# GESTIÓN DE UN SERVIDOR DE IDENTIDAD Y SU RELACIÓN CON LOS USUARIOS

David Navío Torrecilla

**Abstract** — The objective of this project is to make a management application of an identity server to improve security and access to different services of the traceability company ZETES. This is achieved through an API made in the Microsoft .NET framework and the use of the BackOffice and customer identities of the company, of which we need to manage their permissions and provide them with security.

**Index Terms** — ZETES, security, traceability, supply chain, innovation, user management, token management, API, .NET, SQL, C#.

**Resumen** — El objetivo de este trabajo es realizar una aplicación de gestión de un servidor de identidad para mejorar la seguridad y acceso a distintos servicios de la empresa de trazabilidad ZETES. Esto se consigue mediante una API realizada en el framework de Microsoft .NET y la utilización del BackOffice y la base de datos de identidades de clientes de la empresa, de los cuales necesitamos gestionar sus permisos y brindarles seguridad.

**Palabras clave** — ZETES, seguridad, trazabilidad, cadena de suministro, innovación, gestión de usuarios, gestión de tokens, API, .NET, SQL, C#.

## 1 INTRODUCCIÓN

ZETES [1] es una empresa de trazabilidad fundada en 1984 ubicada en 22 países. Desarrolla e implementa sistemas automáticos de identificación y captura datos para diferentes etapas de la cadena de suministro: fabricación, almacenamiento, transporte y logística.


*Figura 1 Diagrama de los distintos servicios y fases de la cadena de suministro en los que ZETES está presente*

La empresa abarca grandes mercados como farmacología, automoción, tabaco y alimentación. Actualmente da soporte al 80% de las compañias de cadena de suministro.

ZETES ayuda a maximizar la agilidad, trazabilidad y visiblidad a través de soluciones otorgadas a los clientes. Los cuales necesitan credenciales seguras y la garantía de que los datos de sus productos se están manipulando con la máxima rigurorisdad y fiabilidad.


- *E-mail de contacte: 1463009@uab.cat*
- *Menció realitzada: Enginyeria de Software*
- *Treball tutoritzat per: Helena Bolta Torrell (Departament de Ciències de la Computació)*
- *Curs 2021/22*


## 2 ESTADO DEL ARTE

En el mundo globalizado [2] y descentralizado en el que vivimos, los productos viajan por todo el mundo. Gracias a este suceso, los procesos de logística y control han cobrado un papel fundamental para que todo funcione de forma óptima, aportando valor tanto a los proveedores de dichos productos como a consumidores finales.

El propósito de la trazabilidad en la cadena de suministro es otorgar información clara de un producto, desde su origen hasta el consumidor final, aportando seguridad en la gestión de riesgos y eficiencia en los procesos.

Para aportar seguridad y confianza en este proceso, es necesario ofrecer garantias y facilidades a los usuarios. Como se ha comentado anteriormente, ZETES trabaja con un gran volumen de clientes, con lo cual necesitamos gestionar de forma ágil y segura una cantidad amplia de datos.

ZETES actualmente utiliza un servidor de identidad basado en OAuth para proteger el acceso a sus aplicaciones. Necesita una aplicación para gestionar las distintas configuraciones del servidor ya que actualmente se tienen que realizar insertando datos directamente en la base de datos con SQL, lo cual es poco óptimo y es sucesceptive a errores humanos ya que no hay lógica que valide que estas modificaciones sean correctas.

Ante este reto, se ha propuesto realizar una API que sea capaz de gestionar esta gran cantidad de datos aportando seguridad, comodidad y mejoras en el rendimiento de esta gestión.

## 3 OBJETIVOS

Se requiere realizar una API capaz de realizar cambios en las bases de datos que contienen la información del gestor de identidad para que después pueda recibir llamadas de una interfaz web desde donde se aplicarán las diferentes configuraciones cubriendo los siguientes **objetivos:**

- Gestión de acceso a la aplicación y permisos de sus usuarios.
- Edición y listado de los usuarios guardados en el servidor de identidad.
- Guardado y gestión de tokens persistentes asociados a los clientes que generen este tipo de tokens.
- Gestión de nuevos clientes en el servidor y su asociación con aplicaciones de BackOffice.
- Gestión de APIResources y APIScopes dentro del dominio del servidor de identidad.

## 4 PLANIFICACIÓN Y METODOLOGÍA

Para cubrir esas funcionalidades, necesitaremos llevar a cabo las siguientes **tareas**:

- Análisis de requisitos.
- Diseño de la arquitectura a nivel de infraestructura.
- Implementación de las principales operaciones para interactuar con los registros de la Base de Datos mediante la API.
- Diseño de políticas de usuarios, control de excepciones y acceso a aplicaciones.
- Realización de tests de integración para las funciones principales.
- Despliegue y pruebas en entorno de Quality Assurance
- Documentación.

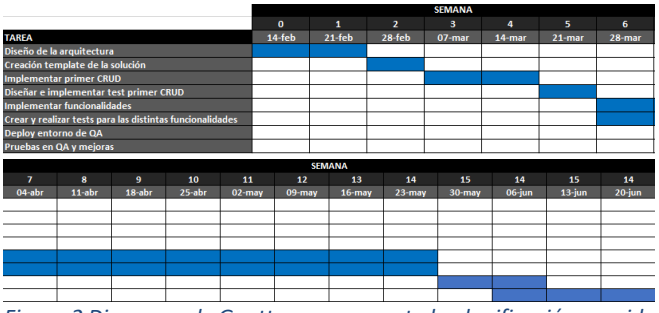


*Figura 2 Diagrama de Gantt que representa la planificación seguida en la realización del proyecto.*

**Características** definidas:

- Realizado mediante NET Core 6.0.
- El lenguaje de programación es C#.
- A parte de la programación, es necesario interactuar con Bases de Datos, con lo cual se utiliza SQL.
- Microsoft Azure para llevar el control de sprints, tareas y versiones.

Se ha utilizado una metodología ágil [3]. Mediante Azure DevOps se ha llevado el control de los sprints semanales, definiendo las tareas y haciendo aproximaciones más precisas sobre el tiempo requerido para conseguir cada objetivo.

Gracias al control de versiones [4], mediante la herramienta GitHub, se ha llevado un buen seguimiento de los avances, ya que se ha podido reproducir y consultar versiones de código anteriores siempre que se requiera, así como garantizar que los avances están guardados en servicios y no únicamente en local.

Las reuniones semanales con el tutor de empresa, una por sprint, han facilitado la resolución de dudas, sobre todo en el tema de la calidad de la solución, ya que la experiencia del tutor ha servido como guía en cuanto a buenas prácticas y optimización del código para obtener los objetivos planteados.

La documentación constante ha facilitado mucho la comprensión de los avances. Ya que al reportarlo se asimilan mejor los conceptos y permite que en un futuro cualquier persona con los conocimientos necesarios pueda entender el funcionamiento de la aplicación en el menor tiempo posible.

## 5 TAREAS REALIZADAS

### 5.1 Diseño de la arquitectura

La aplicación se divide en la parte de **BackOffice**, donde tendremos una Base de Datos separada y gestionaremos las diferentes Applicaciones y *Tenants*. Esta interactuará con la parte de **IdentityServer**, que tendrá otra Base de Datos. En esta parte tendremos los *clientes*, *tokens*, *Api Resources* e *Identity Resources*. Este diseño se puede ver en el anexo A.1.

A la hora de diseñar la aplicación, hemos dividido las diferentes funcionalidades en pequeñas partes, para facilitar la implementación y poder escalar la dificultad.

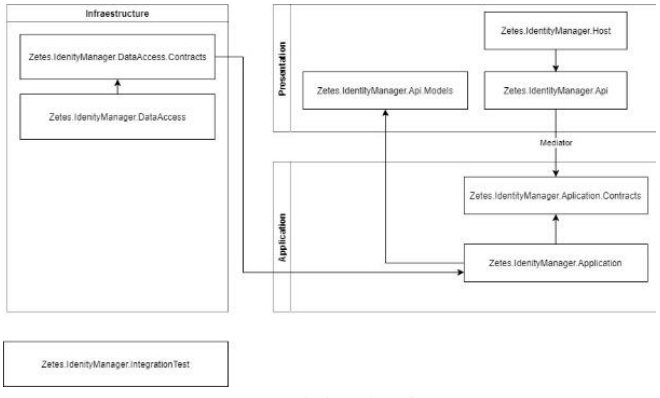


*Figura 3 Diagrama proyectos de la solución*

He dividido la solución en tres capas diferenciadas. El objetivo ha sido poder dividir la solución en partes diferenciadas, favoreciendo así su comprensión y buenas prácticas [5] de cara a este proyecto y futuras modificaciones.

Tenemos la capa ***Presentation*** donde se levantará el servidor, se harán las conexiones a las bases de datos y se levantará la API, con sus respectivos controladores. Estos controladores

necesitarán una lógica y realizar unas acciones concretas, personalizadas para cada controlador. Eso lo haremos en la capa ***Application***.

Como se puede ver en el diseño, conectaremos la capa de *Presentation* y sus controladores con la capa de *Application* mediante *Mediator* [6].

Por último, tendremos la capa ***Infraestructure***, conectada a Application. Donde Application tendrá las entidades y los repositorios para realizar las acciones necesarias.

De esta forma, tenemos tres capas diferenciadas, aplicando así buenas prácticas de diseño de software.

Fuera de estas capas tenemos un proyecto de tests de integración. Donde haré como mínimo un test para cada controlador, replicando su funcionamiento y comprobando que obtenemos el resultado esperado.

## 5.2 Creación template de la solución

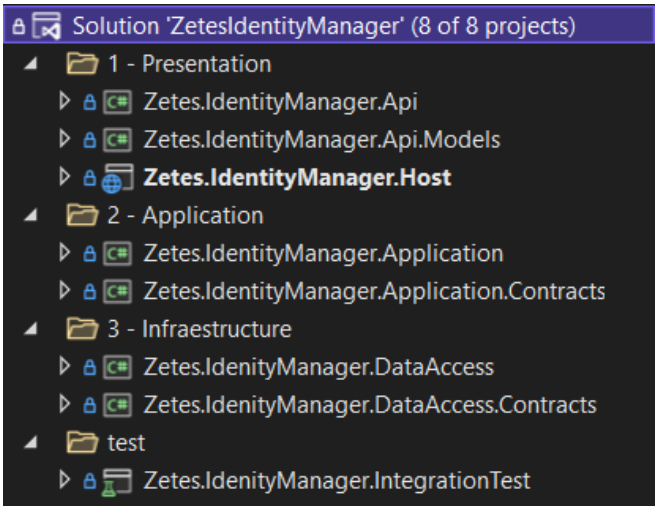


*Figura 4 Template de la solución en Visual Studio 2022*

### 1. Presentation

**Zetes.IdentityManager.Api:**
Contiene los controladores y la configuración de los servicios necesarios para ejecutar la API (clase de inicio, *API net core* [7], *Swagger*[8] y configuración de inyección de dependencia).
**Zetes.IdentityManager.Api.Models:**
*Requests* y *responses* necesarias para hacer las diferentes peticiones en los controladores de la API.
**Zetes.IdentityManager.Host:**
Se inicializa la base de datos, se crea el Web Host y se inicializan servicios, repositorios y *appsettings*.

### 2. Application

Zetes.IdentityManager.Application:
Se definen los diferentes *commands* y *queries*, donde tendremos un *handler* para gestionar las diferentes llamadas por parte de los controladores. También los *mappers* para mapear las diferentes respuestas de la API y los *servicios*, que harán las diferentes operaciones en la Base de Datos a través de los repositorios.
**Zetes.IdentityManager.Application.Contracts:**
*Interfaz* de los diferentes servicios.

### 3. Infraestructure

**Zetes.IdentityManager.DataAccess:**
Aquí tendremos los diferentes repositorios y la configuración de las entidades. A parte, el contexto de la base de datos, donde definiremos la base de datos y las diferentes entidades de esta.
**Zetes.IdentityManager.DataAccess.Contracts:**
Interfaz de los repositorios y del contexto.

### 4. Test

**Zetes.IdentityManager.IntegrationTests:**
Creación e inicialización de una base de datos únicamente de testing. Generadores de datos y Seed. Así como los diferentes tests para abarcar toda la aplicación.

## 5.3 Implementación CRUD Applications

Una vez ya tenemos un diseño de la aplicación y un esqueleto definido que seguir, empezamos a codificar e implementar los diferentes objetivos propuestos.

Después de analizarlo detalladamente, he decidido empezar haciendo un CRUD básico, viendo que este se mueve correctamente por todas las capas y partes del proyecto y que funciona de la forma esperada, en este caso el CRUD de *Applications*:

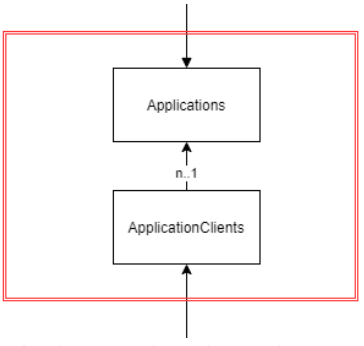


*Figura 5 Relación clases involucradas en el CRUD de Applications*

Una vez se tenga este CRUD finalizado se testeará de forma adecuada y a partir de ahí ya se podrán seguir implementando las diferentes funcionalidades utilizando las mismas prácticas y formas de programar aplicadas en este CRUD, consiguiendo así un mejor entendimiento de código tanto para mí como para las diferentes personas que puedan utilizarlo en el futuro.

Este lo levantaremos en local, definiendo el puerto en el *launchsettings* del proyecto Zetes.IdentityManager.Host. Posteriormente, se levantará en DEV, QA y PROD en último término.

Ejemplo recorrido de uno de *endpoints* UPDATE Application (*POST /api/applications):*

1. **Controlador:** Primero de todo se comprueba que el modelo sea válido, se genera un nuevo comando donde guardaremos todos los datos, en este caso pasados por el body, que se utilizan para realizar la operación. Se envía mediante mediator el comando y este será recogido por el handler en el siguiente paso.

2. **Handler**: Conectado mediante mediator, el programa entrará aquí una vez haga *await _mediator.Send(command)* y realizará las comprobaciones pertinentes y rellenará los objetos que utilizamos para realizar distintas operaciones en la base de datos (Add, Update, Get, Remove,…) y mediante *unitOfWork* guardaremos los cambios.

3. **Servicio**: Realizamos las distintas operaciones en el servicio, que se concentrarán posteriormente con los repositorios de cada entidad.

4. **Repositorio**: Mediante Entity Framework y Linq, estos métodos ya están implementados, mediante los diferentes Nuggets hacemos uso de estos.

### 5.4 Diseño e implementación tests integración

Para realizar el testing se levanta un Web Host y una nueva Base de Datos, únicamente para test. En este se podrán realizar las mismas operaciones, de forma que podemos comprobar el comportamiento del programa de la misma forma.
En este caso he realizado un camino que cree, lea, actualice y elimine una entidad de la Base de Datos, haciendo pruebas en cada una de las partes:

```
[Fact]
0 references | 0 changes | 0 authors, 0 changes
public async Task CreateReadUpdateAndDeleteApplicationShould()
{
    var application = await CreateApplicationShould();

    await ReadApplicationShould(application);

    application = await UpdateApplicationShould(application);

    await DeleteApplicationShould(application);
}
```

*Código 1 Flujo de métodos invocados para realizar testear las operaciones de Applications*

Para realizar cada una de las pruebas siempre se sigue la misma lógica.

1. Se crear un *request* para ese *endpoint* en concreto.
2. Se genera un cliente.
3. Se realiza la petición a la API, que cambia según la operación que queramos hacer.
4. Se recibe la respuesta y se comprueba que sea válida.
5. Se hacen las diferentes comprobaciones. Por ejemplo, para un *Create* se comprueban que los valores de la respuesta son iguales que los valores de la *request*.

*Ejemplo CreateApplicationShould:*

```
var client = Server.CreateClient();
var response = await client.Post(Routes.Applications.Create(), request);

await response.EnsureSuccessStatusWithContentAsync();

var result = await response.ResultAsAsync<ApplicationResult>();

result.Should().NotBeNull();
result.Name.Should().Be(request.Name);
result.CreateDateTime.Should().Be(request.CreateDateTime);
result.IsDeleted.Should().BeFalse();
result.UpdateDateTime.Should().BeNull();
result.UpdateUserId.Should().BeNull();

return result;
```

*Código 2 Comprobaciones realizadas valor a valor para verificar que contienen los valores esperados.*

Para redireccionar y generar las peticiones a los *endpoints* de la API utilizamos:

```
var response = await client.Post(Routes.Applications.Create(), request);
```

*Código 3 Definición de la ruta a seguir al realizar un POST*

Donde dentro de *Routes* se indica la ruta de cada *endpoint*:

```
1 reference | David Navio, 18 days ago | 1 author, 2 changes
internal static string Root() => "/api";

4 references | David Navio, 18 days ago | 1 author, 3 changes
internal static class Applications
{
    4 references | David Navio, 34 days ago | 1 author, 1 change
    internal static string Base() => Root() + "/applications";

    1 reference | David Navio, 18 days ago | 1 author, 2 changes
    internal static string Create() => Base();
```

*Código 4 Fragmento donde se muestra la ruta desglosada para hacer un POST*

Estas rutas corresponden a las mismas rutas que tenemos para realizar las diferentes operaciones:

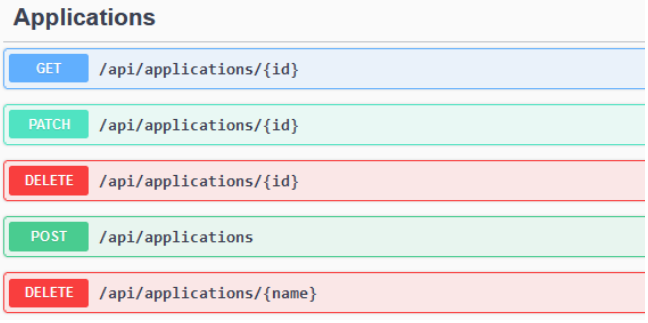


*Figura 6 Visión de las rutas de los endpoints realizados en Applications*

De cara al futuro y a siguientes implementaciones será necesario pasar todos los tests de integración antes de hacer una pull request, ya que quiero asegurarme de que no se ha roto ninguna parte del código al hacer cualquier modificación y sigue funcionando de la forma esperada.

### 5.5 Implementación CRUD Tenants

Una vez realizado el CRUD para *Applications* procederemos a realizar el CRUD para *Tenants*, el cual sigue interactuando con la Base de datos de **Back Office**. Se ha realizado un controlador que interactúa correctamente un mayor número de datos que en *Applications*, lo cual radica en una subida de la dificultad.

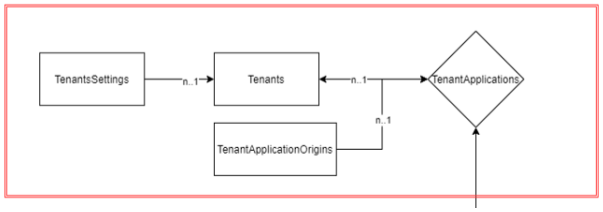


*Figura 7 Relación clases involucradas en el CRUD de Tenants*

*La tabla TenantsSettings se modificará manualmente en la base de datos, con lo cual no es necesario incluirla en el desarrollo de la API.* Como podemos ver en el diagrama anterior, *Tenants* será la clase principal, la cual mantendrá una relación con *Applications* mediante la tabla *TenantApplications* y que tiene a *TenantsSettings* y *TenantApplicationOrigins* dependiendo de la misma.

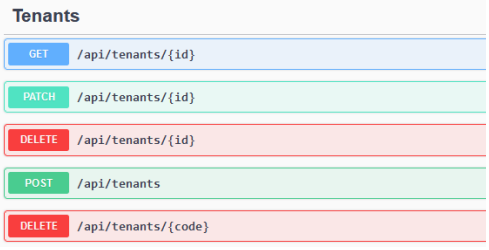


*Figura 8 Visión de los endpoints del CRUD de Tenants*

Mediante un único CRUD queremos llevar un control y poder realizar las operaciones básicas en todas las tablas dependientes, propagando a partir de *Tenants*.

Por ejemplo, para añadir un Tenant mediante la llamada **POST /api/tenants** se enviará lo siguiente:

```
public class AddTenantRequest
{
    [Required]
    public string Name { get; set; }

    public string? Code { get; set; }
    public int MaxAccessFailed { get; set; }
    public string? RegexPassword { get; set; }
    public string? UserBlockedReturnUrl { get; set; }
    public bool IsExternal { get; set; }
    public string? ValidatorName { get; set; }
    public int? DaysExpirationPassword { get; set; }
    public string? InvalidPasswordFormatMessage { get; set; }
    public int? NumberPasswordsRemembered { get; set; }
    public List<TenantApplicationOriginRequest>? TenantApplicationOrigin { get; set; }
    public List<TenantApplicationRequest>? TenantApplication { get; set; }
```

*Código 5 Ejemplo Request utilizada en la API*

Esta request permite informar y añadir objetos a las distintas tablas que dependen de Tenants mediante un único endpoint. En cuanto a Update la request será similar.

Delete borrará manteniendo la jerarquía, de menor a mayor, borrando por último el objeto de la tabla *Tenants*, como se puede ver a continuación:

*Código 6 Flujo eliminación datos mediante DELETE Tenants.*

```
public async Task<TenantResult> Handle(DeleteTenantCommand request, CancellationToken cancellationToken)
{
    var tenant = await _tenantsService.Get(request.Name);

    if (tenant == null)
        throw new BusinessRuleException($"Tenant not found", ErrorCodes.TenantNotFound);

1   var tenantApplicationOrigins = _tenantApplicationOriginsService.GetAllByTenantApplicationId(tenant.Id);
    foreach (var tenantApplicationOrigin in await tenantApplicationOrigins)
        _tenantApplicationOriginsService.Remove(tenantApplicationOrigin);

2   var tenantApplications = _tenantApplicationsService.GetAllByTenantId(tenant.Id);
    foreach (var tenantApplication in await tenantApplications)
        _tenantApplicationsService.Remove(tenantApplication);

3   _tenantsService.Remove(tenant);

    await _unitOfWork.SaveChanges(cancellationToken);
```

La comunicación entre el controlador de *Tenants* y la lógica de la aplicación se hará también mediante *MediatR*, manteniendo una estructura coherente con el desarrollo de *Applications*, facilitando así la comprensión del código:

```
[HttpDelete("{name}")]
[ProducesResponseType(typeof(bool), 200)]
0 references | David Navio, 26 days ago | 1 author, 2 changes
public async Task<IActionResult> DeleteByName([Required] string name)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    var command = new DeleteTenantCommand(name: name);

    await _mediator.Send(command);

    return Ok(true);
}
```

*Código 7 Controlador el cual se comunica con el Handler, visto en el código anterior.*

Para comprobar el correcto funcionamiento del CRUD, realizaremos tests que simulen el comportamiento lógico del mismo. Primero creando un Tenant, seguido de una recuperación del mismo, una modificación y, por último, una eliminación:

```
public CRUDTenantShould(TestFixture fixture) : base(fixture)   { }

[Fact]
0 references | David Navio, 28 days ago | 1 author, 1 change
public async Task CreateReadUpdateAndDeleteTenantShould()
{
    var Tenant = await CreateTenantShould();
    await ReadTenantShould(Tenant);
    Tenant = await UpdateTenantShould(Tenant);
    await DeleteTenantShould(Tenant);
}
```

*Código 8 Flujo de métodos invocados para realizar testear las operaciones de Tenants*

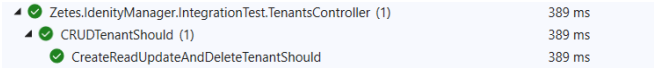


*Figura 9 Tests de Tenants pasados con resultado satisfactorio*

## 5.6 Implementación CRUD Clients

La implementación del CRUD de Clientes supone un reto mayor ya que se interactúa con un número mayor de entidades y mediante un contexto diferente. En vez de utilizar la Base de Datos de Back Office utilizaremos la de **Identity Server**.

A parte de modificar Clientes y sus tablas, también se propagará a *ApiResources* e *IdentityResources*. De forma que podamos operar con todas las tablas de la API mediante este CRUD.

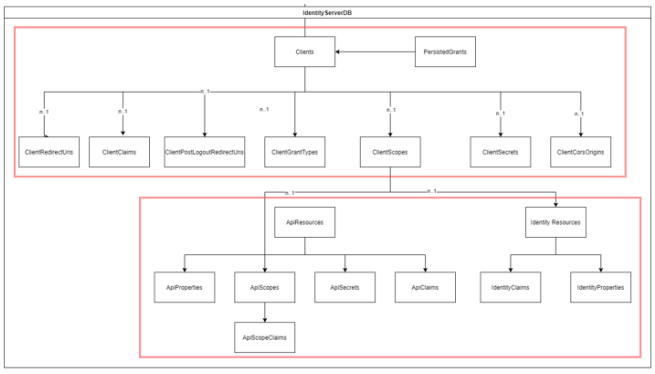


*Figura 10 Relación clases involucradas en el CRUD de Clients*

Como se observa en el diagrama, en este CRUD se leerán y modificarán datos de un número superior de tablas.

Esta implementación requiere una forma de proceder algo diferente a la anterior ya que Identity Server es un servidor de autenticación que proporciona una manera común de autenticar, con lo cual ciertas clases y métodos no hará falta crearlos, sino que aprovecharemos los que Identity Server [9] nos proporciona. Esto supone una ventaja a la hora de desarrollar menos código y tener que definir menos clases, pero también supone un reto ya que es necesario entender el funcionamiento de estas librerías y seguir una forma de programar más cerrada para que todo funcione correctamente.

Al tratar datos y estructuras de Identity Server necesitaremos instalar los siguientes NuGets: IdentityServer4 (4.1.2) y IdentityServer4.EntityFramework (4.1.2).

## 5.7 Autorización

Al utilizar la API queremos que solamente clientes registrados puedan tener acceso. Esto lo conseguimos mediante la autorización:

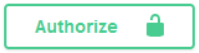


*Figura 11 Botón de autorización encontrado en el Swagger.*

La Base de Datos de Identity Server contiene los datos necesarios para poder autorizarnos en Identity Server, estos datos los obtiene gracias a la parte implementada en el CRUD de clientes, el cual modifica potencialmente todas las tablas de la Base de Datos de Identity Server.

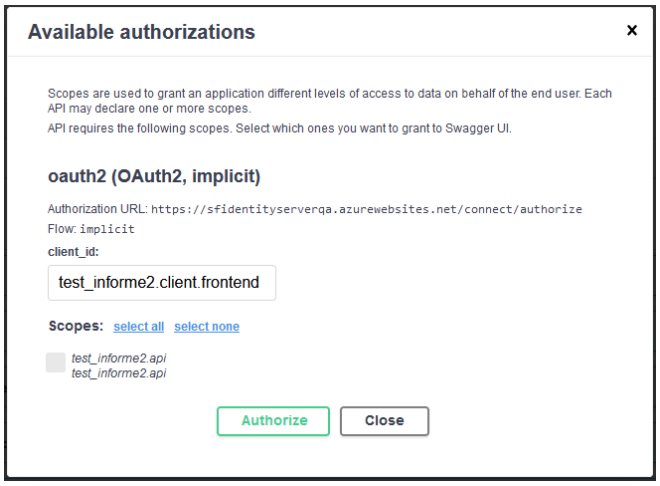


*Figura 12 Visión de la vista surgida de seleccionar el botón visto en la figura anterior.*

Utilizamos tokens JWT [10], los cuales dan acceso a los usuarios certificando su identidad. Este token se devolverá al cliente y este tendrá que enviarlo de vuelta a la API en cada una de sus peticiones.
También utilizamos Oauth 2.0 [11], el cual es un protocolo de autorización que puede utilizar tokens JWT.
Mediante OAuth 2.0 definimos el protocolo y especificamos como se transfieren los tokens y mediante JTW definimos el formato del token [12].
Todo esto lo conseguiremos gracias a añadir Autenticación y JwtBearer en la configuración a la hora de hacer el StartUp de la solución:

```
0 references | David Navio, 22 hours ago | 2 authors, 4 changes
public void ConfigureServices(IServiceCollection services)
{
    services.AddApplicationInsightsTelemetry(Configuration);
    services.AddSingleton<ITelemetryInitializer, TelemetryInitializer>();

    services.Configure<IdentityOptions>(options =>
    {
        options.ClaimsIdentity.UserNameClaimType = ZetesClaimTypes.UserName;
        options.ClaimsIdentity.RoleClaimType = ZetesClaimTypes.Role;
    });

    JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();
    services.AddAuthentication(sharedOptions =>
    {
        sharedOptions.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
        sharedOptions.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    })
    .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options =>
    {

        options.Authority = IdentityServerConfig.Url;
        options.RequireHttpsMetadata = false;
        options.Audience = IdentityServerConfig.Scope;
        options.TokenValidationParameters.NameClaimType = ZetesClaimTypes.UserName;
        options.TokenValidationParameters.RoleClaimType = ZetesClaimTypes.Role;
    });
```

*Código 9 Configuración del servicio para la utilización de Authentication y Jwt Token*

A parte, lo configuramos en la construcción del Swagger:

```
app.UseSwagger();

var locOptions = app.ApplicationServices.GetService<IOptions<RequestLocalizationOptions>>();
app.UseRequestLocalization(locOptions.Value);

app.UseSwaggerUI(c =>
{
    foreach (var apiVersion in apiVersions)
        c.SwaggerEndpoint($"{configuration["IISConfig:VirtualDirectory"]}/swagger/v{apiVersion}/swagger.json", $"Identity Manager API V{apiVersion}");

    c.OAuthClientId(swaggerClientId);
    c.OAuthUseBasicAuthenticationWithAccessCodeGrant();
    c.EnableFilter();
});

app.UseRouting();

app.UseAuthorization();
```

*Código 10 Configuración para conseguir mostrar en el Swagger la posibilidad de Autorizarse.*

## 5.8 Control de excepciones

Una parte importante del desarrollo es el control de excepciones, tanto para poder controlar los errores a la hora de desarrollar como para una futura utilización por parte de usuarios, ayudándoles a encontrar el porque de cualquier error y poder solventarlo.
La implementación realizada permite controlar excepciones de tipo *BadRequest*, *NotFound* y *BusinessRule*:

```
public void OnException(ExceptionContext context)
{
    context.ExceptionHandled = true;

    if (context.Exception is BadRequestException)
    {
        context.Result = new ObjectResult(context.Exception) {
            StatusCode = (int)HttpStatusCode.BadRequest
        };

        _log.Warning($"BadRequest {context.Exception?.Message}");
    }
    else if (context.Exception is BusinessRuleException)
    {
        var businessEx = context.Exception as BusinessRuleException;
        context.Result = new ErrorResult(businessEx);

        _log.Warning($"Conflict Code {businessEx.ExceptionCause}, Message {businessEx.Message}");
    }
    else if (context.Exception is NotFoundException)
    {
        var notFoundEx = context.Exception as NotFoundException;
        context.Result = new ErrorResult(notFoundEx);

        _log.Warning($"NotFound Code {notFoundEx.ExceptionCause}, Message {notFoundEx.Message}");
    }
    else
    {
        context.ExceptionHandled = false;
        if (_env.IsProduction())
            context.Result = new StatusCodeResult((int)HttpStatusCode.InternalServerError);
        else
            context.Result = new ObjectResult(context.Exception) { StatusCode = (int)HttpStatusCode.InternalServerError };

        _log.Error(context.Exception, context.Exception.Message);
    }
```

*Código 11 Configuración y control de excepciones*

Este mensaje se mostrará en la respuesta del endpoint y en el logger.
Siempre se mostrará un mensaje de error, un código y el número que identifica comúnmente a este error (ejemplo: 409 en conflicto):

```
public ErrorResult(NotFoundException value) : base(value)
{
    Code = value.ExceptionCause;
    Message = value.Message;

    Value = new
    {
        Code = Code,
        Message = Message,
        Data = value.AdditionalData,
        DataType = value.AdditionalData?.GetType()
    };
    StatusCode = (int)HttpStatusCode.NotFound;
}
```

```
public ErrorResult(BusinessRuleException value) : base(value)
{
    Code = value.ExceptionCause;
    Message = value.Message;

    Value = new
    {
        Code = Code,
        Message = Message,
        Data = value.AdditionalData,
        DataType = value.AdditionalData?.GetType()
    };
    StatusCode = (int) HttpStatusCode.Conflict;
}
```

*Código 12 Valores guardados y mostrados tanto para NotFoundException como para BusinessRuleException*

Dentro de los BusinessRuleExceptions se encuentran varios tipos. Estos tipos son las excepciones que he contemplado dentro de la implementación de la lógica de negocio y son los siguientes:

```
public enum ErrorCodes
{
    IdentityValidationError,
    UserNotFound,
    UserAlreadyExists,
    TenantNotFound,
    TenantAlreadyExists,
    ApplicationNotFound,
    ApplicationAlreadyExits,
    ClientNotFound,
    ClientAlreadyExists
}
```

*Código 13 Posibles errores contemplados en la API.*

Por ejemplo, si intentamos añadir un Tenant ya existente:
Se mostrará la siguiente excepción:

1. Comprobación de que este Tenant ya existe en Base de Datos:

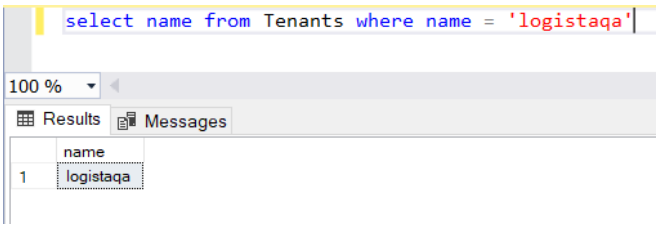


*Figura 13 Busqueda realizada mediante SQL para comprobar la existencia o no de un valor en Base de Datos.*

2. Intentamos crear un Tenant con el mismo nombre ´logistaqa'. Recibimos un mensaje de error conflicto, con el error code que le hemos especificado y el mensaje y código indicados en el handler.

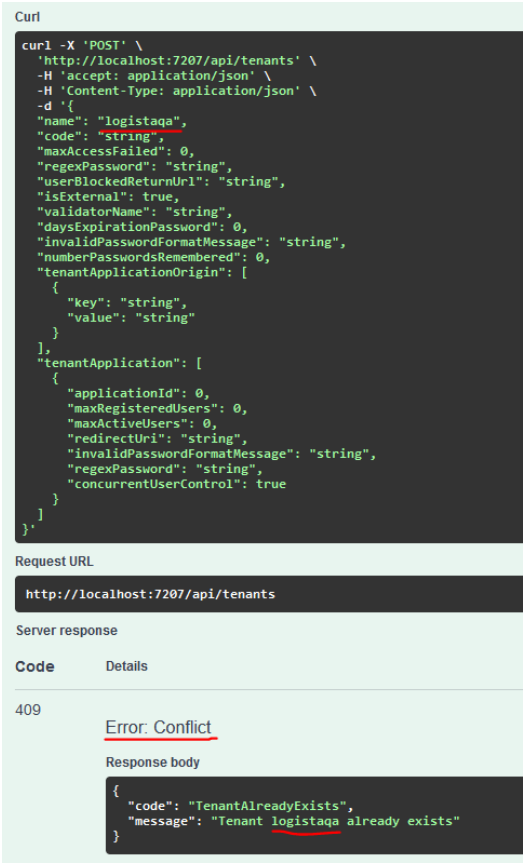


*Figura 14 Respuesta de la API al intentar realizar una operación que genera un error mostrado mediante el Swagger.*

## 5.9 Deploy en QA y mejoras

El deploy lo hemos realizado mediante Azure DevOps. Al gestionar el control de versiones mediante este ha facilitado a la hora de trabajar con Pipelines y Releases.

Mediante Pipeline hemos podido construir y crear el artefacto, el cual luego mediante la Release hemos podido implementar.

Después de realizar el Deploy en QA hemos podido probar y recibir feedback de otros desarolladores para posteriormente realizar cambios.

Algunas mejoras realizadas:

- **Cambio de PUT a PATCH:**
  En los endpoints donde se hace el UPDATE están definidos previamente como PUT, pero esto generaba confusión ya que PUT remplaza la entidad por completo, en cambio PATCH no cambia los valores que no se le envían. Este cambio se adapta mejor a la lógica implementada. [13]
- **Optimización de todos los paths de los endpoints:** Anteriormente varios endpoints recibían parámetros mediante la query. Después de realizar pruebas se ha detectado una volunerabilidad ya que expone a los usuarios el nombre de las variables. Para solucionar esto, se ha cambiado esto por path, de forma que se ocultan los nombres de las variables [14].
- **Resolución de bugs:** Al probarlo detelladamente en QA se han encontrado errores y debilidades en el código que mediante los tests de integración no se habían detectado y los cuales han sido solucionados.

## 7 PROBLEMAS SUPERADOS

La codificación necesaria para la interación con la base de datos de BackOffice no ha supuesto ningún problema ya que esta Base de Datos está realizada íntegramente por la empresa ZETES.

En cambio, el servidor y Base de Datos de Identity Server está realizado apoyándose en librerías de IdentityServer. Esto permite optimizar el código, pero a la hora de comunicarme con este mediante el Identity Manger

En cuanto a la base de datos de IdentityServer, esta está desarollada utilizando los modelos proporcionados mediante librerías de IdentityServer:

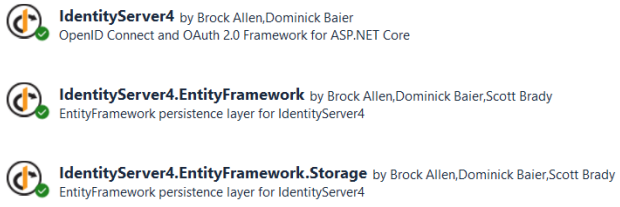


*Figura 14 Librerias de IdentityServer que ofrecen los modelos de las entidades que se insertan y modifican en Base de Datos*

Al principio, estaba intentando modificar datos de IdentityServer definiendo mis propios modelos, lo que generó un problema de compatiblidad. Después de verificar que eso era un problema tuve que volver a diseñar el CRUD de Tenants y pasar a utilizar las librerías de IdentityServer vistas en la Figura14.

Además, la configuración para levantar correctamente la API y los tests ha supuesto un gran reto ya que al interactuar con varios servidores y bases de datos era necesario definir todo con mucho cuidado, lo cual supuso un problema al principio.

## 8 CONCLUSIONES

Después de realizar el proyecto estoy muy satisfecho del camino recorrido ya que se han completado todos los objetivos propuestos, pudiendo aplicar muchos conceptos aprendidos en la universidad. Creo que muchas veces a la hora de realizar un proyecto empezamos a programar sin una fase previa de análisis y eso es un error que luego nos pasa factura. En este trabajo he podido dedicar esfuerzos en estudiar correctamente todo antes de programar, analizando los requisitos, creando diagramas y definiendo tanto una planificación como unas buenas prácticas de trabajo como el control de versiones.

También, a la hora de programar he dedicado esfuerzos al estudio de patrones de diseño y buenas prácticas, además de realizar tests de integración de todas las funcionalidades desarrolladas, lo cual se adapta muy bien, junto a la parte de análisis previo, despliegue y testeo en QA, a lo aprendido en la mención de Software.

El avance ha sido muy satisfactorio, el hecho de crear una solución grande desde cero me ha permitido encajar y entender muchas prácticas útiles para mi futuro como Ingeniero Informático.

## 9 PASOS FUTUROS

Es evidente que los avances en la globalización y el aumento constante del tamaño de la empresa ZETES hace que la gestión de la seguridad sea un aspecto con mucho potencial tanto en el presente como en el futuro.

Este proyecto ha supuesto un gran primer paso en un servidor de gestión de identidades, pero aún tiene mucho campo de mejora.

En el futuro reciente, se continuará mejorando y validando de forma que se pueda posteriormente deployar en PROD y así poder brindar estas funcionalidades a todos nuestros clientes.

Además, todo y que el Swagger sea bastante intuititivo, se continuará desarrollando una FrontEnd con Angular [15] que interactue con esta API de tal forma que se mejore la usabilidad.

## 10 AGRADECIMIENTOS

Me gustaría agradecer a ZETES por la confianza depositada en mi para este projecto, proporcionándome sus datos y dándome retroalimentación durante la escritura de la tesis. Especialmente a Javier Itoiz, Ingeniero de Software especializado en Arquitectura y Net Core, por ejercer el papel de tutor de

empresa en este proyecto y aportarme una valiosa guía a lo largo de la realización de este proyecto.

Gracias a la UAB y en especial a Helena Bolta Torrell, mi tutora, por la ayuda y el feedback proporcionado durante toda la realización de esta tesis.

Por último, pero no menos importante, gracias a mi familia y amigos por el apoyo en los momentos más difíciles en la realización de este gratificante proyecto.

## 11 REFERENCIAS

[1] Zetes, "Optimice su cadena de suministro | Zetes", https://www.zetes.com/es.

[2] Escuela de Organización Industrial, "La importancia de la cadena de suministro para la ventaja competitiva". https://www.eoi.es/blogs/katherinecarolinaacosta/2012/03/27/la-importancia-de-la-cadena-de-suministro-para-la-ventaja-competitiva.

[3] Cibergob, "Metodología de Adaptación Continua y Ágil", https://www.cibergob.es/metodologia-agil-de-adaptacion-continua.

[4] Microsoft Docs, "Control de versiones de Azure DevOps Services: Azure Databricks". https://docs.microsoft.com/es-es/azure/databricks/notebooks/azure-devops-services-version-control.

[5] Unican.es, "Consejos y Buenas Prácticas en Programación". https://personales.unican.es/sanchezbp/teaching/faqs/programming.html.

[6] Dofact, "C# Mediator Design Pattern". https://www.dofactory.com/net/mediator-design-pattern.

[7] Microsoft Docs, "Documentación de ASP.NET". https://docs.microsoft.com/es-es/aspnet/core.

[8] Microsoft Docs, "Documentación de la API web de ASP.NET Core con Swagger/OpenAPI". https://docs.microsoft.com/es-es/aspnet/core/tutorials/web-api-help-pages-using-swagger.

[9] Ultimate Beginner's Guide, "IdentityServer4 in ASP.NET Core". https://code-withmukesh.com/blog/identityserver4-in-aspnet-core.

[10] Ciberseguridad.com, "Autenticación JWT, qué es y cuándo usarla". https://ciberseguridad.com/guias/prevencion-proteccion/autenticacion-jwt.

[11] Dotnettricks.com, "What is OAuth? Secure Your ASP.NET Core App with OAuth 2.0", https://www.dotnettricks.com/learn/aspnetcore/what-is-oauth-secure-aspnet-core-app-oauth-2.

[12] Returngis.net, "OAuth 2.0, OpenID Connect y JSON Web Tokens (JWT) ¿Qué es qué?". https://www.returngis.net/2019/04/oauth-2-0-openid-connect-y-json-web-tokens-jwt-que-es-que.

[13] Josip Miskovic, "When To Use PATCH vs. PUT in Professional REST APIs - Josip Miskovic". https://josip-misko.com/posts/patch-vs-put-rest-api.

[14] Rapidapi.com, "What are Query Parameters (in API terms)". https://rapidapi.com/blog/api-glossary/parameters/query.

[15] Codemag.com, "Angular Front End Construction for ASP.NET Web API". https://www.codemag.com/Article/1705091/Building-an-Angular-Front-End-for-an-ASP.NET-Web-API.

# Anexos

## A.1. Anexo 1

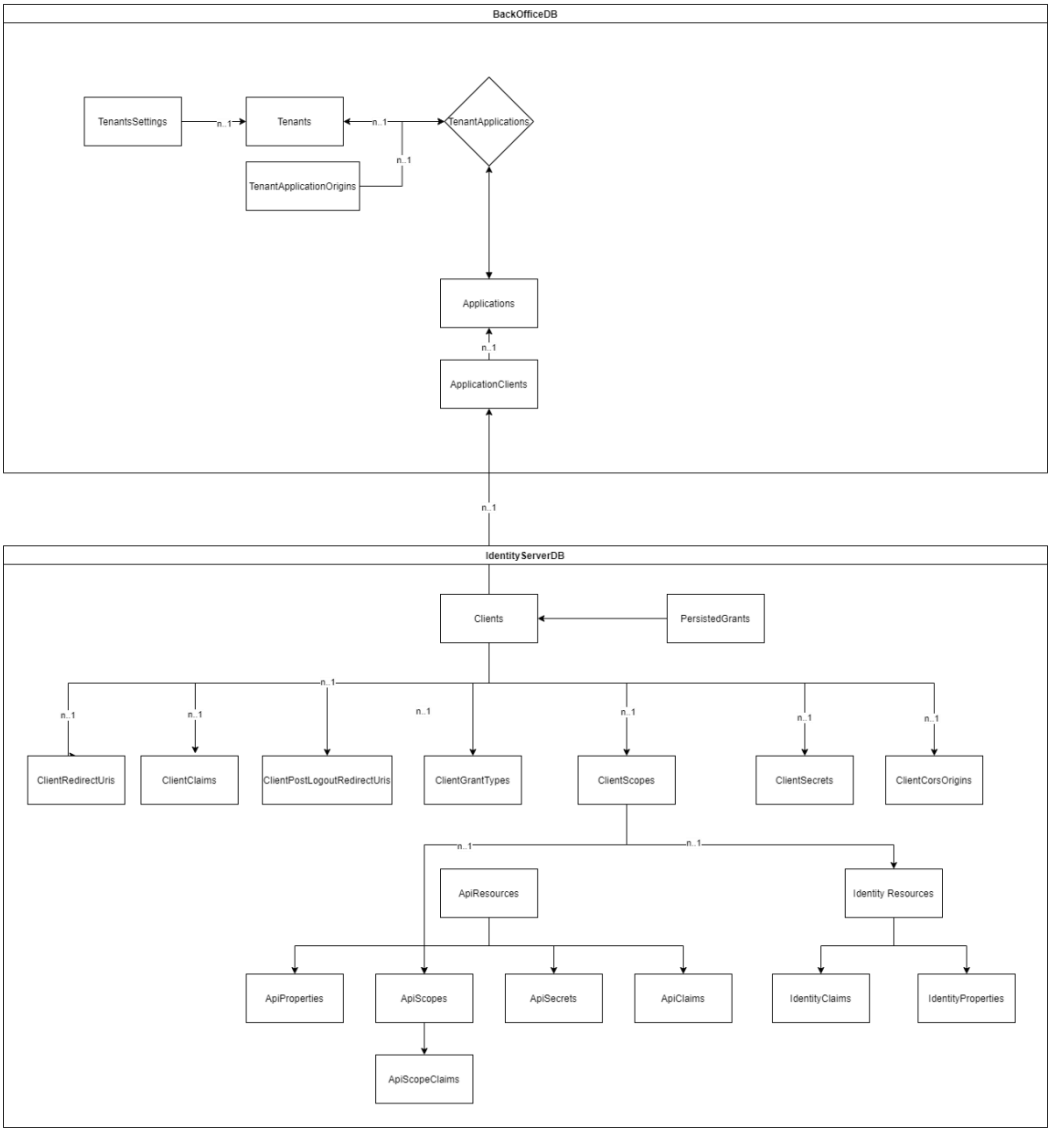


*Figura A.1 Diagrama de clases de la API realizada. Diferenciando las dos bases de datos gestionadas: BackOffice y IdentityServer. Además, se ve el flujo de los tres CRUDs realizados: Tenants, Applications y Clients.*

## A.2. Anexo 2

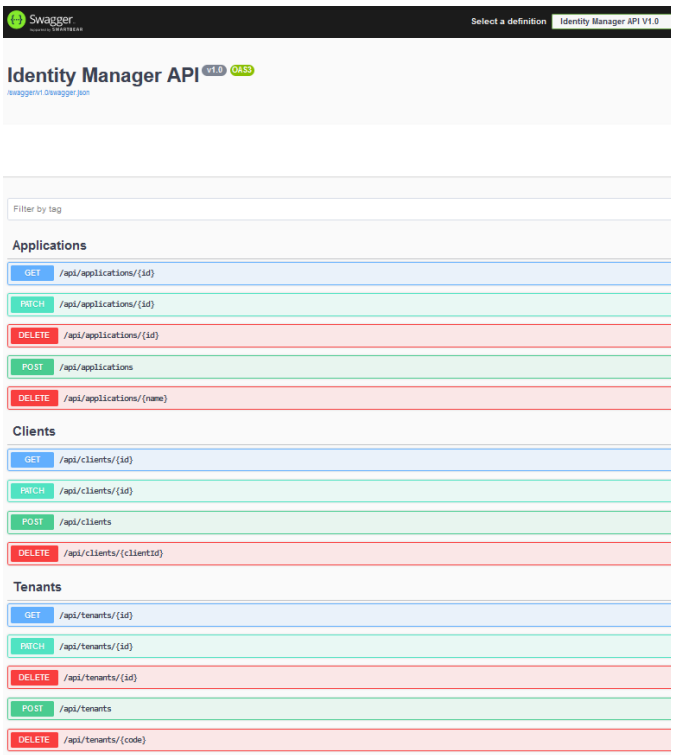


*Figura A.2 Visión completa mediante Swagger de la API realizada.*

## A.3. Anexo 3

### Requisitos Funcionales

| ID | Descripción |
|---|---|
| RF-1 | El sistema debe permitir crear, actualizar, eliminar y recuperar 'Applications' |
| RF-2 | El sistema debe permitir crear, actualizar, eliminar y recuperar 'Tenants'. |
| RF-3 | El sistema debe permitir crear, actualizar, eliminar y recuperar 'Clients'. |
| RF-4 | El sistema debe permitir recuperar múltiples 'Applications'. |
| RF-5 | El sistema debe permitir recuperar múltiples 'Tenants. |
| RF-6 | El sistema debe permitir recuperar múltiples 'Clients'. |
| RF-7 | Para comunicarse con la API, el sistema debe permitir añadir un token de autenticación. |
| RF-9 | El sistema debe permitir realizar operaciones de escritura y lectura en la base de datos de BackOffice. |
| RF-10 | El sistema debe permitir realizar operaciones de escritura y lectura en la base de datos de IdentityServer. |
| RF-11 | Los CRUD de Tenants y Applications permiten escribir y leer en la base de datos de BackOffice. |
| RF-12 | El CRUD de Clientes permite escribir y leer en la base de datos de IdentityServer. |
| RF-13 | El sistema debe ofrecer explicaciones detalladas y aclaradoras de las excepciones producidas al comunicarse con la API. |
| RF-14 | El usuario no podrá repetir nombre de Client, Tenant ni Application. En caso de hacerlo, se le mostrará un error explicativo. |

### Requisitos No Funcionales

| ID | Descripción |
|---|---|
| RNF-1 | El sistema debe permitir únicamente introducir registros que cumplan con los requisitos de la Base de Datos relacionan. Si no los cumple, mostrará un error explicativo. |
| RNF-2 | El sistema debe gestionar las peticiones de forma asíncrona. Utilizando tokens de cancelación. |
| RNF-3 | La base de datos debe responder las consultas en un tiempo menor a 2 segundos. |
| RNF-4 | La tasa de errores cometidos por el usuario deberá ser menor del 1% de las transacciones totales ejecutadas en el sistema. |
| RNF-5 | La API y la base de datos deben hospedarse en Software como Servicios (SaaS). |
| RNF-6 | El tiempo de aprendizaje requerido para utilizar correctamente la API debe ser menor a 1 hora. |
| RNF-7 | La API debe poder desplegarse y probarse en un entorno de Quality Assurance (QA). |
| RNF-8 | Toda operación a la base de datos deberá realizarse en un periodo de tiempo inferior a 15 segundos. En caso de llegar a los 15 segundos se cancelará la operación y saltará una TimeOutException. |
| RNF-9 | El sistema debe permitir generar un archivo json que contenga todas las operaciones de la API. |
| RNF-10 | El sistema debe devolver archivos json como respuesta a cualquier tipo de petición. |
| RNF-11 | Cuando se accede a la API desde el navegador web, se abrirá el Swagger por defecto. |
| RNF-12 | El borrado será lógico, tendremos un campo "isDeleted" que se pondrá a true cuando esté eliminado. Mientras no se elimine este valor será false. |
| RNF-13 | Se contará con los campos "UpdateDateTime", "CreateDateTime" y "UpdateClientId" para auditar quien y cuando se ha realizado la operación en cuestión. |
| RNF-14 | Cuando se crea un Tenant es necesario, como mínimo, introducir el nombre del Tenant via body. |
| RNF-15 | Cuando se elimina un Tenant es necesario introducir como parámetro el id de dicho Tenant. |
| RNF-16 | Cuando se actualiza un Tenant es necesario es necesario introducir como parámetro el id de dicho Tenant, además de un body con los valores a actualizar. |
| RNF-17 | Cuando se recupera un Tenant es necesario introducir como parámetro el id de dicho Tenant. |
| RNF-18 | Cuando se crea una Application es necesario, como mínimo, introducir el nombre de la Application via body. |
| RNF-19 | Cuando se elimina una Application es necesario introducir como parámetro el id de dicha Application. |
| RNF-20 | Cuando se actualiza una Application es necesario es necesario introducir como parámetro el id de dicha Application, además de un body con los valores a actualizar. |
| RNF-21 | Cuando se recupera una Application es necesario introducir como parámetro el id de dicha Application. |
| RNF-22 | Cuando se elimina un Client es necesario introducir como parámetro el id del cliente a eliminar. |
| RNF-23 | Cuando se actualiza un Client es necesario introducir como parámetro el id del cliente a actualizar. En cuanto a los detalles a actualizar, se introducirán mediante un body que tendrá como valores obligatorios: "clientId", "requireClientSecret", "allowOfflineAccess" y "clientSecrets". Los otros valores serán opcionales. |
| RNF-24 | Cuando se crea una Client es necesario introducir mediante body los parámetros "clientId", "requireClientSecret" y "allowOfflineAccess". Los otros valores serán opcionales. |
| RNF-25 | Cuando se recupera un Client es necesario introducir como parámetro el id del cliente. |

*Figura A.3 Análisis de requisitos*