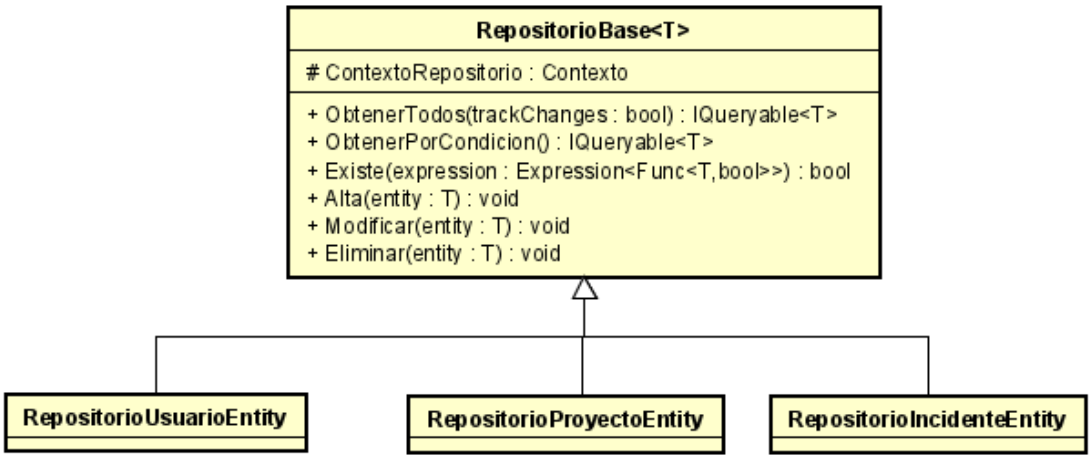
```
namespace Incidentes.Datos
{
    public class RepositorioGestores : IRepositorioGestores
    {
        private Contexto _contexto;
        private IRepositorioUsuario _repositorioUsuario;
        private IRepositorioProyecto _repositorioProyecto;
        private IRepositorioIncidente _repositorioIncidente;

        public RepositorioGestores(Contexto unContexto)
        {
            _contexto = unContexto;
        }

        public IRepositorioUsuario RepositorioUsuario
        {
            get
            {
                if (_repositorioUsuario == null)
                    _repositorioUsuario = new RepositorioUsuariosEntity(_contexto);

                return _repositorioUsuario;
            }
        }
```

Se utilizó Herencia en conjunto con la implementación de Generics <T> con el objetivo de reutilizar código:



1.6.2 Descripción del mecanismo de acceso a datos utilizado

La persistencia de datos fue realizada utilizando Entity Framework code FIRST, de esta manera definimos las clases mediante código y luego EF se encargó de generar la base de datos y todo lo necesario para mapear nuestros objetos a las tablas creadas.

1.6.3 Descripción del manejo de excepciones

Para el manejo de errores creamos diferentes clases de excepciones. En el paquete de lógica como vimos en el diagrama en el punto 1.3.2 capturamos las diversas excepciones que se pueden generar durante la ejecución del programa, y luego lanzamos las mismas al paquete de la API donde son controladas y mostradas de manera clara para que el usuario comprenda que ocurrió con la información necesaria.

1.6.4 Decisiones de diseño propias

Utilizando como base la letra del obligatorio y las consultas del foro se tomaron las siguientes decisiones:

1- Para los usuarios se creó una clase **Usuario**, y cada uno de los roles existentes (administrador, desarrollador, tester) se guarda en el atributo Rol. Al principio del proyecto lo habíamos separados en clases que heredaban de Usuario, pero a medida que fuimos avanzando nos dimos cuenta de que las distintas clases concretas no tenían atributos tan específicos a cada clase, sino que la diferencia es según el rol a que funcionalidades pueden acceder por lo que no justificaba hacer la herencia.

2- El usuario administrador se cargó directamente en la base de datos ya que necesitamos un usuario previamente registrado para poder ejecutar las funcionalidades.

3- Para actualizar los desarrolladores y/o testers correspondientes a un proyecto, se debe enviar todo el listado en una sola acción, es decir por ejemplo que, si tenemos 5 desarrolladores asociados al proyecto y deseamos agregar 1 desarrollador más, debemos incluir los que ya se encontraban antes asociados. Se piensa que a futuro cuando se tenga un front end, en la funcionalidad de update, los que ya se encuentren asociados al proyecto van a estar en la vista del usuario ya seleccionados.

4- Los incidentes pueden tener dos estados: activos o resueltos.

5- Se creó una clase **Validaciones** con métodos estáticos, que sirve para validar atributos de diferentes clases con las siguientes validaciones:

- Validar largos de textos.
- Validar passwords.
- Validar emails.

Si bien se tiene en cuenta que no es la mejor opción tener una clase estática, se provee eliminar en la futura entrega.

6- Cuando un desarrollador cambia el estado de un incidente a resuelto, automáticamente se la asigna a dicho desarrollador el atributo Desarrollador Id**.**

7- Para utilizar la funcionalidad de importar incidentes como no se aclaró que rol tiene permitido usar este recurso se decide que puede ser público para el caso que lo quiera importar una empresa.

1.7 Diagrama de componentes