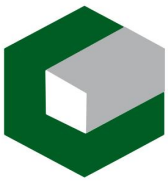
Universidad Tecnológica de La Habana

“José Antonio Echeverría”

Facultad de Ingeniería Informática.



Corte 1 de la asignatura Diseño de Software

Autor: Carlos Daniel Vilaseca Illnait

Tutores: Dra. C. Raisa Socorro Llanes

Dra. C. Lisandra Bravo Ilisastigui

La Habana, Cuba

Febrero, 2024

Resumen

El presente trabajo tiene como objetivo principal definir la arquitectura candidata de un software para la gestión de usuarios y grupos de un Directorio Activo en la empresa Avangenio. Se abordarán los puntos clave como la definición de la arquitectura candidata, la selección y modelado de estilos y patrones arquitectónicos, la identificación y justificación de los principios de diseño presentes, así como la formalización de los patrones de diseño en la solución propuesta. Se realizarán comparaciones entre diferentes arquitecturas y tecnologías, justificando la selección de las tecnologías utilizadas en base a las necesidades específicas del proyecto.

Palabras clave: Noswork, arquitectura, arquitectura candidata, estilos arquitectónicos, principios de diseño.

Abstract

The main objective of this work is to define the candidate architecture of a software for managing users and groups in an Active Directory at the company Avangenio. This will involve addressing key points such as the definition of the candidate architecture, the selection and modeling of architectural styles and patterns, identification and justification of design principles present, as well as the formalization of design patterns in the proposed solution. Comparisons will be made between different architectures and technologies, justifying the selection of technologies used based on the specific needs of the project.

Keywords: Noswork, architecture, candidate architecture, architectural styles, design principles.

Índice

Tablas y figuras

Introducción

Un software colaborativo es una herramienta que permite a los miembros de un equipo trabajar juntos de manera más eficiente y productiva, independientemente de su ubicación física. Estas herramientas proporcionan una plataforma en línea para compartir información, comunicarse y colaborar en tiempo real. Los softwares colaborativos suelen incluir funciones como chat, videoconferencia, edición de documentos en tiempo real y gestión de proyectos, lo que facilita el trabajo en equipo y la coordinación entre los miembros del equipo [1, 2].

Los softwares colaborativos surgen como una respuesta a la necesidad de mejorar la comunicación, el intercambio de información y el trabajo en equipo dentro de las empresas. Al tener toda la información y los recursos ubicados en un solo sitio y con cada miembro teniendo el acceso correspondiente, se aumenta la transparencia entre departamentos y se facilita el trabajo en equipo. Ejemplos de software colaborativos incluyen *Google Workspace*, *Microsoft 365* y *Zoho* [2, 3].

NosWork Workspaces es un ejemplo de software colaborativo que permite crear espacios de trabajo virtuales colaborativos, donde los usuarios pueden compartir archivos, comunicarse por chat o videoconferencia, y acceder a diversas aplicaciones web integradas. Esta plataforma facilita el trabajo remoto y la gestión de proyectos de forma ágil y segura [7].

Las aplicaciones colaborativas generalmente tienen un módulo integrado para la gestión de usuarios, facilitando la administración de los miembros del equipo y sus permisos dentro de la plataforma [55, 56, 57]. Esto permite a los administradores agregar, eliminar o modificar usuarios y controlar su acceso a diferentes áreas y funciones de la plataforma. Al tener una herramienta de gestión integrada, los administradores pueden asegurarse de que cada miembro del equipo tenga acceso a las herramientas y recursos necesarios para realizar su trabajo de manera efectiva. Al mismo tiempo que se protege la información confidencial y se mantiene la seguridad de la plataforma [3].

Comúnmente estas herramientas utilizan un procedimiento de autenticación que permite a los usuarios acceder a múltiples sistemas, recursos, o aplicaciones con una sola identificación base (dígase usuario y contraseña), denominado *Single Sign-On*

(SSO). El SSO se suele utilizar en un contexto empresarial, cuando las aplicaciones de los usuarios las asigna y gestiona un equipo interno de tecnologías de la información (TI) [4, 5, 6]. Existen varios sistemas que implementan SSO como lo son el *Central Authentication Service* (CAS), el *Security Assertion Markup Language* (SAML) y OAuth [8, 9, 10, 13].

El Protocolo de Acceso Ligero a Directorios (LDAP por sus siglas en inglés) es un protocolo utilizado por muchos sistemas de SSO como proveedor de identidad, incluido Directorio Activo de Microsoft, para acceder a la información del directorio y gestionarla [10, 11, 12, 16].

Los servicios de Directorio Activo, actúan como una base de datos centralizada para almacenar y gestionar información sobre usuarios, computadoras y otros recursos en una organización. Los administradores pueden utilizar esta base de datos para controlar el acceso a los recursos y redes, verificando la identidad de los usuarios y asignándoles permisos adecuados. [12, 16].Existen varios sistemas que implementan Directorio Activo, como *Azure Active Directory* de Microsoft, *Apache Directory Studio* y *Oracle Directory Server Enterprise Edition* [12, 14, 15]. Sin embargo, no todos son gratuitos, o sencillos de configurar.

Samba 4 es una de las alternativas gratuitas que implementan Directorio Activo y actúa como controlador de dominio en sistemas Unix. Es una implementación libre del protocolo de archivos compartidos de Microsoft para sistemas de tipo Unix [16, 17]. En Noswork se utiliza un servidor de Samba 4 como controlador de dominio en el directorio. Para la administración remota de este servidor se desarrolló un módulo propio, dado que las herramientas existentes como Remote Server Administration Tools (RSAT) o Webmin, no cumplen con las necesidades específicas del producto. El módulo está construido en dos partes, un *backend* llamado *AD-manager-API* y un *frontend* llamado *Admin Console*.

Definición de la arquitectura

Dado que el flujo del sistema esta basado en capas evidentemente se presenta una **arquitectura de n capas**:

Es un patrón de diseño de software que divide un sistema en capas horizontales, donde cada capa tiene un rol específico y solo se comunica con las capas adyacentes.

La arquitectura de n capas está presente en un sistema cuando este se divide en capas lógicas independientes, cada una con una responsabilidad específica. Las capas se comunican entre sí a través de interfaces bien definidas.

Ejemplos de areas en las que se usa:

- **Aplicaciones web**: Interfaz de usuario, lógica de negocio, acceso a datos.
- Aplicaciones móviles: Presentación, lógica de negocio, acceso a datos, almacenamiento local.
- otros

Capas comunes:

- Presentación: Interfaz de usuario, interacción con el usuario.
- Lógica de negocio: Reglas y procesos del sistema.
- Acceso a datos: Interacción con la base de datos o almacenamiento.

¿Por qué no otra arquitectura?

No existe una arquitectura de software que sea perfecta para todos los sistemas. La arquitectura adecuada para un sistema depende de sus requisitos específicos. En el caso de las aplicaciones web, la arquitectura de N capas es una opción común ya que resuelve la mayor parte de los problemas comunes, especialmente los relacionados a seguridad, escalabilidad y mantenibilidad. Sin embargo, también se puede combinar con una arquitectura de microservicios para mejorar el rendimiento y/o la escalabilidad (si así se requiere y el presupuesto lo permite).

Por ejemplo si se compara con una arquitectura hexagonal la respuesta obvia es que la arquitectura hexagonal es muy compleja de implementar y llegar a ese punto.

- Complejidad: La arquitectura hexagonal puede ser más compleja de implementar que otras arquitecturas, como la arquitectura de N capas. Esto se debe a que

requiere un mayor conocimiento de los patrones de diseño y de las técnicas de inyección de dependencias.

- Costo: La arquitectura hexagonal puede ser más costosa de implementar que otras arquitecturas. Esto se debe a que requiere un mayor esfuerzo de desarrollo y mantenimiento.

En la Figura 1 se muestra el diagrama enfocado en responsabilidades y de la implementación de la arquitectura de n capas en el sistema.

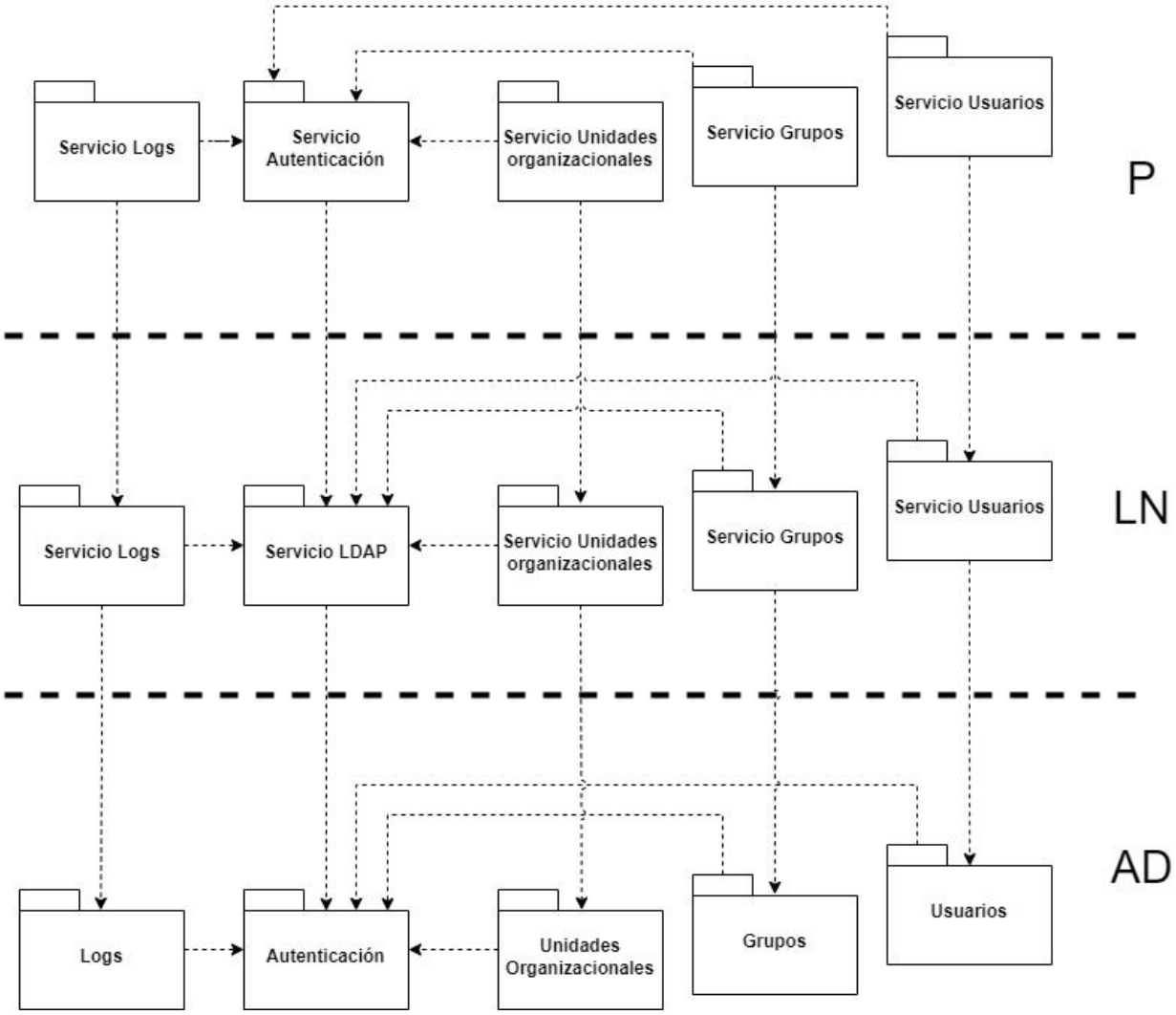


Figura 1 Diagrama n Capas: Enfoque de Responsabilidades

Stack tecnológico

En el contexto del desarrollo de software, un stack tecnológico es el conjunto de tecnologías, herramientas y lenguajes de programación que se utilizan para crear un sistema o aplicación. Incluye la base de datos, los lenguajes de programación, los

frameworks, las bibliotecas y otros componentes. Esta sección está centrada en la justificación de tecnologías como los frameworks y el directorio.

Directorio Activo y LDAP

Un directorio activo es un servicio de directorio que proporciona un sistema de administración de identidades y acceso para redes informáticas. Un directorio activo puede almacenar información sobre usuarios, equipos, dispositivos y otros recursos de red, proporcionando un entorno seguro para almacenar datos de identidad y acceso [10, 11, 12, 16].

El protocolo LDAPS, que es una versión segura de LDAP, utiliza el cifrado para proteger la comunicación entre el directorio activo y la aplicación web (similar a HTTP vs HTTPS) [10, 11, 12, 16].

Directorio Activo

- **Requisitos de integración con otros sistemas**: Un directorio puede actuar como un proveedor de identidad (Identity Provider, IdP) para proporcionar SSO a otros sistemas empresariales.
- Requisitos de seguridad: Un directorio proporciona un entorno seguro para almacenar datos de identidad y acceso. El protocolo LDAPS, que es una versión segura de LDAP, utiliza el cifrado para proteger la comunicación entre el directorio y la aplicación web.
- Requisitos de escalabilidad: Un directorio está diseñado para ser escalable. Puede admitir un gran número de usuarios y recursos.
- Requisitos de facilidad de administración: Un directorio es fácil de administrar. Proporciona herramientas y servicios para administrar usuarios, equipos y otros recursos de red.

Postgres

PostgreSQL es un sistema de gestión de bases de datos relacional de código abierto que ha ganado una gran popularidad en la comunidad de desarrolladores. Conocido por su robustez, fiabilidad y capacidades avanzadas, PostgreSQL ofrece una amplia gama de características que lo convierten en una opción atractiva para proyectos de cualquier tamaño [24]:

- Integración con otros sistemas: Postgres puede integrarse con diferentes sistemas a través de herramientas y extensiones, permitiendo la comunicación y colaboración con otros sistemas.

- Escalabilidad: Postgres es altamente escalable y puede manejar grandes volúmenes de datos y usuarios a través de la implementación de clusters de alta disponibilidad y técnicas de particionamiento.

- Seguridad: Postgres ofrece múltiples características de seguridad, como roles de usuario, cifrado de datos, SSL/TLS y autenticación externa, garantizando la protección de la información almacenada en la base de datos.

- Facilidad de administración: Postgres proporciona una variedad de herramientas y utilidades para gestionar bases de datos, como pgAdmin, psql y extensiones como pg_cron. Además, tiene un sistema de gestión de autenticación y autorización basado en roles que facilita la administración de usuarios y permisos.

MongoDB

MongoDB es una base de datos NoSQL que ha revolucionado la forma en que las empresas gestionan sus datos. Con su enfoque en la escalabilidad, flexibilidad y rendimiento, MongoDB se ha convertido en una opción popular para una amplia gama de aplicaciones, desde startups hasta grandes empresas [25].

- Requisitos de integración con otros sistemas: MongoDB puede integrarse con diferentes sistemas a través de conectores y APIs, permitiendo la interoperabilidad con diferentes plataformas.
- Requisitos de seguridad: MongoDB ofrece características de seguridad como autenticación, autorización, cifrado de datos en reposo y en tránsito, integración con protocolos de seguridad estándar, entre otros.
- Requisitos de escalabilidad: MongoDB es altamente escalable y puede manejar grandes volúmenes de datos y usuarios, especialmente a través de su arquitectura distribuida y capacidad de fragmentación.
- Requisitos de facilidad de administración: MongoDB ofrece herramientas de administración como MongoDB Compass y la interfaz de línea de comandos para gestionar bases de datos de manera efectiva.

En resumen, al comparar Directorio Activo, Postgres y MongoDB en términos de gestión de identidades, accesos y seguridad, podemos concluir que:
Directorio Activo destaca en:

- Integración con sistemas empresariales: Es un proveedor de identidad especializado que facilita la integración con otros sistemas a través de SSO.
- Seguridad mejorada: Ofrece un entorno seguro para almacenar datos de identidad y acceso, con protocolos específicos de cifrado.
- Escalabilidad para entornos corporativos: Está diseñado para manejar grandes cantidades de usuarios y recursos de manera eficiente.
- Facilidad de administración: Proporciona herramientas especializadas para gestionar identidades y accesos de forma centralizada y eficiente.

Por otro lado, Postgres y MongoDB tienen sus propias fortalezas:
Postgres es altamente escalable y ofrece robustas características de seguridad, así como herramientas para administrar bases de datos de manera efectiva. Aunque puede integrarse con sistemas externos, no cuenta con las capacidades especializadas de un proveedor de identidad como Directorio Activo.
MongoDB es altamente escalable y flexible, con capacidades de fragmentación y distribución. Ofrece características de seguridad avanzadas, pero no está diseñado específicamente para la gestión de identidades y accesos como Directorio Activo.

En conclusión, para proyectos donde la gestión de identidades y el control de accesos son prioritarios, la elección del Directorio Activo sobre Postgres y MongoDB puede ofrecer beneficios significativos en términos de seguridad y eficiencia operativa. Directorio Activo es la preferencia de las empresas debido a su amplia adopción, conjunto completo de características de seguridad y administración, y compatibilidad con una amplia gama de sistemas empresariales.

Framework de frontend: SvelteKit

La selección del framework de frontend se tomó teniendo en cuenta los siguientes puntos clave: eficiencia, flexibilidad, escalabilidad, curva de aprendizaje y soporte de la comunidad.

A continuacion se realiza un análisis comparativo sobre los siguientes frameworks: Sveltekit, Nextjs, Nuxt.

SvelteKit

SvelteKit es un marco de trabajo moderno y potente para la construcción de aplicaciones web, que se basa en la popular biblioteca de JavaScript Svelte. Con SvelteKit, los desarrolladores pueden crear aplicaciones rápidas, eficientes y fáciles de mantener, utilizando un enfoque basado en componentes y una arquitectura robusta [27].

- Eficiencia: Utiliza un enfoque de compilación previa para generar sitios estáticos de alto rendimiento.
- Flexibilidad: Ofrece gran flexibilidad en personalización y configuración, con un enfoque basado en componentes y reactividad.
- Escalabilidad: Altamente escalable, adecuado para proyectos de diferentes tamaños y complejidades.
- Curva de aprendizaje: Curva de aprendizaje relativamente baja, gracias a una sintaxis simple y familiar basada en componentes reactivos.
- Comunidad: Aunque más pequeña que otras, es una comunidad en crecimiento con soporte activo y recursos disponibles.

Next.js

Next.js es un marco de trabajo de React muy popular que permite a los desarrolladores crear aplicaciones web modernas de forma rápida y eficiente. Con Next.js, se puede aprovechar todo el poder de React junto con las ventajas de un enfoque de renderizado del lado del servidor, enrutamiento dinámico, pre-renderizado y muchas otras características avanzadas [28, 29].

- Eficiencia: Ofrece una buena eficiencia, con la capacidad de renderizado del lado del servidor y en el cliente.
- Flexibilidad: Es flexible y permite la creación de diferentes tipos de aplicaciones web, desde estáticas hasta complejas aplicaciones de una sola página.
- Escalabilidad: Tiene una buena capacidad de escalar y manejar proyectos de gran envergadura.

- Curva de aprendizaje: La curva de aprendizaje puede ser moderada, ya que requiere familiarizarse con sus conceptos y funcionalidades.
- Comunidad: Next.js cuenta con una amplia comunidad de desarrolladores, lo que facilita encontrar recursos y soporte en línea.

Nuxt

Nuxt.js es un marco de trabajo de código abierto basado en Vue.js que facilita la creación de aplicaciones web modernas y potentes con Vue.js. Con Nuxt.js, los desarrolladores pueden aprovechar las capacidades de Vue.js para crear aplicaciones de una sola página (SPA), aplicaciones de múltiples páginas (MPA) e incluso aplicaciones estáticas o generadas de forma dinámica [30, 31].

- Eficiencia: Ofrece eficiencia en el desarrollo de aplicaciones web, con facilidades para la creación de aplicaciones universal y estáticamente generadas.
- Flexibilidad: Es flexible y adaptable a diferentes tipos de proyectos, con un enfoque en la simplicidad y facilidad de uso.
- Escalabilidad: Puede escalar adecuadamente para manejar proyectos de diversos tamaños y complejidades.
- Curva de aprendizaje: La curva de aprendizaje puede ser moderada, especialmente para aquellos que están familiarizados con Vue.js.
- Comunidad: Nuxt cuenta con una comunidad activa y en crecimiento, lo que proporciona soporte y recursos disponibles para los desarrolladores.

Es muy complicado realizar una comparación de cada aspecto de todos los frameworks por las siguientes razones:

- Hay muchos frameworks disponibles. En la actualidad, hay cientos de frameworks de desarrollo web disponibles. Cada framework tiene sus propias características y ventajas.
- Los frameworks son complejos. Los frameworks de desarrollo web son complejos y ofrecen una amplia gama de características. Es difícil comparar todas estas características de forma exhaustiva.

- Los requisitos de las aplicaciones varían. Las aplicaciones web tienen diferentes requisitos. Un framework que es ideal para una aplicación web puede no serlo para otra.

Por estas razones, es difícil decir que un framework es mejor que otro. El mejor framework para una aplicación web específica depende de los requisitos de la aplicación y de las preferencias del equipo de desarrollo.

En última instancia, todo se reduce a si puedes resolver el problema que se tiene entre manos con ese framework. Si un framework ofrece las características que se necesita para crear la aplicación que se desea, entonces ese framework es el adecuado.

Framework de backend: FastAPI

La selección del framework de backend se tomó teniendo en cuenta los siguientes puntos clave: rendimiento, facilidad de uso, escalabilidad y compatibiidad.

A continuacion se realiza un análisis comparativo sobre los siguientes frameworks: FastAPI, Django, Flask.

FastAPI

FastAPI es un marco de trabajo moderno y de alto rendimiento para la creación de APIs web con Python. Con FastAPI, los desarrolladores pueden crear APIs rápidas y eficientes utilizando Python tipo anotación y aprovechar al máximo las características modernas de Python, como el tipado estático y la generación automática de documentación API [32, 33].

- Rendimiento: FastAPI es conocido por su alto rendimiento debido a su uso de Pydantic para la validación de datos y Starlette para el manejo de peticiones HTTP, lo que permite un procesamiento rápido de las solicitudes.
- Facilidad de uso: FastAPI es reconocido por su sintaxis intuitiva y fácil de entender, que se basa en la tipificación de datos para mejorar la productividad del desarrollador y facilitar la escritura de código limpio y legible.
- Escalabilidad: FastAPI es altamente escalable, lo que permite manejar grandes volúmenes de tráfico y adaptarse a entornos de alta demanda de manera eficiente.

- Compatibilidad: FastAPI es compatible con las últimas versiones de Python y es ampliamente aceptado en la comunidad de desarrollo de Python.

Django

Django es un marco de trabajo web de alto nivel que fomenta un desarrollo rápido y limpio en Python. Con una amplia gama de características integradas, Django facilita la creación de aplicaciones web seguras y escalables, desde simples sitios estáticos hasta complejas plataformas empresariales [34, 35, 37].

- Rendimiento: Django, al ser un framework más completo, puede presentar un rendimiento ligeramente inferior en comparación con FastAPI en aplicaciones muy intensivas en operaciones de entrada y salida.
- Facilidad de uso: Django es conocido por su amplia funcionalidad integrada y su forma de "hágalo usted mismo", que facilita el desarrollo rápido de aplicaciones web complejas.
- Escalabilidad: Django es altamente escalable y ha demostrado su capacidad para manejar proyectos de gran envergadura a lo largo del tiempo.
- Compatibilidad: Django es un framework maduro y ampliamente utilizado que cuenta con una amplia gama de bibliotecas y extensiones para facilitar el desarrollo de aplicaciones web.

Un factor tambien que se debe tener en cuenta es que la capa de datos es un Directorio, o sea, no se estaría usando todo el potencial de Django dado que este reside en su ORM y base de datos relacional integradas.

Flask

Flask es un micro marco de desarrollo web basado en Python que ofrece una gran flexibilidad y simplicidad para la creación de aplicaciones web rápidas y eficientes. Aunque es ligero en comparación con otros marcos, Flask proporciona una amplia variedad de extensiones que permiten añadir funcionalidades según sea necesario, lo que lo convierte en una opción popular entre los desarrolladores [36, 37].

- Rendimiento: Flask es conocido por su ligereza y simplicidad, lo que le permite ofrecer un rendimiento excelente en aplicaciones más livianas o en casos donde se requiere una menor sobrecarga de recursos.

- Facilidad de uso: Flask es reconocido por su simplicidad y minimalismo, lo que lo convierte en una excelente opción para proyectos más pequeños o para desarrolladores que prefieren un enfoque minimalista en su codificación.

- Escalabilidad: Flask es escalable y puede adaptarse a proyectos de diferentes tamaños, pero puede requerir más configuración y ajustes para escalar en comparación con FastAPI.

- Compatibilidad: Flask es compatible con una amplia variedad de extensiones y librerías, lo que facilita la integración de funcionalidades adicionales en las aplicaciones web desarrolladas con este framework.

En este contexto, FastAPI es una mejor opción dado que provee un alto rendimiento y facilidad de uso. Django puede ser una mejor opción para aplicaciones web más complejas que requieren una base de datos relacional y un ORM.

Problemas frecuentes y solución

Algunos problemas frecuentes que se encuentran en las aplicaciones web y cómo solucionarlos se describen a continuación.

- Seguridad de la información y acceso: La gestión de identidades y el control de accesos son fundamentales para garantizar la seguridad de una aplicación web. Active Directory proporciona un entorno seguro para almacenar datos de identidad y acceso, permitiendo una autenticación segura y centralizada para los usuarios. Integrar Active Directory con SvelteKit y FastAPI puede asegurar un control riguroso sobre quién tiene acceso a la aplicación y a qué datos.

- Rendimiento y tiempos de carga: Problemas de rendimiento y tiempos de carga lentos pueden afectar la experiencia del usuario. SvelteKit ofrece un enfoque de compilación previa que permite generar sitios estáticos de alto rendimiento, mientras que FastAPI es conocido por su velocidad y eficiencia en el procesamiento de solicitudes HTTP. Combinar estas tecnologías puede ayudar a mejorar el rendimiento global de la aplicación.

- Escalabilidad y mantenibilidad: A medida que una aplicación web crece en usuarios y funcionalidades, la escalabilidad y mantenibilidad se vuelven críticas. FastAPI es altamente

escalable y permite manejar grandes volúmenes de tráfico, por lo que se puede garantizar que la aplicación pueda crecer y evolucionar con éxito.

- Integración de APIs y servicios externos: Las aplicaciones web suelen necesitar integrarse con APIs externas y servicios web. FastAPI es ideal para la creación de APIs RESTful y su integración con diferentes servicios y bases de datos, mientras que SvelteKit puede consumir estas APIs de forma eficiente en el frontend. La combinación de estas tecnologías facilita la integración con servicios externos de manera eficaz.

Al emplear un enfoque de múltiples capas en la arquitectura de la aplicación web, se logra una mayor separación de responsabilidades y una mejor organización del código. Al mismo tiempo, al aprovechar las fortalezas de cada tecnología en su respectiva capa, se mejora la gestión y funcionalidad de la aplicación, permitiendo abordar de manera efectiva los problemas frecuentes que pueden surgir en el desarrollo y mantenimiento de aplicaciones web complejas.

Selección y modelado de estilos y patrones

Los estilos arquitectónicos desempeñan un papel crucial al establecer las restricciones y normas que guían la composición y relación de los componentes en un sistema de software. Cada estilo arquitectónico aporta conceptos, herramientas, problemas y experiencias específicos que influyen en la forma en que se construye y despliega un sistema.

Estilo llamada-retorno

El estilo arquitectónico "Llamada-Retorno" es un enfoque basado en la separación e interacción de capas en un sistema de software, creando una estructura eficiente y escalable para la comunicación y procesamiento de datos. Este estilo arquitectónico se fundamenta en la idea de separar las capas del sistema de software en distintos niveles de abstracción, donde cada capa tiene un rol específico y se comunica con capas adyacentes a través de llamadas y retorno de resultados [38, 39].

En el contexto de este proyecto, el estilo "Llamada-Retorno" se manifiesta en la estructura de capas del sistema de gestión de usuarios y grupos en el Directorio Activo de Noswork. Por ejemplo, la capa de presentación se encarga de la interacción con el usuario final, la capa de lógica de negocio implementa las reglas y procesos del sistema, y la capa de acceso a datos se encarga de la interacción con el Directorio Activo para el almacenamiento y recuperación de la información [38, 39].

La implementación del estilo "Llamada-Retorno" en la arquitectura del sistema garantiza una separación clara de responsabilidades, facilitando la mantenibilidad, la escalabilidad y la eficiencia en el desarrollo y evolución del software. A través de este enfoque arquitectónico, se establece una estructura robusta y coherente que permite una comunicación eficaz entre las diferentes capas del sistema, mejorando así su rendimiento y calidad [38, 39].

Patrón n-capas

El patrón de arquitectura de n-capas es un enfoque comúnmente utilizado en el diseño de sistemas de software que se basa en la división del sistema en capas horizontales, cada una con un propósito específico y responsabilidades bien definidas. Este enfoque facilita la separación de preocupaciones, la modularidad y la escalabilidad del sistema, lo que resulta en un diseño más robusto y mantenible [40].

La comunicación entre las capas en el patrón de n-capas se realiza a través de llamadas y retornos de datos, siguiendo el estilo "Llamada-Retorno". Por ejemplo, la capa de presentación envía una solicitud a la capa de lógica de negocio, la cual procesa la solicitud y retorna los resultados a la capa de presentación para su visualización al usuario [40].

Este enfoque arquitectónico permite una mayor modularidad y separación de responsabilidades en el sistema, lo que facilita la evolución y el mantenimiento del software a lo largo del tiempo. Asimismo, la implementación del patrón de n-capas basado en el estilo "Llamada-Retorno" contribuye a una arquitectura organizada y eficiente, promoviendo una estructura bien definida y escalable para el sistema [40].

En la Figura 2 se representa el patrón de n-capas según [47].

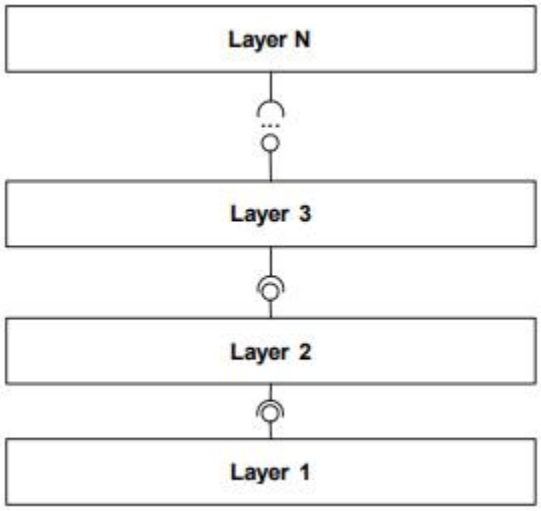


Figura 2 Representación del patrón n-capas

Patrón cliente-servidor

El patrón Cliente-Servidor es un modelo de arquitectura común en el diseño de sistemas de software, donde un cliente solicita servicios o recursos al servidor, que actúa como proveedor de estos servicios. La comunicación entre el cliente y el servidor se basa en intercambios de mensajes, siguiendo el estilo "Llamada-Retorno" para la transmisión de datos y resultados entre ambas partes [40].

- Cliente: El cliente es la parte del sistema que envía solicitudes al servidor para obtener datos o realizar operaciones específicas. En este caso, el cliente puede ser la interfaz de usuario o cualquier componente del sistema que requiera acceder a los servicios proporcionados por el servidor.
- Servidor: El servidor es la parte central que recibe las solicitudes del cliente, procesa las peticiones y retorna los resultados al cliente. Aquí es donde se alojan y ejecutan los servicios que brindan la funcionalidad requerida por el cliente.

La interacción entre el cliente y el servidor en el patrón Cliente-Servidor se realiza a través de llamadas y respuestas, siguiendo el flujo de datos en ambas direcciones. El cliente envía una solicitud al servidor, el cual procesa la solicitud y envía una respuesta de vuelta al cliente con los resultados solicitados.

Este enfoque arquitectónico basado en el estilo "Llamada-Retorno" ofrece ventajas como la separación clara de responsabilidades entre el cliente y el servidor, la interoperabilidad y la escalabilidad del sistema. Además, promueve una comunicación

eficaz entre las partes, facilitando el intercambio de información y la ejecución de operaciones de manera eficiente y estructurada [40, 39].

La implementación del patrón Cliente-Servidor en el contexto del estilo "Llamada-Retorno" contribuye a la definición de una arquitectura organizada y eficaz para el sistema, mejorando la modularidad, la escalabilidad y la interacción entre los componentes del sistema.

En la Figura 3 se representa el patrón cliente-servidor en una aplicacion de 3 capas según [47].

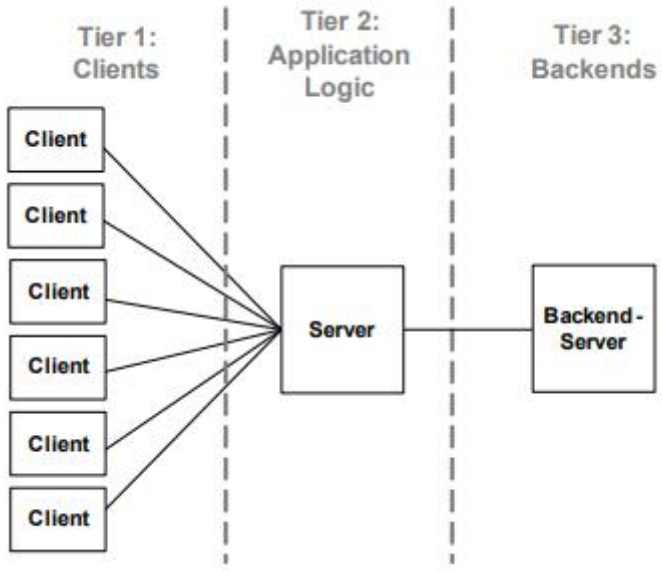


Figura 3 Representación del patrón cliente servidor

Estilo SOA

Service Oriented Architecture (SOA) es un estilo arquitectónico que se basa en la idea de descomponer una aplicación en servicios independientes y reutilizables que pueden comunicarse entre sí a través de estándares abiertos. Es decir, SOA proporciona una forma de diseñar sistemas de software que promueven la reutilización, la interoperabilidad y la flexibilidad [41, 42, 43, 44, 45].

Al integrar la arquitectura orientada a servicios en el diseño del sistema, se pueden identificar los siguientes aspectos relevantes:

- Servicios Independientes: El sistema se descompone en servicios separados, cada uno con funcionalidades específicas y bien definidas. Cada servicio opera de manera independiente y se comunica con otros servicios a través de interfaces estándar.

- Comunicación Basada en Protocolos Estándar: Los servicios se comunican entre sí a través de protocolos estandarizados, como HTTP, SOAP o REST, lo que facilita la interoperabilidad y la integración de los diferentes componentes del sistema.

- Reutilización y Escalabilidad: La arquitectura orientada a servicios fomenta la reutilización de servicios existentes y la escalabilidad del sistema mediante la incorporación de nuevos servicios según sea necesario.

Al combinar el estilo "Llamada-Retorno" con la arquitectura orientada a servicios, se logra una estructura modular, flexible y adaptable para el sistema de gestión de usuarios y grupos en el Directorio Activo de Avangenio. La integración de estos estilos arquitectónicos permite una comunicación eficiente entre los componentes del sistema, promueve la reutilización de servicios y facilita la evolución y expansión del software de manera sostenible.

Patrón de Acceso a Datos

El patrón de Acceso a datos se centra en la gestión del acceso a los datos y recursos compartidos a través de servicios en una arquitectura orientada a servicios (SOA). Este patrón actúa como un intermediario entre los servicios que requieren acceso a los datos y las fuentes de datos subyacentes, proporcionando una capa de abstracción que facilita la manipulación de los datos de manera segura y eficiente [43, 44, 45].

Características clave del patrón Acceso a Datos [43, 44, 45]:

- Gestión de acceso a datos: El acceso a datos se encarga de administrar el acceso a los datos almacenados en diferentes fuentes de datos, como bases de datos, sistemas de archivos o servicios externos. Permite a los servicios acceder y manipular los datos de forma centralizada y reutilizable, sin exponer directamente la lógica de acceso a los datos internos.

- Capa de abstracción: Al implementar el patrón Acceso a Datos , se establece una capa de abstracción entre los servicios y las fuentes de datos, lo que permite

encapsular la complejidad de las operaciones de acceso a los datos y facilitar la reutilización de la lógica de acceso.

- Promueve la cohesión y el desacoplamiento: Al separar la lógica de acceso a los datos de la lógica de negocio de los servicios, se promueve la cohesión y el desacoplamiento entre los componentes del sistema. Esto facilita la evolución y mantenimiento del sistema al permitir cambios en las fuentes de datos sin afectar directamente a los servicios consumidores de dichos datos.
- Seguridad y consistencia de los datos: El patrón Acceso a Datos también contribuye a garantizar la seguridad y la consistencia de los datos al centralizar y estandarizar las operaciones de acceso, permitiendo la implementación de mecanismos de control de acceso y validación de datos de forma centralizada.

Principios de diseño

Los principios de diseño son fundamentales para garantizar que el software desarrollado cumpla con los estándares de calidad, funcionalidad y rendimiento esperados. Estos principios ayudan a los desarrolladores a tomar decisiones informadas sobre la arquitectura, la estructura y la implementación del software, y a asegurarse de que el código sea coherente, fácil de mantener y extender en el futuro [46].

En este capítulo se describen los principios identificados en el sistema.

Principio de responsabilidad única

El Principio de Responsabilidad Única es un principio fundamental de diseño de software que establece que una clase o módulo debe tener una sola razón para cambiar. En otras palabras, cada componente de software debe tener una única responsabilidad y estar enfocado en realizar una tarea específica [48].

Al aplicar el Principio de Responsabilidad Única, se busca promover la cohesión y la modularidad en el código, evitando la sobrecarga de responsabilidades en una sola entidad. Esto conduce a un código más claro, mantenible y fácil de entender, ya que cada parte del sistema se encarga de un aspecto específico de la funcionalidad [48].

En la Figura 10 se evidencia la aplicación de este principio en las clases **UserService, GroupService** y **AuthService** tienen la responsabilidad única de llamar a sus respectivos endpoints para realizar las operaciones.

Ley demeter

El Principio de Programación Tímida, también conocido como Ley de Demeter, es un principio de diseño de software que promueve la encapsulación y la reducción del acoplamiento entre objetos. Este principio se basa en la idea de que un objeto debe tener un conocimiento limitado sobre la estructura interna de otros objetos, evitando que una clase acceda directamente a las propiedades de objetos relacionados.

La Ley de Demeter se resume en la famosa frase "No hables con extraños", lo que significa que un objeto solo debe interactuar con sus objetos cercanos y no con los objetos anidados dentro de ellos. En lugar de acceder a las propiedades de los objetos internos, se deben delegar las operaciones a los objetos relacionados a través de métodos públicos.

En la Figura 10 se evidencia la aplicación de este principio dado que las clases no interactúan entre sí.

Principio Hollywood

Este principio se basa en que los componentes de software de bajo nivel o módulos no controlan la ejecución del programa, sino que son "llamados" o "invocados" por componentes o módulos de alto nivel, es decir al diseñar piezas de alto nivel que interactuarán con piezas de más bajo nivel, se debe evitar las llamadas de estas últimas a las primeras, las piezas de más alto nivel deben mantenerse como coordinadoras, y no viceversa. De esta forma, se promueve la reutilización de componentes y la flexibilidad en la estructura del programa, ya que se permite la desconexión de los componentes de bajo nivel de la implementación general del programa.

Dado que el sistema presenta una arquitectura de n-capas, donde las capas inferiores no “invocan” a las capas superiores, el principio Hollywood se aplica.

Principio Abierto-Cerrado

El Principio Abierto-Cerrado es un principio fundamental de diseño de software que establece que una clase debe estar abierta para su extensión pero cerrada para su modificación. En otras palabras, las entidades de software deben permitir la extensión de su comportamiento sin necesidad de modificar su código fuente existente.

Este principio promueve la creación de sistemas que puedan adaptarse a nuevas funcionalidades y requisitos a través de la extensión de clases existentes o la creación de nuevas clases, en lugar de realizar cambios directos en el código original. Esto se logra mediante la introducción de puntos de extensión y la aplicación de abstracciones y patrones de diseño que permitan la incorporación de nuevas funcionalidades de forma modular y sin alterar el funcionamiento existente.

En la Figura 10 se evidencia la aplicación de este principio, dado que las clases que heredan de **BaseService** no modifican el comportamiento del mismo, sino que lo extienden, mediante el uso de tipos genéricos y porlimorfismo.

Patrones de diseño

En el desarrollo de software, los patrones de diseño son soluciones probadas y reutilizables para problemas comunes que los desarrolladores enfrentan al diseñar aplicaciones. Estos patrones ofrecen una guía estructurada para abordar situaciones específicas, facilitando la creación de sistemas flexibles, mantenibles y eficientes. En este capítulo, exploraremos varios patrones de diseño fundamentales que son ampliamente utilizados en la industria de la programación y que pueden aplicarse a diversos contextos de desarrollo de software [49, 50, 51].

En este capítulo se detallan los patrones de diseño identificados en la solución propuesta.

Patrón de injección de dependencias

El patrón de Inyección de Dependencias (Dependency Injection) es un patrón de diseño utilizado en el desarrollo de software que se centra en la separación de la creación y la gestión de dependencias de un componente de software. En lugar de que un

componente cree directamente las dependencias que necesita, estas se proporcionan externamente (inyectadas) al componente, lo que permite una mayor flexibilidad, desacoplamiento y reutilización de código [50, 51, 52].

En primer lugar, se ha implementado la Inyección de Dependencias para gestionar la conexión con servicios externos, como la base de datos, y obtencion de credenciales. Esta técnica nos permite definir las dependencias requeridas por cada componente de la aplicación de manera clara y estructurada, facilitando la configuración y la sustitución de estos servicios en el futuro.

En la Figura 4 se presenta la injección de dependencias en el endpoint de crear usuario para la obtención de los credenciales del usuario que está accediendo y para obtener el objeto **Session** con el que se realizan operaciones sobre la base de datos de logs.

```python
@self.router.post("/users", status_code=status.HTTP_201_CREATED)
@auth_service.ldap_auth(Config.ADMIN_GROUP)
def user_create(
    user_create_input: UserCreateInput,
    credentials: HTTPBasicCredentials = Depends(self.security),
    db: Session = Depends(self.get_db),
):
    user = user_service.user_create(
        credentials=credentials, user_create_input=user_create_input
    )
    logs_service.generate_user_created_log(
        target=user["sAMAccountName"], actor=credentials.username, db=db
    )
    return JSONResponse(status_code=status.HTTP_201_CREATED, content=user)
```

Figura 4 Injección de dependencias en el endpoint de crear usuario

Patrón Proxy

En el desarrollo de aplicaciones web modernas, el patrón de diseño Proxy juega un papel crucial al permitir la comunicación eficiente entre diferentes componentes del sistema y gestionar las interacciones entre el cliente y el servidor. En el contexto específico del uso de SvelteKit como frontend y FastAPI como backend, el patrón Proxy es fundamental para redirigir y gestionar las solicitudes del cliente hacia el servidor backend de forma transparente.

En esta configuración, SvelteKit actúa como el cliente que genera las solicitudes de los usuarios y FastAPI actúa como el servidor que procesa y responde a esas solicitudes. El patrón Proxy se convierte en un intermediario inteligente que facilita la comunicación entre el frontend y el backend, permitiendo una interacción fluida y eficaz entre ambos componentes.

El Proxy en este caso específico se encarga de redirigir las solicitudes del cliente desde SvelteKit hacia el backend de FastAPI, proporcionando una capa adicional de abstracción que gestiona la comunicación, mejora el rendimiento y controla el flujo de datos entre el frontend y el backend. Esto ayuda a centralizar la lógica de comunicación y a encapsular las complejidades del intercambio de datos, mejorando la escalabilidad y mantenibilidad del sistema en su conjunto.

En la Figura 5 se muestra el flujo que sigue una petición X en el sistema, evidienciando la aplicación del patrón proxy dado que el servidor de sveltekit redirecciona las peticiones a las rutas correspondientes del backend de FastAPI.

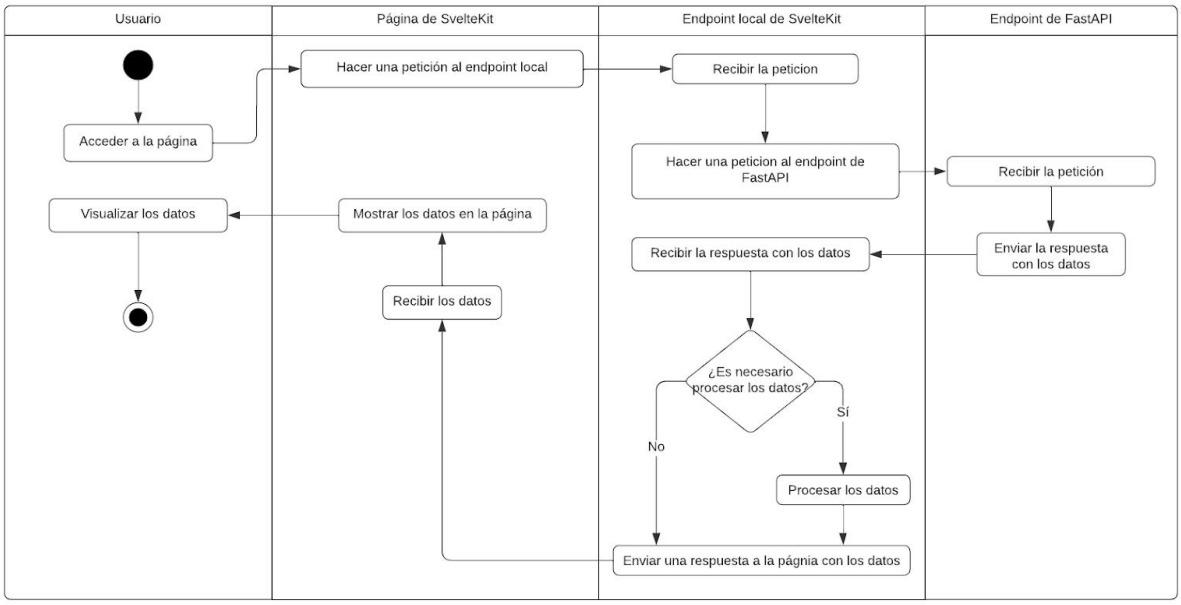


Figura 5 Flujo de una petición en el sistema

Patrón observador

El patrón de diseño Observer es una poderosa técnica utilizada en el desarrollo de software para establecer una relación de dependencia uno a muchos entre objetos, de manera que, cuando un objeto cambia su estado, todos los objetos dependientes sean

notificados y actualizados automáticamente. En el caso específico de SvelteKit, el patrón Observer es fundamental en el manejo de Stores y la reactividad de la interfaz de usuario.

En el entorno de SvelteKit, las Stores desempeñan un papel clave al actuar como contenedores reactivos de datos que pueden ser observados por múltiples componentes de la aplicación. Cuando una Store sufre un cambio en su estado, los componentes que están suscritos a esta Store serán notificados y se actualizarán de forma automática, manteniendo la coherencia de la aplicación y mejorando la eficiencia en la comunicación de datos.

El patrón Observer permite la implementación de un flujo de datos reactivo en SvelteKit, donde los cambios en el estado de una Store desencadenan automáticamente acciones en los componentes que observan esa Store. Esto facilita la actualización dinámica de la interfaz de usuario en respuesta a los cambios de datos y eventos en la aplicación, proporcionando una experiencia de usuario más ágil y dinámica.

En la Figura 6 se muestra un fragmento de código de la cómo se declaran variables observables, estas variables en el contexto de Sveltekit se denominan **stores**. Una vez declarada una store los componentes pueden usarla para mutar su estádo automáticamente una vez el valor de la store cambia. En la Figura 7 se muestra un componente que se muestra solo si el valor de la store de la Figura 6 tiene valor **true**. Por lo tanto cualquier componente que se modifique el valor de esta store provoca una mutación en el estado del componente de la Figura 7.

```
export const loading = writable<boolean>(false);
```

Figura 6 Declaración de una variable observable en Svelte

```
<script lang="ts">
    import { fade } from 'svelte/transition';
    import { loading } from '$stores/regularStores';
</script>

{#if $loading}
    <div
        class="fixed h-full w-full flex justify-center align-middle z-[999] bg-[#00000057] cursor-wait"
        transition:fade={{ duration: 200, delay: 0 }}
    >
        <img src="img/loader.svg" alt="loading..." width="350" height="350" />
    </div>
{/if}
```

Figura 7 Reación al cambio de una variable observable

Patrón decorador

En el desarrollo de aplicaciones web con FastAPI, el patrón de diseño Decorador desempeña un papel esencial al permitir la extensión y la personalización de funcionalidades en las rutas de nuestra API. En este sistema específico, los decoradores se utilizan de diversas maneras para mejorar la modularidad, la seguridad y la eficiencia de nuestro sistema.

En primer lugar, los decoradores se utilizan como la forma de declarar rutas en FastAPI, proporcionando una manera elegante y estructurada de definir y configurar las distintas rutas de nuestra API. Estos decoradores permiten especificar las operaciones HTTP asociadas a cada ruta, así como los parámetros y validaciones necesarios para cada solicitud.

Además, se ha creado un decorador personalizado para manejar la autorización en las rutas de nuestro sistema. Este decorador se encarga de verificar las credenciales del usuario antes de permitir el acceso a ciertas rutas, garantizando la seguridad y el control de acceso en nuestra API.

Por último, se ha implementado un decorador que condiciona la ejecución de la función decorada a la configuración de la base de datos para los logs del sistema. Este decorador proporciona una capa adicional de control y flexibilidad, asegurando que ciertas funcionalidades solo se activen si se cumplen ciertas condiciones preestablecidas.

En la Figura 8 se muestra la funcion del decorador **plug,** este decorador condiciona la ejecucion de la funcion que extiende a si la base de datos de logs esta configurada o no. Este decorador evita la repeticion de código en las funciones de este servicio haciéndolo mas simple.

```python
def plug():
    def _my_decorator(view_func):
        def _decorator(*args, **kwargs):
            if Config.has_log_database():
                return view_func(*args, **kwargs)

        return wraps(view_func)(_decorator)

    return _my_decorator


@plug()
def get_logs(db: Session):
    logs = db.query(LogModel).order_by(desc(LogModel.time)).all()
    return _get_logs_as_schemas(logs)
```

Figura 8 Decorador "plug"

En la Figura 9 se muestra el fragmento de código correspondiente al decorador **ldap_auth**. Este decorador se encarga de manejar la autorizacion en los endpoints, obteniendo los credenciales del usuario que está intentando acceder y verificando si se encuentra en un grupo del directorio que tiene permisos de acceso para ese recurso.

Este decorador permite simplificar la lógica de la autorizacion en cada endpoint evitando así la repetición y favoreciendo la simplicidad.

En la Figura 10 se muestra el uso de este decorador en el endpoint encargado de crear un usuario, en este caso solo los usuarios que estén en el grupo de administradores pueden acceder al recurso. En la Figura 6 tambien se ve el uso de los decoradores de FastAPI usados principalmente en la declaracion de rutas.

```python
def ldap_auth(group_name="Domain Users"):
    def _my_decorator(view_func):
        def _decorator(*args, **kwargs):
            _credentials = dict(kwargs["credentials"])
            if not _credentials:
                exceptions.unauthorized_exception()
            username = _credentials["username"]
            password = _credentials["password"]
            user_credentials = HTTPBasicCredentials(
                username=username, password=password
            )
            admin_ldap_service = auth(credentials=user_credentials, use_d=True)
            if group_name:
                members = admin_ldap_service.get_members(group_name)
                if members is None:
                    exceptions.group_in_auth_decorator_not_found()
                user = admin_ldap_service.get_entry_simple({"sAMAccountName": username})
                if not user or user["distinguishedName"] not in members:
                    exceptions.unauthorized_exception()
            return view_func(*args, **kwargs)
        return wraps(view_func)(_decorator)
    return _my_decorator
```

Figura 9 Decorador "ldap_auth"

```python
@self.router.post("/users", status_code=status.HTTP_201_CREATED)
@auth_service.ldap_auth(Config.ADMIN_GROUP)
def user_create(
    user_create_input: UserCreateInput,
    credentials: HTTPBasicCredentials = Depends(self.security),
    db: Session = Depends(self.get_db),
):
    user = user_service.user_create(
        credentials=credentials, user_create_input=user_create_input
    )
    logs_service.generate_user_created_log(
        target=user["sAMAccountName"], actor=credentials.username, db=db
    )
    return JSONResponse(status_code=status.HTTP_201_CREATED, content=user)
```

Figura 10 Uso de los decoradores de FastAPI para las rutas y del decorador "ldap_auth"

Patrón singleton

En el desarrollo de nuestro sistema, el patrón de diseño Singleton desempeña un papel crucial en la gestión de instancias únicas de ciertos objetos y componentes clave. En este contexto específico, el Singleton se utiliza para garantizar que ciertas configuraciones y recursos importantes sean compartidos de manera coherente en todo el sistema, evitando la creación de múltiples instancias innecesarias y asegurando la coherencia y la integridad de los datos.

En último lugar, el Singleton se encarga de mantener una única instancia de ciertos objetos esenciales en el sistema, como un gestor de usuarios o un gestor de grupos. Esto asegura la coherencia en la manipulación de estos objetos dentro de la aplicación y evita duplicaciones innecesarias que puedan afectar al funcionamiento del sistema.

En el sistema se tienen varias clases que implementan este patrón, esto se evidencia en la Figura 13, donde **UserService**, **GroupService** y **AuthService** son singletons, dado que solo mantienen una única instancia.

Conclusiones

Al finalizar trabajo se llegó a las siguientes conclusiones:

- El estudio detallado de arquitecturas, tecnologías y principios de diseño en el desarrollo de sistemas informáticos ha permitido comprender la importancia de tomar decisiones acertadas en cada etapa del proceso. La elección de una arquitectura adecuada y un stack tecnológico bien fundamentado impacta significativamente en la eficiencia, escalabilidad y mantenibilidad de las aplicaciones.

- La investigación sobre tecnologías específicas como Directorio Activo, LDAP, bases de datos relacionales como Postgres y NoSQL como MongoDB, junto con frameworks de frontend como SvelteKit y backend como FastAPI, ha revelado las ventajas y consideraciones clave para su implementación en diferentes contextos de desarrollo de software.

- Los problemas frecuentes en el desarrollo de sistemas, desde la gestión de dependencias hasta la implementación de estilos y patrones de diseño, han sido identificados y abordados con soluciones concretas. Se ha enfatizado la necesidad de mantener buenas prácticas de diseño para garantizar la cohesión, la modularidad y la mantenibilidad del código.

- La selección cuidadosa de estilos y patrones de diseño, como el estilo llamada-retorno, el Patrón de Acceso a Datos, el Principio de responsabilidad única y el Patrón Singleton, demuestran ser fundamentales para la creación de sistemas robustos y escalables, al tiempo que facilitan la adaptación a cambios futuros en los requisitos del sistema.

La adhesión a principios de diseño clave como el Principio Hollywood, la Ley de Demeter y el Principio Abierto-Cerrado, junto con la implementación de patrones de diseño como la inyección de dependencias y el patrón Proxy, son elementos esenciales en la construcción de sistemas software de calidad que sean flexibles y fáciles de mantener.

Referencias bibliográficas

[1] R. Mercurio and B. Merrill, “Beginning Microsoft 365 collaboration apps: working in the Microsoft cloud”. Berkeley, CA: Apress, 2021.

[2] J. B. Keegan, “An Overview Of Collaboration Software" Sep. 18, 2019. https://www.forbes.com/sites/forbestechcouncil/2019/09/18/an-overview-of-collaboration-software/?sh=3ac85b342cee (accessed Jun. 11, 2023).

[3] Google, “Manage Workspace with Admin Dashboard - Google Workspace.” https://workspace.google.com/intl/en/products/admin/ (accessed Jun. 22, 2023).

[4] Microsoft, “Single Sign-On (SSO): Secure App Access Solutions | Microsoft Security.” https://www.microsoft.com/en-us/security/business/identity-access/azure-active-directory-single-sign-on (accessed Jun. 21, 2023).

[5] V. Beltran, “Characterization of web single sign-on protocols" IEEE Commun. Mag., vol. 54, no. 7, pp. 24–30, Jul. 2016, doi: 10.1109/MCOM.2016.7514160.

[6] E. C. Dias and R. de F. Ribeiro, “Single sign-on: an information security approach" in Proceedings of the 13th CONTECSI International Conference on Information Systems and Technology Management, 2016, vol. 13, pp. 2874–2896, doi: 10.5748/9788599693124-13CONTECSI/PS-4061.

[7] Avangenio, “Productos propios" 2023. https://avangenio.com/productos-propios/ (accessed Jun. 21, 2023).

[8] Usmanu Danfodiyo University, M. K. Hamza, H. Abubakar, Y. M. Danlami, “Identity and Access Management System: a Web-Based Approach for an Enterprise" PoS, vol. 4, no. 11, pp. 2001–2011, Nov. 2018, doi: 10.22178/pos.40-1.

[9] P. Sharma, V. K. Sihag, “Hybrid Single Sign-On Protocol for Lightweight Devices" in 2016 IEEE 6th International Conference on Advanced Computing (IACC), 2016, pp. 679–684, doi: 10.1109/IACC.2016.131.

[10] N. Naik, P. Jenkins, “A secure mobile cloud identity: criteria for effective identity and access management standards" presented at the 2016 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2016, pp. 89–90, doi: 10.1109/MobileCloud.2016.22.

[11] RedHat, “What is lightweight directory access protocol (LDAP) authentication?" Jun. 03, 2022. https://www.redhat.com/en/topics/security/what-is-ldap-authentication (accessed Jun. 21, 2023).

[12] Microsoft, “LDAP authentication with Azure Active Directory" Oct. 01, 2023. https://learn.microsoft.com/en-us/azure/active-directory/fundamentals/auth-ldap (accessed Jun. 22, 2023).

[13] J. Schwenk, “Web security and Single Sign-On protocols" in *Guide to Internet Cryptography: Security Protocols and Real-World Attack Implications*, Cham: Springer International Publishing, 2022, pp. 467–503.

[14] Apache, “Welcome to Apache Directory Studio" Jul. 24, 2021. https://directory.apache.org/studio/ (accessed Jun. 22, 2023).

[15] Oracle, “Oracle Directory Server Enterprise Edition Directory.” https://docs.oracle.com/en/cloud/paas/identity-cloud/idcsc/odsee.html (accessed Jun. 22, 2023).

[16] A. Bartlett, “Samba 4-Active Directory”, *Samba. Org., 78*, vol. 78, 2005.

[17] Samba, “What is Samba?” https://www.samba.org/samba/what_is_samba.html (accessed Jun. 22, 2023).

[18] T. Carpenter, “Microsoft Windows Server Administration Essentials”, 1st ed. Indianapolis, Ind: Sybex, 2011, p. 400.

[19] J. Cooper, “The Book of Webmin: Or How I Learned to Stop Worrying and Love UNIX*”*, 1st ed. San Francisco: No Starch Press, 2003, p. 312.

[21] V. S. Garófalo Jerez, “VicentGJ/AD-webmanager: A web interface for administration of Active Directory Domains, made in Python, with focus on ease of use and simplicity.” https://github.com/VicentGJ/AD-webmanager/ (accessed Jun. 22, 2023).

[22] L. Mattias , “DOM benchmark comparison of the front-end JavaScript frameworks React, Angular, Vue, and Svelte”, Åbo Akademi University, 2020.

[23] V. Fernández Palacio , “Desarrollo de gateway de seguridad en python con FastAPI”, UNIVERSIDAD DE CANTABRIA, Mayo 2022.

[24] PostgreSQL Community, “PostgreSQL: About” https://www.postgresql.org/about/, 2024 (Accessed Feb. 25, 2024).

[25] IBM, “What is MongoDB” https://www.ibm.com/topics/mongodb (accessed Feb. 25, 2024)

[26] D. Wernersson and V. Sjölund, “Choosing a Rendering Framework : A Comparative Evaluation of Modern JavaScript Rendering Frameworks,” Dissertation, 2023.

[27] “SvelteKit, Web development, Streamlined,” Svelte.dev, 2023. https://kit.svelte.dev/ (accessed Feb. 25, 2024).

[28] V. Patel, “Analyzing the Impact of next. JS on Site Performance and SEO,” International Journal of Computer Applications Technology and Research, vol. 12, pp. 24–27, 2023.

[29] K. Konshin, Next. js quick start guide: Server-side rendering done right. Packt Publishing Ltd, 2018.

[30] O. Omole, “Nuxt. js: a minimalist framework for creating universal vue. js apps,” 2019.

[31] P. Halliday, Vue. js 2 design patterns and best practices: Build enterprise-ready, modular vue. js applications with vuex and nuxt. Packt Publishing Ltd, 2018.

[32] Malhar Lathkar, High-Performance Web Apps with FastAPI. 2023.

[33] J. H. Peralta, Microservice APIs: Using Python, Flask, FastAPI, OpenAPI and More. Simon and Schuster, 2023. Accessed: Feb. 25, 2024. [Online]. Available: https://books.google.com/books?hl=en&lr=&id=EDehEAAAQBAJ&oi=fnd&pg=PA1&dq=fastapi&ots=fm94xWBYf4&sig=tW2k2pTFIdLkDkmo9V0JZvx4Ieo#v=onepage&q=fastapi&f=false

[34] J. Forcier, P. Bissex, and W. J. Chun, Python Web Development with Django. Addison-Wesley Professional, 2008. Accessed: Feb. 25, 2024. [Online]. Available: https://books.google.com/books?hl=en&lr=&id=M2D5nnYlmZoC&oi=fnd&pg=PT31&dq=django&ots=v_SLLw9SRS&sig=QA93gZVKBNQ4QlktEAM5DpbzqYY

[35] S. Dauzon, A. Bendoraitis, and A. Ravindran, Django: Web Development with Python. Packt Publishing Ltd, 2016. Accessed: Feb. 25, 2024. [Online]. Available:

https://books.google.com/books?hl=en&lr=&id=vKjWDQAAQBAJ&oi=fnd&pg=PP1&dq=django&ots=2oNICBj1wN&sig=gjRo7BpAyUvpV4og0ScdUjd7sJE

[36] M. Grinberg, Flask Web Development. “O’Reilly Media, Inc.,” 2018. Accessed: Feb. 25, 2024. [Online]. Available: https://books.google.com/books?hl=en&lr=&id=cVlPDwAAQBAJ&oi=fnd&pg=PT25&dq=flask&ots=xPB_jq-sbR&sig=_oIIrWoUOaiFoCDQK2dOa3lJMkw

[37] D. Ghimire, “Comparative Study on Python Web frameworks: Flask and Django,” www.theseus.fi, 2020. https://www.theseus.fi/handle/10024/339796 (accessed Aug. 25, 2021).

[38] M. María, M. Vitturini, and D. Arquitectónico, “Análisis Y Diseño De Sistemas - 1er.Cuatrimestre De 2012. ANÁLISIS Y DISEÑO DE SISTEMAS Clase 23: Conceptos De Diseño Arquitecturas De SW,” 2012. Accessed: Feb. 25, 2024. [Online]. Available: http://cs.uns.edu.ar/~td/ayds2012/downloads/Clases/ADS_23_2012-%20Conceptos%20de%20Disenio%20II.pdf

[39] R. Álvarez León, “Aplicación Web Para La Gestión Del Alojamiento En La Residencia Estudiantil De La CUJAE,” repositorio.uci.cu, Nov. 01, 2022. https://repositorio.uci.cu/handle/123456789/10559 (accessed Feb. 25, 2024).

[40] N. S. Alseelawi, E. K. Adnan, H. T. Hazim, H. Alrikabi, and K. Nasser, Design and Implementation of an E-learning Platform Using N-Tier Architecture. International Association of Online Engineering, 2020. Available: https://www.learntechlib.org/p/216473/

[41] T. Erl, “Service-Oriented Architecture a Field Guide to Integrating XML and Web Services,” Feb. 2005. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=9ea109acd0e47c29a0ca9316fd67c554d83d1543

[42] Amazon Web Services, “What Is SOA? - SOA Architecture Explained - AWS,” Amazon Web Services, Inc., 2024. https://aws.amazon.com/what-is/service-oriented-architecture/ (accessed Feb. 25, 2024).

[43] A. Rotem-Gal-Oz, SOA Patterns. Simon and Schuster, 2012. Accessed: Feb. 25, 2024. [Online]. Available: https://books.google.com/books?hl=en&lr=&id=BzozEAAAQBAJ&oi=fnd&pg=PT16&dq=

SOA+service+data+access+pattern&ots=Cwuvg_U1Dx&sig=osEOr8IRnbWfUa0a263_NHFy8hs

[44] A. Demange, Naouel Moha, and G. Tremblay, “Detection of SOA Patterns,” Lecture Notes in Computer Science, pp. 114–130, Jan. 2013, doi: https://doi.org/10.1007/978-3-642-45005-1_9.

[45] L. Resende, “Handling Heterogeneous Data Sources in a SOA Environment with Service Data Objects (SDO),” Jun. 2007, doi: https://doi.org/10.1145/1247480.1247582.

[46] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice. Addison-Wesley Professional, 2003. Accessed: Feb. 25, 2024. [Online]. Available: https://books.google.com/books?hl=en&lr=&id=mdiIu8Kk1WMC&oi=fnd&pg=PA1&dq=design+principles+software+architecture&ots=UgJ4Wbi8TN&sig=38C1ytbVsnPumcL6o9GToZEJxp0

[47] P. Avgeriou and U. Zdun, “Architectural Patterns Revisited -A Pattern Language,” 2005. Available: http://eprints.cs.univie.ac.at/2698/1/ArchPatterns.pdf

[48] A. Ampatzoglou et al., “Applying the Single Responsibility Principle in Industry,” Proceedings of the Evaluation and Assessment on Software Engineering, Apr. 2019, doi: https://doi.org/10.1145/3319008.3320125.

[49] A. Shvets, “Refactoring and Design Patterns,” refactoring.guru, 2024. https://refactoring.guru

[50] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns : elements of reusable object-oriented software. Boston: Addison-Wesley, 1994.

[51] R. Harmes and D. Diaz, Pro JavaScript Design Patterns. Berkeley, Ca: Apress ; New York, Ny, 2008.

[52] Dhananjay Prasanna, Dependency Injection. Simon and Schuster, 2009.

Anexos

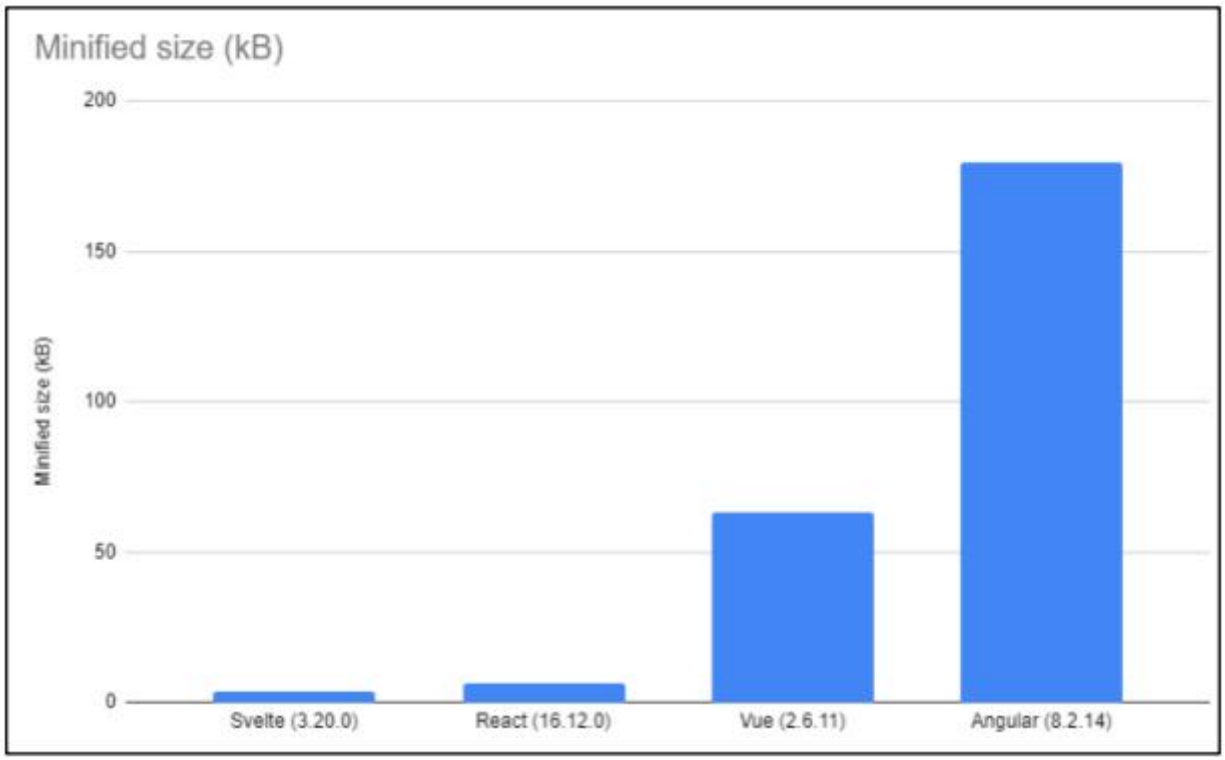


Figura 11 Tamaño minificado de cada framework en el registro de npm

Framework Control structure	React v16.12.0 reference	Vue v2.6.11 reference	Angular v8.2.14 getElementById	Svelte v3.20.0 getElementById
Attempt 1	16.51	22.25	6.71	0.11
Attempt 2	16.06	21.28	5.64	0.11
Attempt 3	17.48	22.77	5.44	0.11
Attempt 4	16.59	23.34	6.32	0.11
Attempt 5	16.81	21.41	6.43	0.11
Attempt 6	16.55	18.61	6.57	0.12
Attempt 7	15.61	23.60	5.70	0.11
Attempt 8	16.40	22.23	6.12	0.11
Attempt 9	16.35	22.66	5.61	0.12
Attempt 10	17.39	24.16	6.29	0.11
Average (ms)	**16.58**	**22.23**	**6.08**	**0.11**

Figura 12 Rendimiento editando un elemento entre 10000 elementos presentes en el DOM

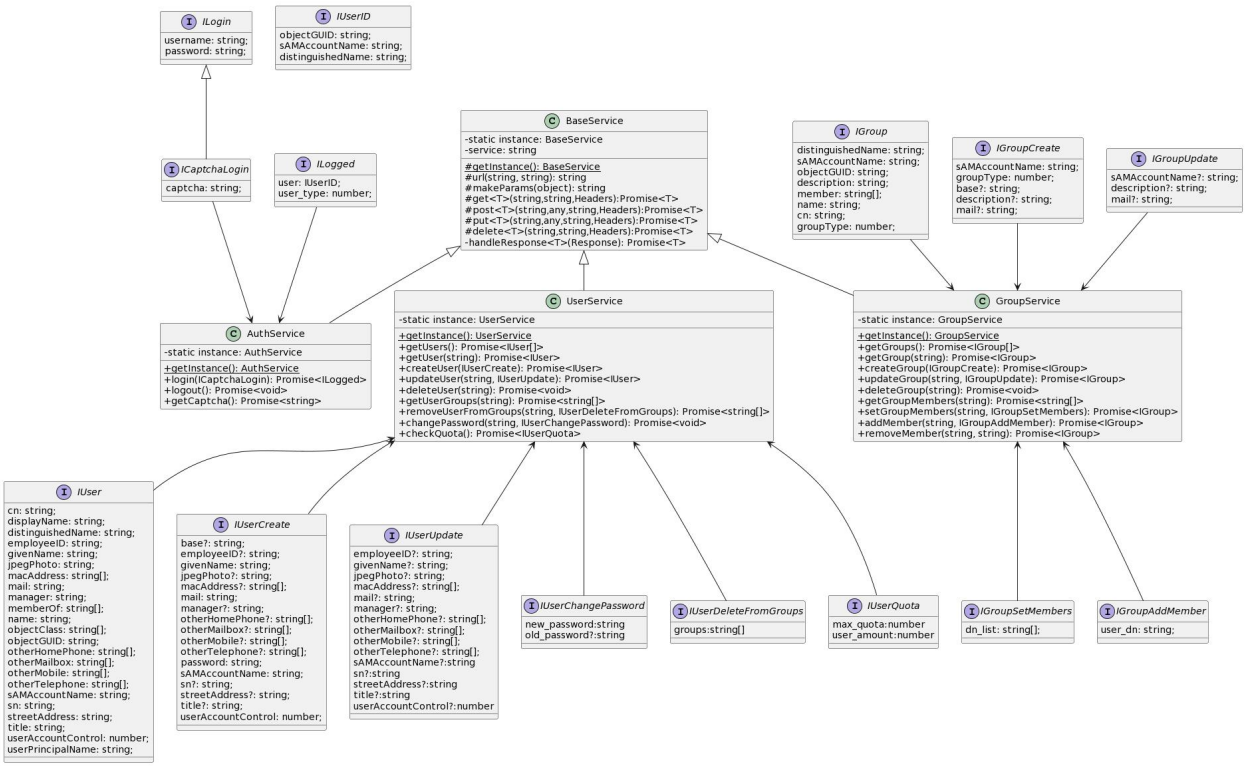


Figura 13 Diagrama de clases del frontend