AGRADECIMIENTOS

A mi familia, por seguir animándome a continuar creciendo en la Universidad de Almería. A mis compañeros, porque desde el Grado hemos seguido apoyándonos continuamente en nuestras vidas. A mis amigos, por estar conmigo tanto en las buenas como en las malas. Al profesorado de esta universidad, donde gracias a ellos he podido adquirir estos nuevos conocimientos en esta ingeniería, y también y sobre todo a Manuel Torres Gil, por darme la oportunidad de realizar este proyecto donde he podido poner a prueba toda mi experiencia profesional previa junto a lo aprendido en este máster para mi desarrollo y desempeño personal y laboral.

ACKNOWLEDGMENTS

To my family, for continuing to encourage me to continue growing at the University of Almería. To my classmates, for constantly supporting each other throughout our lives since the Bachelor's degree. To my friends, for being there for me through thick and thin. To the faculty at this university, thanks to whom I have been able to acquire this new knowledge in this engineering field. And also, and above all, to Manuel Torres Gil, for giving me the opportunity to complete this project, where I have been able to put all my previous professional experience to the test, along with what I have learned in this master's degree, to help me develop and perform personally and professionally.

RESUMEN

Este Trabajo Fin de Máster propone una solución para reforzar la seguridad en aplicaciones que no disponen de sistemas propios de autenticación y control de accesos. Esta propuesta de solución consiste en crear un sistema externo que gestione de forma centralizada a los usuarios, sus permisos y credenciales, sin un gran impacto sobre el código fuente nativo, consiguiendo una capa externa de seguridad para este tipo de aplicaciones delegada en servicios externos fácilmente integrables y escalables.

ABSTRACT

This Master's Thesis proposes a solution to strengthen security in applications that lack their own authentication and access control systems. This proposed solution consists of creating an external system that centrally manages users, their permissions, and credentials, without significant impact on the native source code. This provides an external layer of security for these types of applications, delegated to easily integrable and scalable external services.

Índice de contenidos

Índice de figuras

Capítulo 01: Introducción

1.1. Motivación

En la actual era digital, la tecnología avanza a un ritmo vertiginoso, lo que conlleva un aumento significativo en la complejidad y funcionalidad de las aplicaciones. Este crecimiento no solo amplía sus capacidades, sino que también incrementa su exposición a potenciales amenazas de seguridad [1]. Las aplicaciones modernas manejan grandes volúmenes de datos sensibles y requieren mecanismos robustos para proteger la integridad, confidencialidad y disponibilidad de la información.

Sin una gestión adecuada de la seguridad, las aplicaciones son vulnerables a ataques como el robo de credenciales, acceso no autorizado y exfiltración de datos. Implementar internamente sistemas de autenticación y gestión de credenciales puede ser un proceso laborioso y propenso a errores, especialmente si no se cuenta con experiencia especializada en seguridad.

Delegar estas funciones críticas a plataformas especializadas para autenticación, autorización [2] y gestión segura de secretos y credenciales [3] permite a los desarrolladores centrarse en la lógica principal del negocio. Esto no solo mejora la eficiencia en el desarrollo, sino que también garantiza que las prácticas de seguridad estén alineadas con los estándares y mejores prácticas de la industria [4], reduciendo significativamente los riesgos asociados a la seguridad de la información.

1.2. Objetivo general

El objetivo de este Trabajo Fin de Máster es diseñar e implementar una solución integral que proporcione un sistema de autenticación, autorización y gestión segura de credenciales para aplicaciones que carecen de estas funcionalidades de forma nativa. Al ofrecer una plataforma especializada para estas funciones críticas, se busca permitir a los desarrolladores centrarse en la lógica principal del negocio, mejorando la eficiencia en el desarrollo y garantizando que las prácticas de seguridad estén alineadas con los estándares y mejores prácticas de la industria. Incluso para el caso de aplicaciones ya en producción que carecen de este sistema o servicio de autentificación, tener la posibilidad de integrarla en ellas sin una modificación o relativamente mínimo del código fuente original. Esto reducirá significativamente los riesgos asociados a la seguridad de la información.

En concreto, esto se realizará haciendo uso de la combinación de las tecnologías Keycloak y Vault para la creación de un sistema de autenticación, autorización y gestión segura de credenciales para su integración aplicaciones que carecen de estas funcionalidades de forma nativa, combinando con la herramienta utilizando Docker como motor para poder incluir en contenedores estos servicios que componen este sistema.

1.3. Objetivos específicos

Los objetivos para alcanzar a nivel técnico son los siguientes:

- Configurar un servidor de identidad para autenticación y autorización. Se implementará un servidor que actúe como proveedor de identidad (IdP), donde se definirán y gestionarán usuarios, grupos y roles. Se configurarán clientes y recursos protegidos, estableciendo políticas

de acceso basadas en roles y atributos. Además, se integrará este servidor con las aplicaciones objetivo utilizando protocolos estándar como OpenID Connect [5].

- Utilizar una solución para la protección y gestión de credenciales sensibles. Se instalará y configurará un servicio dedicado a la gestión de secretos [6]. Se definirán políticas y controles de acceso para usuarios y aplicaciones que necesiten acceder a información confidencial. Se implementarán mecanismos de autenticación [7] como *tokens*, *App Role* o integración con el proveedor de identidad.
- Implementar ambas herramientas en un entorno aislado mediante contenedores. Se crearán imágenes personalizadas con las configuraciones específicas necesarias [8] para cada servicio almacenado en contenedores y se orquestarán de forma conjunta. Se garantizará que los contenedores se ejecuten en redes aisladas y seguras, gestionando adecuadamente los puertos y las comunicaciones internas entre ellos.
- Automatizar el despliegue con una solución basada en IaaC para garantizar escalabilidad. Se implementarán scripts de automatización de despliegue de toda la infraestructura necesaria para el funcionamiento y arranque de este nuevo entorno [9] que compone todo el nuevo sistema de autentificación, autentificación y gestión de credenciales.

1.4. Planificación

El proyecto se ha organizado llevando a cabo una serie de iteraciones o hitos, introduciendo desde un primer inicio tareas de investigación sobre el uso de las diferentes tecnologías que se aplican hasta el desarrollo final del proyecto, realizando revisiones del estado de cada uno de los hitos. Se ha llevado a cabo esta estrategia para que el proyecto durante sus fases de desarrollo con las siguientes características:

- Sea susceptible a continuos cambios.
- Se obtenga una mayor calidad del software.
- Permita una mayor productividad. Se consigue entre otras razones, gracias a la eliminación de la burocracia y a la motivación del equipo que proporciona el hecho de que sean autónomos para organizarse.
- Facilidades en las estimaciones de tiempos de las iteraciones.
- Reducción de posibles riesgos durante su avance. El hecho de llevar a cabo las funcionalidades de más valor en primer lugar y de conocer la velocidad de avance en el proyecto, se permiten detectar errores y riesgos de manera anticipada, pudiendo actuar y tomar decisiones de forma proactiva y consensuada.

El desarrollo del TFM se compone de las siguientes fases:

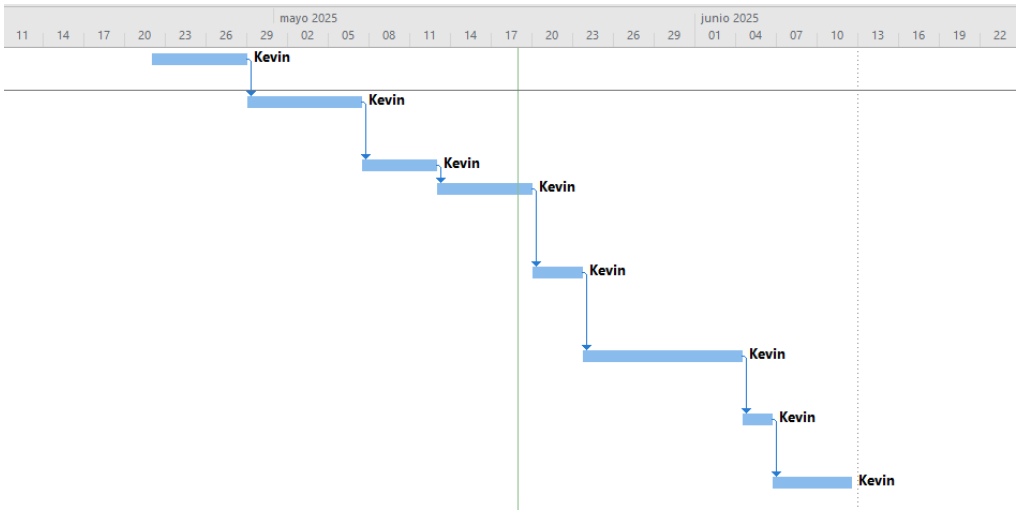

Figura 1. Diagrama de Gantt

1. Estudio y aprendizaje de Keycloak (40h).
2. Estudio de los diferentes métodos de autenticación de usuarios (50h).
3. Estudio y aprendizaje de Vault (30h).
4. Creación servicio gestión y autentificación de usuarios (40h).
5. Integración servicio gestión y autentificación de usuarios (30h).
6. Automatización configuración del nuevo servicio (60h).
7. Despliegue automatizado infraestructura (20h).
8. Documentación memoria TFM (30h).

1.5. Estructura del documento

El documento seguirá una estructura lógica y coherente para facilitar su lectura y comprensión:

- Capítulo 01: Introducción. En este primer punto se introduce el contexto general del trabajo, sus objetivos y motivación.
- Capítulo 02: Estado del arte. En este apartado se recoge el estado del arte, presentando estudios previos, tecnologías relacionadas y enfoques existentes.
- Capítulo 03: Descripción de herramientas. En él se describen las herramientas y tecnologías utilizadas a lo largo del desarrollo.
- Capítulo 04: Descripción de la solución desarrollada. En este punto se detallan el diseño y la implementación de la solución propuesta. Además, se exponen los resultados obtenidos y las pruebas realizadas sobre el trabajo realizado.
- Capítulo 05: Conclusiones y trabajo futuro. En esta sección se recogen las conclusiones extraídas del trabajo, así como posibles líneas de mejora o desarrollo futuro
- Capítulo 06: Bibliografía. Como último punto se presenta la bibliografía utilizada como base teórica y técnica del estudio.

Capítulo 02: Estado del arte

Este capítulo ha profundizado en la evolución y el estado actual de la seguridad en aplicaciones, destacando las limitaciones de los métodos tradicionales y la necesidad de adoptar tecnologías y prácticas modernas. La integración de herramientas como Keycloak y Vault, junto con enfoques como DevOps, DevSecOps y la orquestación con Kubernetes, proporciona un marco sólido para abordar los desafíos de seguridad en el panorama tecnológico actual. Además, el uso de *Infrastructure as Code* y herramientas proporcionadas por compañías como HashiCorp facilita la gestión eficiente y segura de infraestructuras en la nube, permitiendo a las organizaciones mejorar significativamente su postura de seguridad, proteger datos sensibles y cumplir con la normativa y estándares de seguridad.

2.1 Introducción a la seguridad en aplicaciones

La seguridad en aplicaciones es un pilar fundamental en el ámbito de la tecnología de la información y las comunicaciones. Con la creciente dependencia de sistemas informáticos en todos los sectores, desde la banca y la salud hasta la educación y el comercio electrónico, garantizar la protección de datos y recursos es más crítico que nunca. La evolución de la seguridad en aplicaciones ha pasado de simples validaciones de usuario y contraseñas estáticas a sistemas avanzados que incorporan inteligencia artificial, análisis de comportamiento y criptografía avanzada.

En las primeras etapas de la computación, la seguridad se centraba en el control de acceso físico y en la protección contra errores humanos. Sin embargo, con la expansión de las redes y el acceso remoto, las amenazas han aumentado exponencialmente. Los ataques cibernéticos modernos son altamente sofisticados, dirigidos y pueden causar daños significativos, incluyendo pérdidas financieras, robo de propiedad intelectual y daños a la reputación.

La CIA (*Confidenciality, Integrity and Avaliability*) sigue siendo la base de la seguridad de la información:

- Confidencialidad. Garantiza que la información sea accesible solo para las personas autorizadas.
- Integridad. Asegura que la información sea precisa y completa, y que no haya sido alterada de manera no autorizada.
- Disponibilidad. Asegura que los sistemas y datos estén disponibles para los usuarios autorizados cuando se necesiten.

Figura 2. CIA

La creciente adopción de tecnologías emergentes como la nube, IoT *(Internet of Things)* y la inteligencia artificial ha introducido nuevas formas de ataque. Además, la pandemia de COVID-19 aceleró la digitalización de muchos procesos de negocio de las distintas empresas y servicios online que ofrecen, aumentando aún más la exposición a amenazas cibernéticas. Por lo tanto, es imperativo que las organizaciones adopten enfoques proactivos y tecnologías avanzadas para proteger sus aplicaciones y datos.

2.2 Métodos tradicionales de autenticación y autorización

2.2.1 PAP *(Password Access Protocol)*

Los sistemas basados en contraseñas, cuyo protocolo se conoce como PAP, son uno de los métodos de autenticación más antiguos y ampliamente utilizados en la historia de la informática. Su origen se remonta a la década de 1960 con el desarrollo del Compatible *Time-Sharing System* (CTSS) en el Instituto Tecnológico de Massachusetts (MIT), donde se implementaron por primera vez contraseñas para separar y proteger los archivos de diferentes usuarios [1].

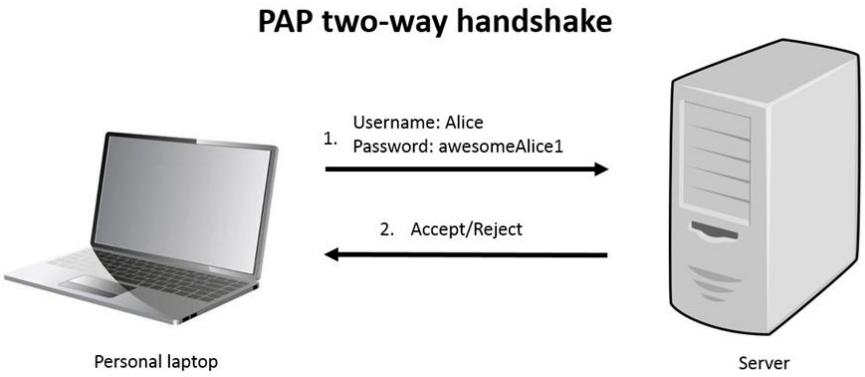

Figura 3. PAP

Antes de la introducción de las contraseñas, el acceso a los sistemas informáticos era limitado y controlado físicamente, lo que significaba que la seguridad dependía en gran medida del control de acceso al hardware, cuyas limitaciones eran las siguientes:

- **Reutilización de contraseñas:** Estudios indican que aproximadamente el 60% de las personas reutilizan las mismas contraseñas en múltiples sitios web [2]. Esto significa que, si un atacante obtiene una contraseña en un sitio, es muy probable que pueda acceder a otras cuentas del usuario.
- **Debilidad de contraseñas**: Los usuarios suelen elegir contraseñas fáciles de recordar, como "123456" o "contraseña", que son fácilmente vulnerables a ataques de fuerza bruta y de diccionario [3].
- **Phishing e ingeniería social:** Los atacantes pueden engañar a los usuarios para que revelen sus contraseñas mediante correos electrónicos fraudulentos o sitios web falsos [4].
- **Gestión y almacenamiento:** Las organizaciones deben almacenar las contraseñas de forma segura, generalmente mediante técnicas de *hashing*.

Estas limitaciones evidenciaron la necesidad de sistemas más seguros y centralizados para la gestión de identidades. A medida que las organizaciones crecían y los sistemas se volvían más complejos, se hizo evidente que las contraseñas por sí solas no eran suficientes para garantizar la seguridad.

2.2.2 *Lightweight Directory Access Protocol* (LDAP)

Para afrontar las limitaciones de los sistemas basados en contraseñas y mejorar la gestión de identidades, se desarrollaron protocolos que permitían una administración centralizada. En la década de 1990, el protocolo **LDAP** fue creado como una versión simplificada del protocolo **DAP** *(Directory Access Protocol)* utilizado en el modelo X.500 de directorios de la ISO [5]. Algunas de sus características a destacar son:

- **Almacenamiento centralizado de identidades:** Permite gestionar usuarios, grupos y permisos en un repositorio único.
- **Integración con sistemas existentes:** Compatible con diversas aplicaciones y sistemas operativos.
- **Escalabilidad:** Adecuado para organizaciones de diferentes tamaños.

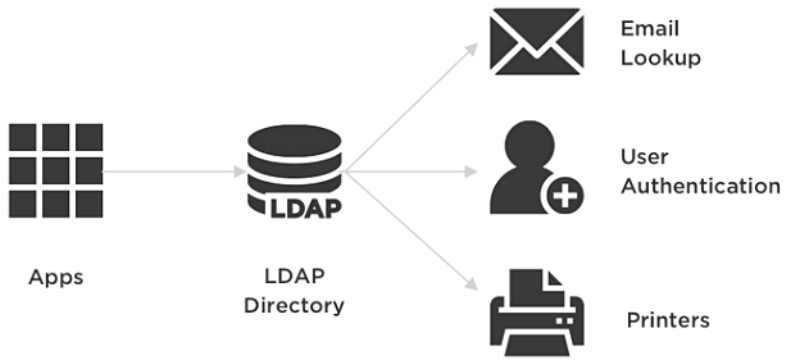

Figura 4. Esquema LDAP

LDAP mejoró significativamente la gestión de identidades al proporcionar un método estandarizado para acceder y mantener directorios de información. Sin embargo, aún dependía

principalmente de contraseñas para la autenticación, lo que no resolvía completamente los problemas de seguridad asociados. Además, su implementación y mantenimiento requerían conocimientos especializados, lo que podía ser un obstáculo para algunas organizaciones.

2.2.3 MFA *(Multifactor Authentication)*

Las crecientes amenazas y vulnerabilidades asociadas con las contraseñas llevaron a desarrollar métodos que añadieran capas adicionales de seguridad. A finales de los años 90 y principios de los 2000, las organizaciones comenzaron a implementar la autenticación multifactor conocida como MFA [6]. MFA surgió como respuesta directa a las limitaciones de la autenticación basada únicamente en contraseñas, proporcionando una forma de verificar la identidad del usuario utilizando múltiples factores independientes.

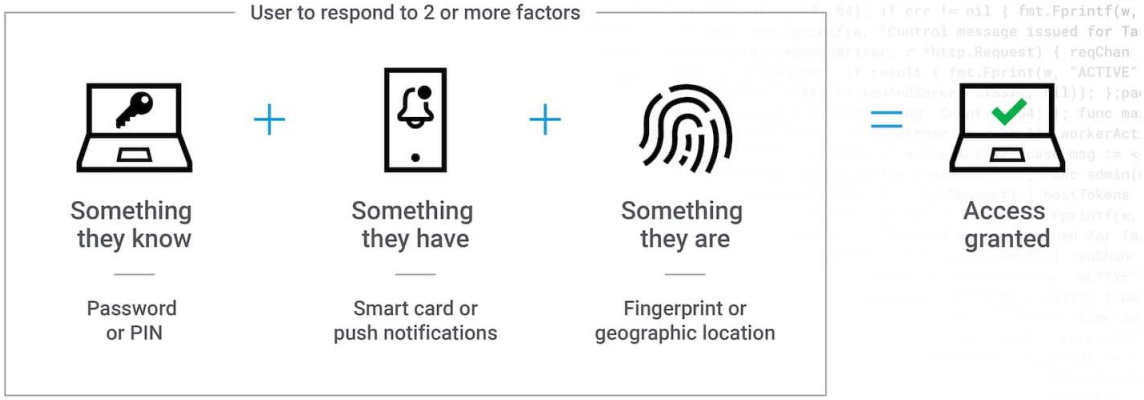

Figura 5. Protocolo MFA

La autenticación multifactor requiere que los usuarios proporcionen dos o más factores de autenticación independientes para verificar su identidad [7]. Los factores de autenticación se dividen en:

- Conocimiento: Algo que el usuario sabe (contraseña, PIN).
- Posesión: Algo que el usuario tiene (token, tarjeta inteligente, dispositivo móvil).
- Inherencia: Algo que el usuario es (huella dactilar, reconocimiento facial, patrón de iris).

MFA mejoró significativamente la seguridad al hacer más difícil para los atacantes comprometer una cuenta. Sin embargo, también introdujo desafíos en términos de costo, complejidad y experiencia del usuario. La necesidad de dispositivos adicionales y la posible resistencia de los usuarios a procesos más complicados limitaron su adopción generalizada en algunos contextos.

2.2.4 SAML *(Security Assertion Markup Language)* y *Single Sign-On* (SSO)

Con el aumento de servicios y aplicaciones web a principios de los 2000, surgió la necesidad de una forma segura y estandarizada de intercambiar información de autenticación y autorización

entre diferentes dominios. Las soluciones propietarias dificultaban la interoperabilidad, lo que llevó al desarrollo de SAML por parte de OASIS en 2002 [8].

SAML es un estándar abierto basado en XML para el intercambio de datos de autenticación y autorización entre dominios de seguridad [9], en donde se requiere de un proveedor de identidad (IdP) para autenticar al usuario y un proveedor de servicio (SP) para permitir su acceso.

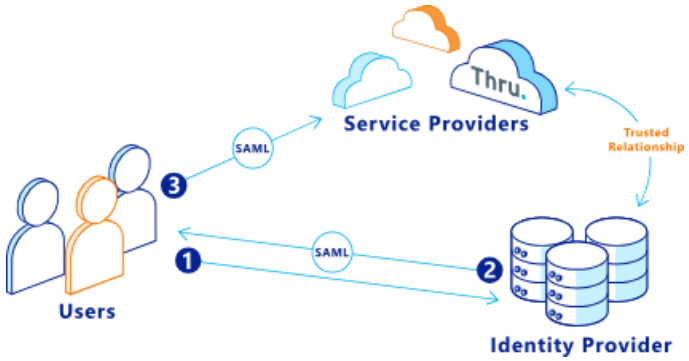

Figura 6. Protocolo SAML

Por otro lado, *Single Sign-On* (SSO) permite el acceso a múltiples aplicaciones con una sola autenticación [10], mejorando consigo también la seguridad y la experiencia del usuario, siguiendo un sistema de validación de usuario y acceso similar para la autentificación a la que se emplea con SAML.

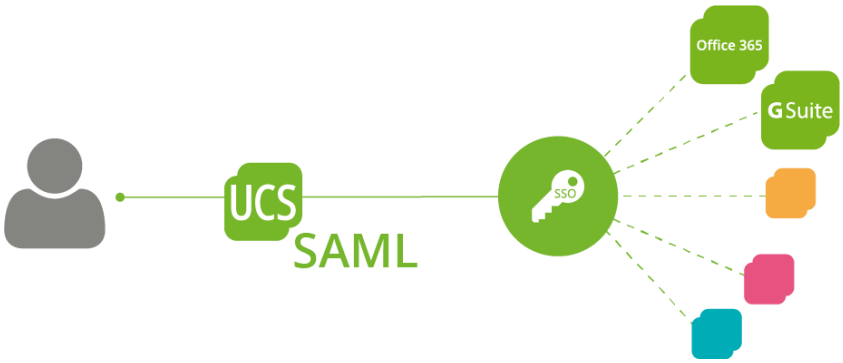

Figura 7. Single Sign-On (SSO)

SAML y SSO abordaron la necesidad de interoperabilidad y mejoraron la experiencia del usuario al reducir la cantidad de veces que necesitaban autenticarse en diferentes aplicaciones y servicios. Sin embargo, la complejidad en su implementación y la necesidad de una comprensión profunda del protocolo limitaron su adopción en algunos casos. Además, con la evolución de las aplicaciones web y móviles, surgieron nuevas necesidades que SAML no podía satisfacer completamente.

2.2.5 Problemas en soluciones tradicionales

A pesar de los avances, las soluciones tradicionales enfrentaron varios problemas en el contexto de las tecnologías emergentes:

- **Escalabilidad y adaptabilidad:** Con la aparición de aplicaciones móviles y servicios en la nube, los sistemas basados en contraseñas, LDAP y SAML mostraron limitaciones para escalar eficientemente y adaptarse a nuevos modelos arquitectónicos como los microservicios [11].
- **Experiencia del usuario:** Los procesos de autenticación podían ser engorrosos, afectando la usabilidad y la productividad.
- **Seguridad insuficiente:** Las amenazas avanzadas como el phishing dirigido y los ataques de intermediario (MITM) requerían soluciones más robustas.
- **Cumplimiento normativo:** Regulaciones como el GDPR impusieron requisitos estrictos sobre la protección de datos y la privacidad, que las soluciones tradicionales podían no cumplir adecuadamente.

Estas limitaciones impulsaron la búsqueda de nuevas tecnologías que pudieran satisfacer las necesidades de seguridad, usabilidad y cumplimiento en un entorno tecnológico en rápida evolución.

2.3 Tecnologías modernas

En respuesta a los desafíos presentados por las soluciones tradicionales, surgieron tecnologías modernas diseñadas para ofrecer mayor seguridad, flexibilidad y mejor experiencia de usuario. Estas tecnologías adoptan enfoques innovadores para la autenticación y autorización, adaptándose a las demandas actuales del mercado y las regulaciones vigentes.

2.3.1 *OAuth*

Con el auge de las redes sociales y las *APIs REST* a finales de la década de 2000, surgió la necesidad de permitir que aplicaciones de terceros accedieran a los datos del usuario sin exponer sus credenciales. En 2007, ***OAuth 1.0*** fue desarrollado para abordar esta necesidad [12]. Este protocolo permitió la delegación de acceso, permitiendo a los usuarios autorizar a aplicaciones a acceder a sus datos en otro servicio de forma segura, aunque el proceso de autentificación requería de unas firmas criptográficas bastante complejas de implementar y mantener, además de que no era fácilmente adaptable a las aplicaciones.

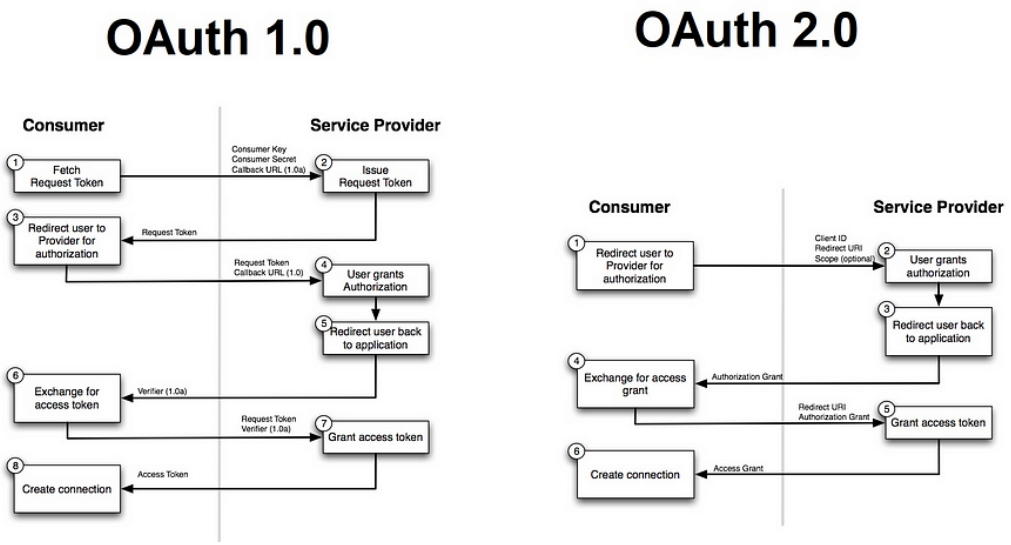

Figura 8. OAuth

Para superar las limitaciones de *OAuth 1.0*, apareció una evolución de este protocolo conocido como ***OAuth 2.0*** en 2012, proporcionando un marco más flexible y simplificado [13]. *OAuth 2.0* se centró en mejorar la facilidad de uso y la adaptabilidad a diferentes tipos de aplicaciones, desde aplicaciones web hasta móviles y de escritorio.

Entre los avances hacia la versión actual de OAuth 2.0 y los beneficios que aporta son:

- **Flexibilidad mejorada:** Soporta diversos flujos de autorización (*grant types*) para adaptarse a diferentes escenarios y tipos de aplicaciones.
- **Simplicidad:** Eliminó la necesidad de firmar cada solicitud, reduciendo la complejidad y facilitando su adopción.
- **Amplia adopción:** Se convirtió en el estándar de facto para autorización en aplicaciones modernas y servicios en línea.

Sin embargo, no aborda directamente la autenticación del usuario, lo que llevó a interpretaciones erróneas y posibles vulnerabilidades si se implementaba incorrectamente, con lo que surgió la necesidad de un estándar que proporcionase un método de autenticación más seguro y simple de implementar.

2.3.2 *OpenID Connect* (OIDC)

Reconociendo que *OAuth 2.0* no proporcionaba un mecanismo estándar para la autenticación de usuarios, la Fundación OpenID desarrolló ***OpenID Connect*** en 2014 [14]. Esta tecnología actúa como una capa de identidad sobre el protocolo *OAuth 2.0*, añadiendo el paso de autenticación al proceso de autorización ya existente. *OAuth 2.0* se centra en la autorización (otorgar acceso limitado a recursos), pero no resuelve la autenticación (verificar la identidad del usuario). *OpenID Connect* consigue esto al añadir una capa de autenticación que, al finalizar el proceso de OAuth 2.0, permite a la aplicación no solo saber que el usuario concedió acceso, sino también quién lo solicitó.

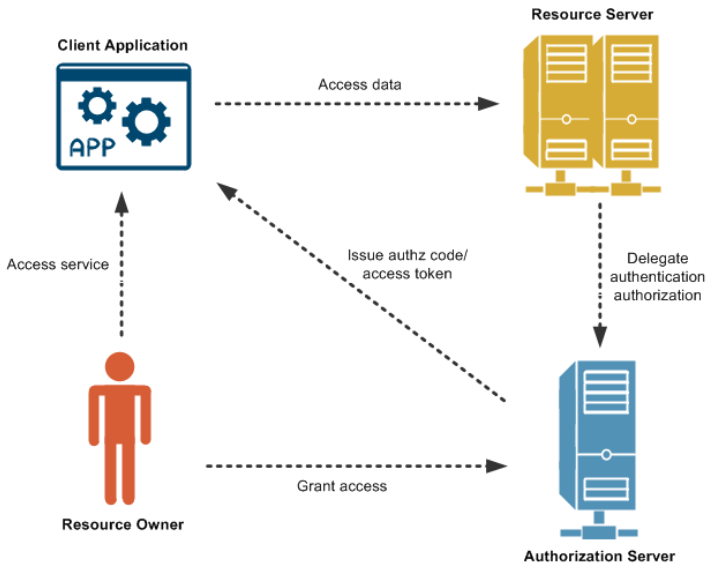

Figura 9. OIDC

Al emplear estándares como *JSON Web Tokens* (JWT), *OpenID Connect* agiliza la implementación de la autenticación en diferentes plataformas y lenguajes, permitiendo compartir de forma segura la información de identidad entre varios sistemas.

En cuanto a las características y beneficios de *OpenID Connect*, ofrece un marco sencillo y seguro para confirmar la identidad del usuario de manera estandarizada, lo que garantiza que el sistema reconozca con fiabilidad quién está iniciando sesión. Además, facilita la interoperabilidad al ser un estándar ampliamente adoptado, con lo cual se simplifica la integración entre diversos proveedores de identidad y aplicaciones o servicios sin necesidad de soluciones personalizadas. Por último, utiliza JWT para transmitir de forma segura la información de identidad del usuario mediante el *ID Token*, lo que simplifica tanto la validación como la integración con otros sistemas.

2.3.3 *JSON Web Tokens* (JWT)

Con la creciente adopción de arquitecturas sin estado y de *APIs REST*, surgió la necesidad de contar con un mecanismo eficiente para transmitir información de autenticación y autorización de forma segura y ligera. En 2015, se estandarizaron los ***JSON Web Tokens*** (JWT) a través de la RFC 7519 [15], convirtiéndose en una solución ampliamente utilizada.

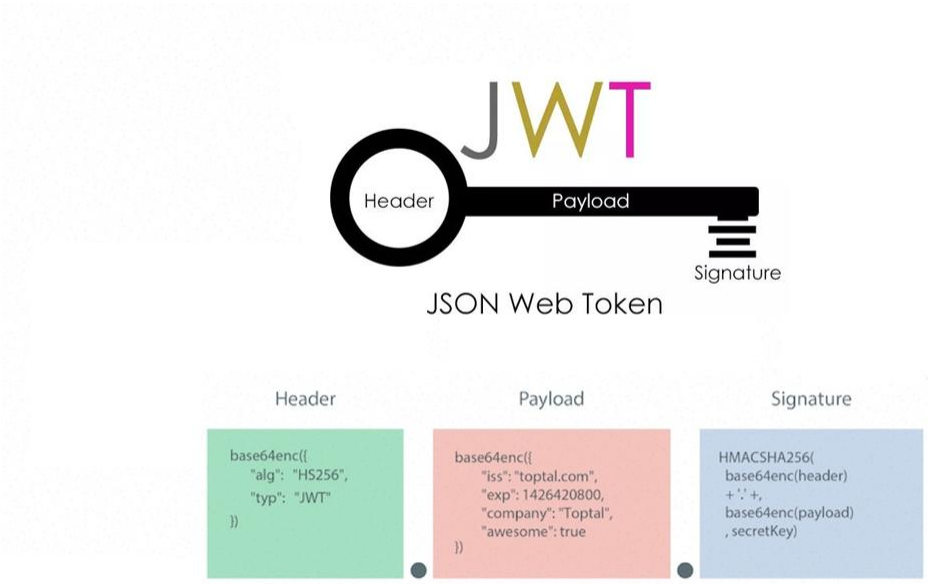

Figura 10. Token JWT

El uso de JWT hizo que se mejorara sobre los siguientes aspectos:

- Autenticación sin estado: Permite a los servidores validar la identidad del usuario sin mantener sesiones en el servidor, reduciendo la complejidad y los recursos necesarios para la gestión de estados.
- Eficiencia y portabilidad: Al ser compacto y autónomo, es ideal para aplicaciones web, móviles y APIs, facilitando la integración y reduciendo el consumo de ancho de banda.

- **Integridad y seguridad**: Los tokens están firmados criptográficamente, lo que permite comprobar su autenticidad y evitar manipulaciones, garantizando así la confiabilidad de los datos transmitidos.

La incorporación de JWT mejoró además las capacidades de ***OAuth 2.0*** y ***OpenID Connect*** al proporcionar un formato estandarizado y seguro para transmitir información. En OpenID Connect, el ***Token ID*** se implementa como un JWT que incluye datos relativos a la autenticación y la identidad del usuario, unificando así el proceso de autorización y autenticación dentro de un mismo flujo.

2.3.4 *Identity as a Service* (IDaaS)

Con el auge de los servicios en la nube y las aplicaciones SaaS a partir de la década de 2010, las organizaciones empezaron a buscar formas más eficientes de gestionar identidades en entornos híbridos y *multicloud*. Como respuesta a estas necesidades surgió ***Identity as a Service*** (IDaaS) [16], un modelo que centraliza la gestión de identidades y accesos a través de servicios totalmente basados en la nube, que ofrece servicios de administración de identidades sin requerir infraestructura local (on-premise), reduciendo así costos y complejidad en el despliegue. Además, es compatible con protocolos de autenticación y autorización como *OAuth 2.0*, *OpenID Connect* y SAML, facilitando la interoperabilidad con múltiples sistemas y proveedores actuales. También tiene una gran escalabilidad y se adapta de forma rápida a los cambios en la demanda, permitiendo aumentar o disminuir recursos según sea necesario.

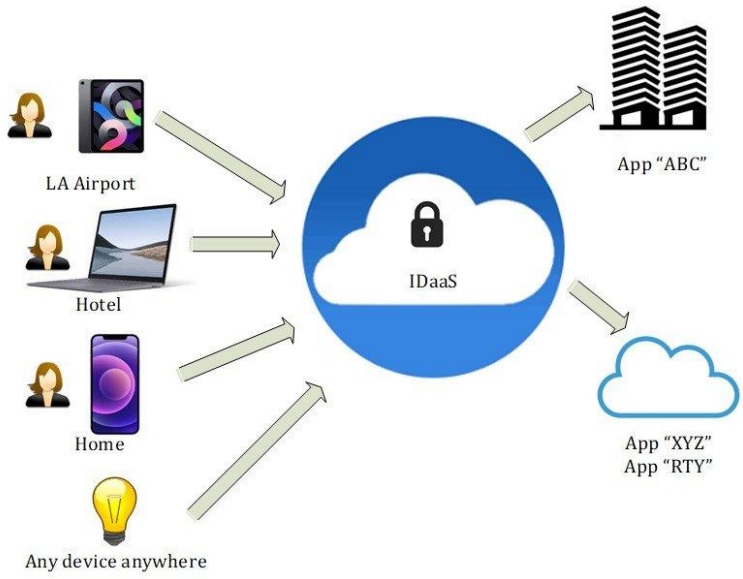

Figura 11. Esquema IDaaS

Este modelo presenta una serie de anomalías:

- **Dependencia del proveedor**: La disponibilidad y la seguridad del servicio dependen en gran medida del proveedor, lo que implica ceder parte del control directo.

- **Cumplimiento y regulación**: Resulta fundamental garantizar que el almacenamiento y procesamiento de datos cumplan con normativas como el GDPR, lo cual puede exigir medidas adicionales de protección de datos.
- **Integración con sistemas heredados**: La vinculación de IDaaS con aplicaciones y sistemas existentes puede requerir un esfuerzo considerable de adaptación y migración.

En definitiva, **IDaaS** representa la evolución natural de la gestión de identidades en un entorno cada vez más marcado por la movilidad, la adopción de la nube y la necesidad de flexibilidad.

2.4 La propuesta de Keycloak y Vault

2.4.1 Keycloak

Keycloak es una solución de gestión de identidades y accesos (IAM) de código abierto desarrollada por Red Hat [17]. Ofrece autenticación y autorización para aplicaciones y servicios modernos, incluyendo inicio de sesión único (SSO) que posibilita autenticar al usuario una sola vez para luego acceder a múltiples aplicaciones sin repetir el proceso de autenticación. Además, es compatible con protocolos estándares como *OAuth 2.0*, *OpenID Connect* y SAML 2.0, puede integrarse con sistemas LDAP o *Active Directory* para sincronizar usuarios y grupos, y admite flujos de autenticación personalizados que añaden pasos como MFA o preguntas de seguridad. También brinda soporte para varios idiomas gracias a su función de internacionalización y permite la administración basada en roles (RBAC), con lo cual se definen permisos y roles de manera detallada.

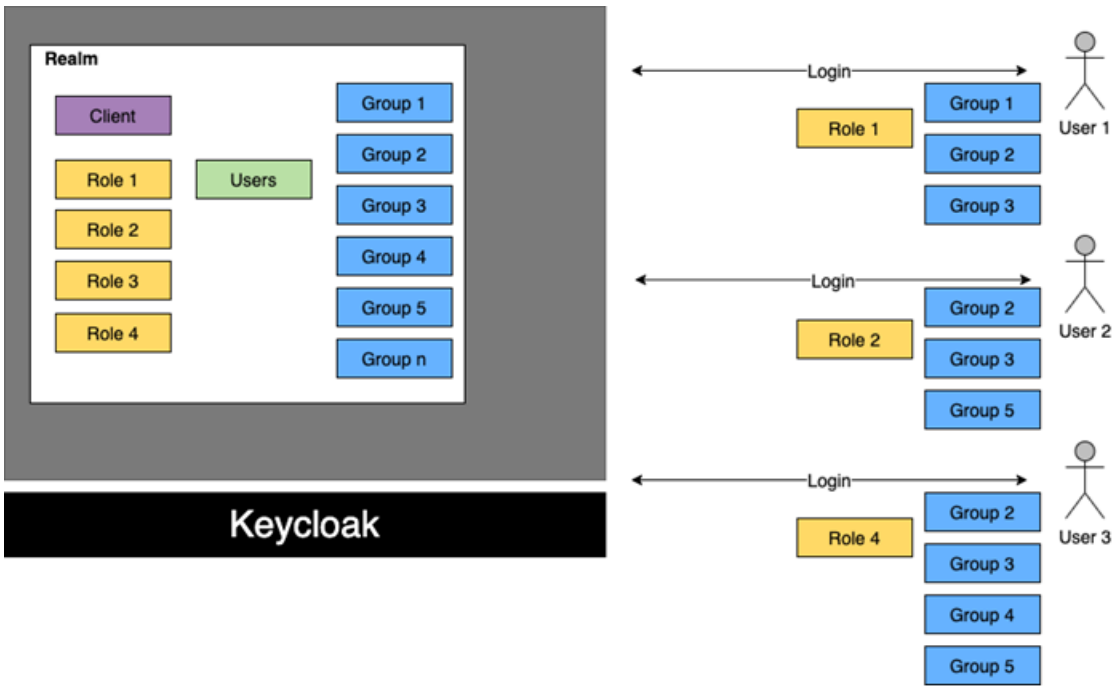

Figura 12. Arquitectura de Keycloak

Keycloak destaca por su extensibilidad a través de SPI (*Service Provider Interfaces*) [18], que posibilita adaptar la solución con autenticaciones o proveedores de usuarios a medida, y por su flexibilidad de despliegue tanto en entornos locales (*on-premise*) como en la nube. Al ser de código abierto, cuenta con el respaldo de una comunidad que contribuye con mejoras y parches, y se encarga de ofrecer soporte. Entre sus principales casos de uso se incluyen la autenticación en aplicaciones web y móviles, la protección de microservicios y *endpoints* de API con tokens de acceso y roles de autorización, así como la federación de identidades mediante la integración de diferentes fuentes de usuarios y proveedores externos de identidad.

2.4.2 Vault

Vault es una herramienta de código abierto creada por HashiCorp [19] para administrar secretos y proteger datos sensibles. Se centra en el control de acceso y el almacenamiento seguro de información como claves API, contraseñas y certificados, utilizando cifrado avanzado para mantenerlos a salvo. Además de definir políticas detalladas que restringen quién puede acceder a cada secreto y en qué condiciones, también permite generar credenciales dinámicas para bases de datos y servicios en la nube con tiempos de vida limitados, lo que fortalece la seguridad y reduce riesgos. Su sistema de auditoría registra todas las operaciones, facilitando la identificación de actividades sospechosas y el cumplimiento de normativas como PCI DSS, HIPAA y GDPR. Vault es compatible con Kubernetes, Docker y sistemas de CI/CD, ofreciendo una integración sencilla en entornos más actuales.

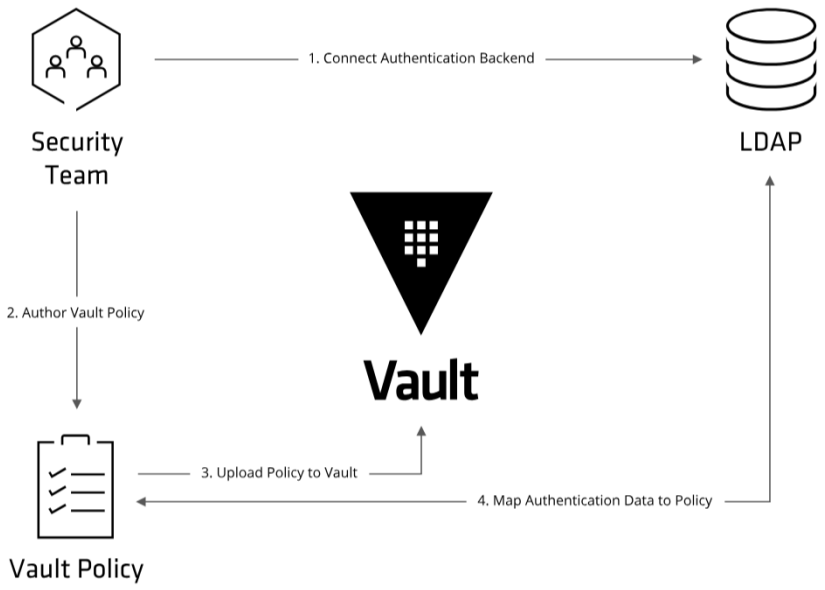

Figura 13. Arquitectura de Vault

En proyectos reales, puede actuar como un almacén centralizado de secretos para arquitecturas de microservicios [20], evitando que estos datos se incluyan en el código o en archivos de configuración. También automatiza la rotación de claves y contraseñas para minimizar el riesgo ante posibles filtraciones, y proporciona APIs de "encriptación como servicio" que permiten cifrar y descifrar

datos sin exponer las claves directamente a las aplicaciones. En definitiva, ofrece seguridad centralizada, facilidad de cumplimiento normativo y la flexibilidad necesaria para adaptarse a diversos entornos y tecnologías, tanto *on-premise* como en la nube.

2.5 Contenedores, orquestación y DevOps

2.5.1 Contenedores

Los **contenedores** son una forma de virtualización a nivel de sistema operativo que aíslan las aplicaciones en entornos independientes [21]. **Docker**, como una de las plataformas de contenedorización más usadas, permite que los contenedores compartan el kernel del sistema anfitrión, lo cual mejora la eficiencia de recursos en comparación con las máquinas virtuales tradicionales que requieren un sistema operativo completo por instancia [22].

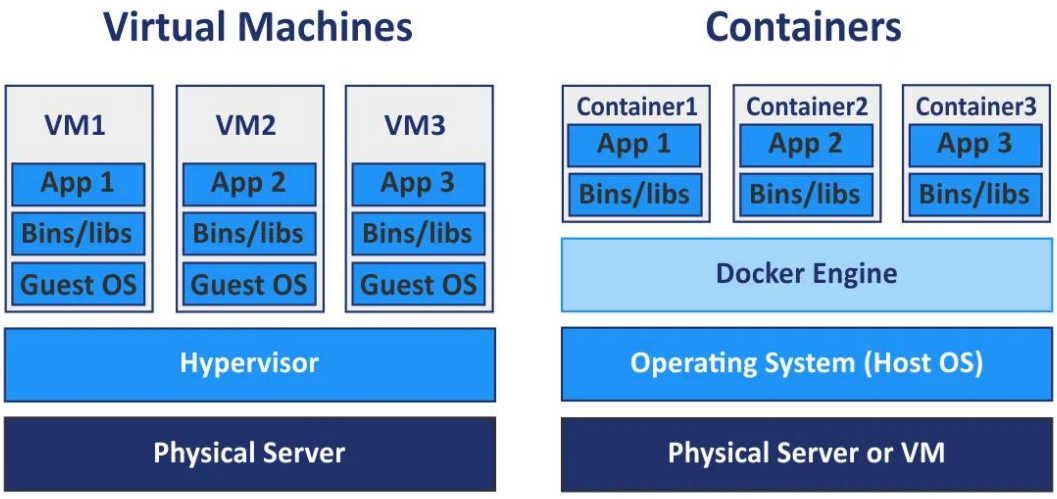

Figura 14. Comparativa máquinas virtuales y contenedores

Esta tecnología también proporciona portabilidad, garantizando que los contenedores se ejecuten de forma consistente en diferentes entornos de desarrollo, pruebas y producción, y agiliza el arranque, lo que facilita el escalado dinámico. Además, fomenta la **modularidad**, ya que las aplicaciones pueden dividirse en componentes más pequeños (microservicios) que pueden desarrollarse y desplegarse de forma independiente.

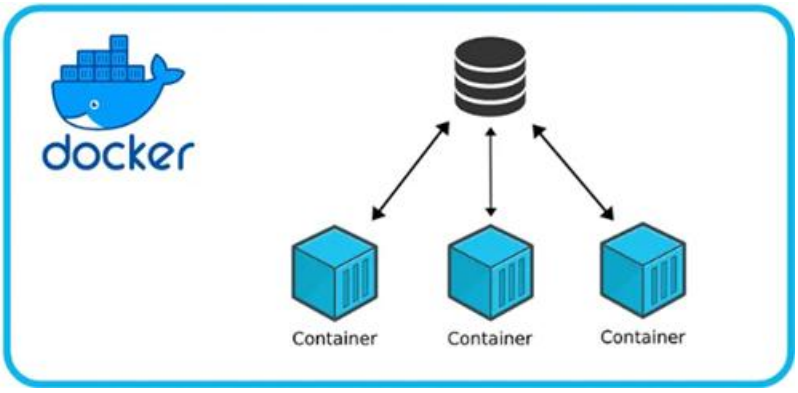

Figura 15. Modularidad Docker

No obstante, los contenedores no ofrecen el mismo nivel de aislamiento que las máquinas virtuales. Si se produce una vulnerabilidad en el kernel, todos los contenedores pueden verse afectados. También existe el riesgo de usar imágenes inseguras, las cuales pueden contener fallos o configuraciones deficientes [23]. Asimismo, la gestión de secretos (por ejemplo, claves o contraseñas) dentro de las imágenes o de variables de entorno plantea riesgos de exposición de datos sensibles.

2.5.2 Orquestación con Kubernetes

Kubernetes es una plataforma de orquestación de contenedores que automatiza el despliegue, escalado y gestión de aplicaciones en contenedores [24]. Se encarga de administrar los recursos del clúster, implementa y escala las aplicaciones según la demanda y reemplaza o reinicia los contenedores que no responden, proporcionando un mecanismo de autorreparación. Además, ofrece descubrimiento de servicios y balanceo de carga para distribuir el tráfico de manera eficiente, y maneja configuraciones y almacenamiento de datos de forma persistente.

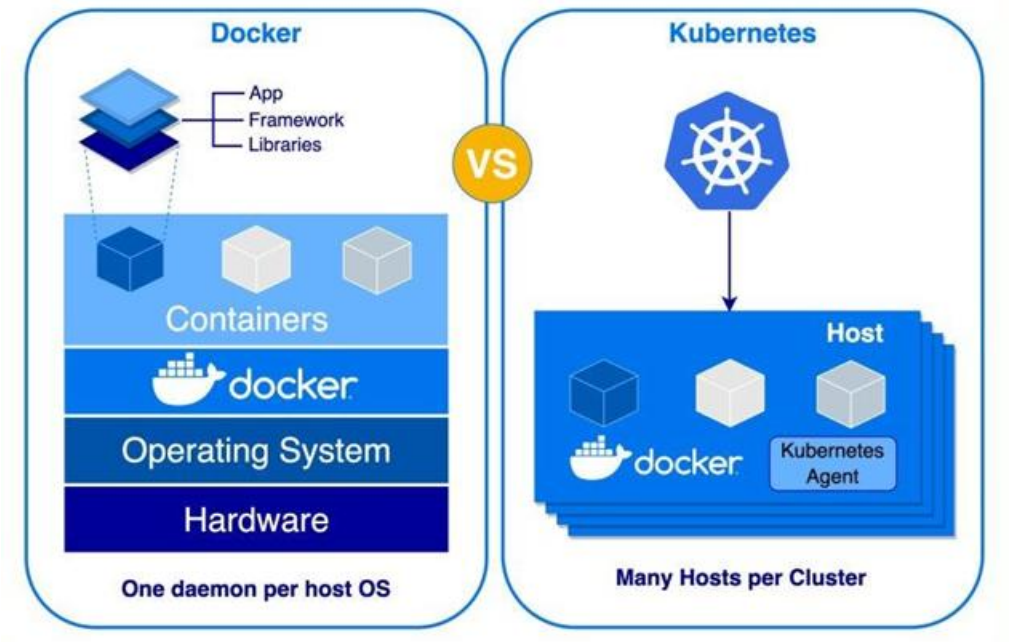

Figura 16. Docker y Kubernetes

Para automatizar despliegues de sistemas seguros, Kubernetes permite definir políticas de seguridad a nivel de contenedor y pod, restringiendo capacidades y accesos [25]. Puede integrarse con herramientas como Vault para gestionar secretos de forma segura, y facilita la aplicación de actualizaciones y parches con interrupciones mínimas. Entre las buenas prácticas de seguridad se encuentran el uso de **Control de Acceso Basado en Roles** (RBAC) para restringir las acciones de usuarios y servicios, el uso de namespaces para aislar recursos y aplicaciones, y la configuración de políticas de red para regular el tráfico entre pods y servicios

2.5.3 DevOps

DevOps es una metodología que integra las prácticas de desarrollo de software (Dev) y operaciones de TI (Ops) para reducir el ciclo de vida de desarrollo y ofrecer entregas continuas de alta calidad [26]. Su base está en la colaboración y la comunicación, haciendo que desarrolladores y operaciones trabajen juntos desde el inicio y que se automatice e integre gran parte del proceso. Dentro de sus pilares se encuentra la **integración continua (CI)**, que consiste en automatizar la inserción de cambios en el código para detectar errores con rapidez, y la **entrega Continua (CD)**, que agiliza y hace confiable el despliegue de aplicaciones en entornos de producción. También promueve la **infraestructura como código (IaC)**, un enfoque que gestiona y aprovisiona la infraestructura mediante definiciones en código en lugar de configuraciones manuales.

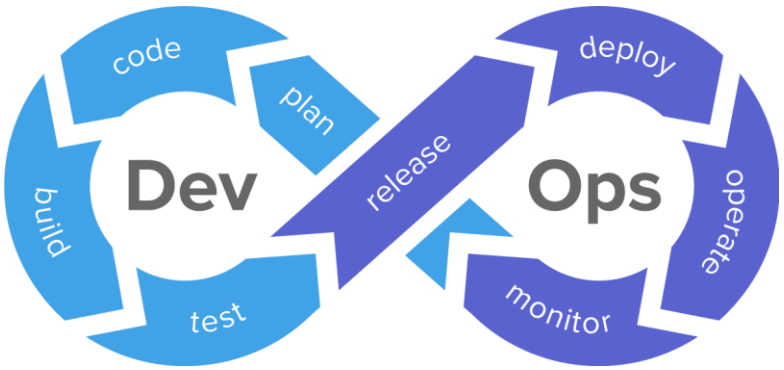

Figura 17. Arquitectura DevOps

Los beneficios de DevOps incluyen una mayor velocidad de entrega de soluciones, una mejora continua en la calidad del software gracias a la automatización de pruebas y la posibilidad de escalar las aplicaciones de forma eficiente al automatizar tareas repetitivas, lo que libera tiempo para actividades de mayor valor. Para implementar DevOps, suelen usarse **sistemas de control de versiones** (Git, SVN), **herramientas de CI/CD** (Jenkins, GitLab CI/CD, CircleCI), **herramientas de gestión de configuración** (Ansible, Puppet, Chef) y **soluciones de monitorización y registro** (Prometheus, Grafana, ELK Stack). Sin embargo, su adopción plantea desafíos como la necesidad de un cambio cultural, la integración de la seguridad en despliegues rápidos y la complejidad a la hora de elegir y configurar las herramientas más adecuadas.

2.5.4 DevSecOps

DevSecOps es la evolución de DevOps que integra las prácticas de seguridad a lo largo de todo el ciclo de vida de desarrollo de software [27]. Reconoce que la seguridad es responsabilidad de todos y no solo del equipo especializado en esta área, por lo que promueve la integración temprana de controles y pruebas de seguridad desde las fases iniciales de desarrollo, así como la automatización de análisis y despliegues. También impulsa la colaboración entre desarrolladores, operaciones y equipos de seguridad, y la capacidad de responder de manera ágil ante vulnerabilidades e incidentes.

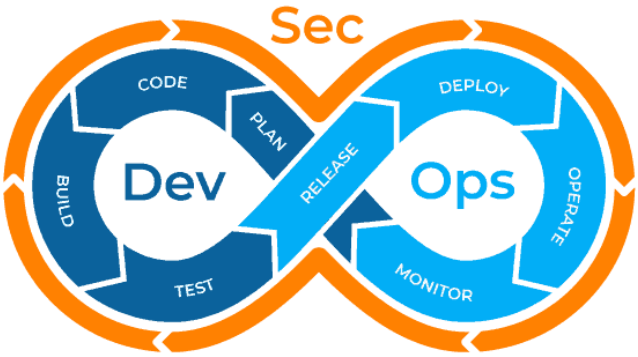

Figura 18. Arquitectura DevSecOps

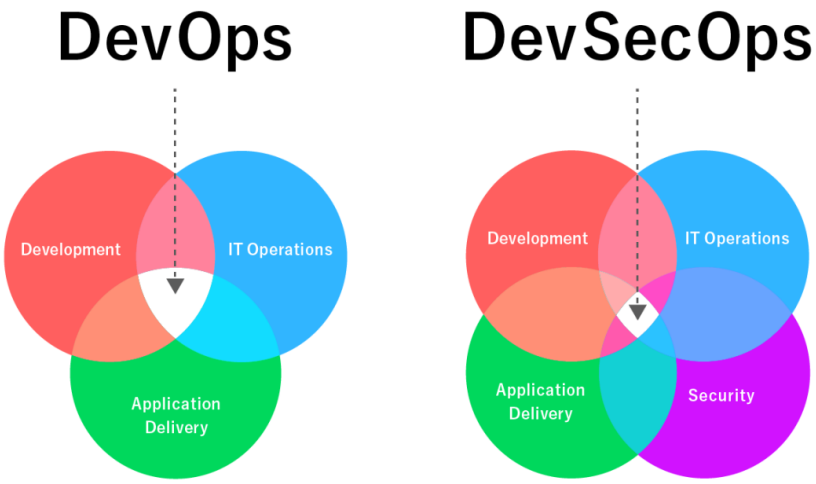

Figura 19. Comparativa DevOps y DevSecOps

Sus principales beneficios incluyen la identificación temprana de riesgos, lo que minimiza el costo y el impacto de las correcciones, y el cumplimiento continuo de normativas gracias a la automatización de auditorías e informes. Además, fortalece la calidad del software al aplicar buenas prácticas de codificación segura y revisiones constantes del código. Entre las herramientas y prácticas más comunes se encuentran el análisis de código estático (SAST) para detectar vulnerabilidades en el código fuente, el análisis de código dinámico (DAST) que evalúa las aplicaciones en tiempo de

ejecución y la gestión de dependencias que examina librerías y componentes de terceros en busca de fallos conocidos [28].

2.6 Modelos de *Cloud Computing*

Los modelos de *Cloud Computing* [29] ofrecen diferentes niveles de abstracción y servicios que permiten a las organizaciones adoptar soluciones flexibles y escalables, ajustándose a sus exigencias tecnológicas y de negocio. Además de brindar la posibilidad de desplegar recursos de forma más rápida y con un menor coste inicial, estos modelos facilitan la administración y actualización continua de la infraestructura. De acuerdo con la definición del Instituto Nacional de Estándares y Tecnología de los Estados Unidos (NIST) [30], los principales modelos de servicio en *Cloud Computing* son Infraestructura como Servicio (IaaS), Plataforma como Servicio (PaaS) y Software como Servicio (SaaS). Cada uno de ellos implica un grado diferente de control y responsabilidad sobre la infraestructura, así como la delegación de tareas de administración y mantenimiento en el proveedor de servicios, lo que ofrece a las empresas la flexibilidad necesaria para centrar sus esfuerzos en la innovación y el desarrollo de soluciones de valor.

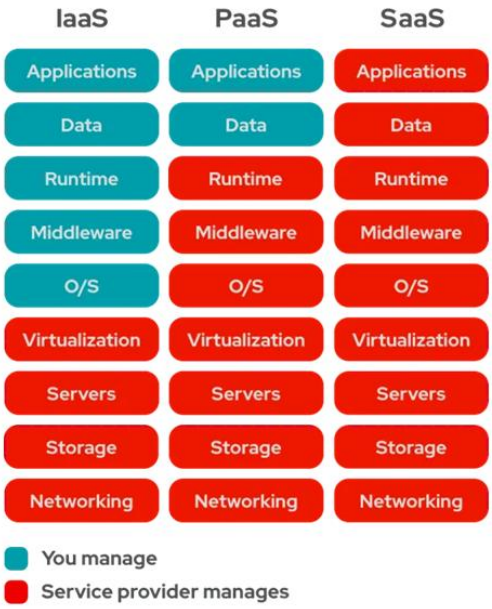

Figura 20. Modelos cloud computing

Estos tres modelos pueden coexistir dentro de una estrategia de nube híbrida o múltiple, donde cada servicio se emplea para tareas específicas de acuerdo con las demandas del negocio. Sin embargo, la adopción de la nube implica contemplar aspectos de seguridad, gobernanza y disponibilidad, así como evaluar las necesidades particulares de cada organización. Una planeación adecuada y la supervisión constante de la infraestructura son esenciales para maximizar los beneficios de los modelos de *cloud computing*.

2.6.1 Infraestructura como Servicio (IaaS)

En este modelo, el proveedor de servicios proporciona recursos de computación virtualizados, tales como máquinas virtuales, capacidad de procesamiento, redes y almacenamiento. El cliente puede instalar sistemas operativos, software y herramientas según sus necesidades, administrando todo lo relacionado con la configuración de la infraestructura y el despliegue de aplicaciones. Algunos ejemplos de herramientas conocidas son Amazon Elastic Compute Cloud (EC2), Microsoft Azure Virtual Machines y Google Compute Engine [31]. IaaS aporta un alto grado de flexibilidad y control, pero requiere mayor responsabilidad en la gestión y seguridad por parte del usuario.

2.6.2 Plataforma como Servicio (PaaS)

Ofrece un entorno completo de desarrollo y despliegue de aplicaciones, proporcionando no solo la infraestructura subyacente, sino también el sistema operativo, el middleware y servicios de base de datos, entre otros componentes. Esto permite a los desarrolladores concentrarse en la creación de aplicaciones sin tener que preocuparse por la gestión de servidores o la configuración de sistemas. Algunos ejemplos pueden ser Google App Engine, Heroku y Microsoft Azure App Service [32]. Dado que el proveedor gestiona la mayor parte de la infraestructura y las actualizaciones, las empresas pueden agilizar los ciclos de desarrollo y reducir costos de mantenimiento.

2.6.3 Software como Servicio (SaaS)

Se trata de la entrega de aplicaciones listas para usar a través de Internet, donde el proveedor se encarga de la infraestructura, la plataforma y la propia aplicación. El usuario simplemente accede al software mediante un navegador o interfaz cliente, pagando por su uso o mediante suscripciones. Soluciones como Salesforce, Microsoft Office 365 y Google Workspace permiten la implementación de este modelo [33]. SaaS elimina la necesidad de instalar, mantener o actualizar el software de forma local, lo que reduce significativamente la complejidad técnica y los costos de soporte en las organizaciones.

2.7 Infraestructura como código (IaC)

La Infraestructura como Código (IaC) consiste en describir y gestionar los recursos de TI (servidores, redes y almacenamiento, entre otros) mediante archivos de configuración legibles por los usuarios, los cuales se almacenan en sistemas de control de versiones como Git, posibilitando un control y seguimiento similar al del desarrollo de software [34]. Gracias a esta práctica, se facilita la revisión por pares, la colaboración entre los distintos equipos de desarrollo y operaciones, y la trazabilidad de los cambios, contribuyendo así a una cultura ágil dentro de las organizaciones.

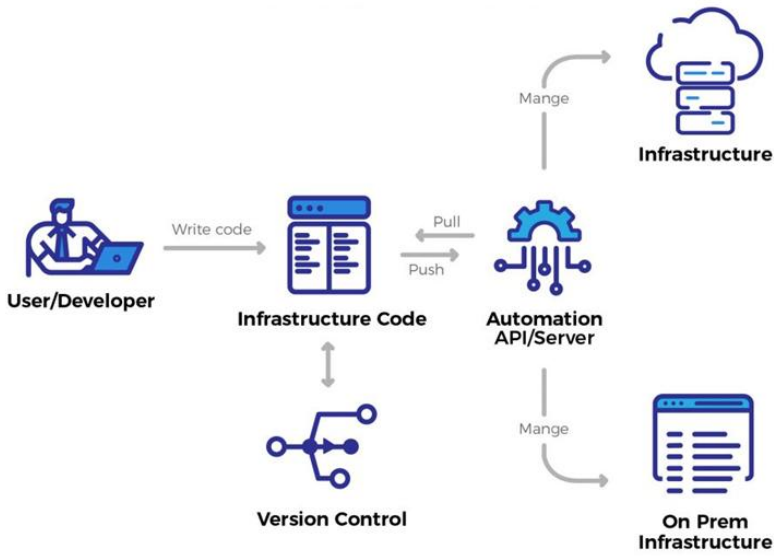

Figura 21. Esquema flujo IaC

Una característica que destacar de la IaC es la inmutabilidad de los entornos. Si se precisa modificar la infraestructura, en vez de realizar cambios directos se recrean los servidores y recursos con la nueva configuración, lo que minimiza el riesgo de inconsistencias o de "deriva de configuración". Asimismo, se promueve la idempotencia, es decir, la capacidad de obtener siempre el mismo resultado al aplicar varias veces la misma configuración, lo que reduce la complejidad y la probabilidad de error [35].

2.7.1 Terraform

Terraform [36] es una herramienta que facilita la definición y administración de recursos en múltiples proveedores de nube (AWS, Azure, Google Cloud, etc.) mediante un enfoque declarativo, donde el usuario especifica la infraestructura y configuración específicas para que el sistema se encargue de crearlo o actualizarlo. Terraform usa su propio lenguaje, el HCL (*HashiCorp Configuration Language*).

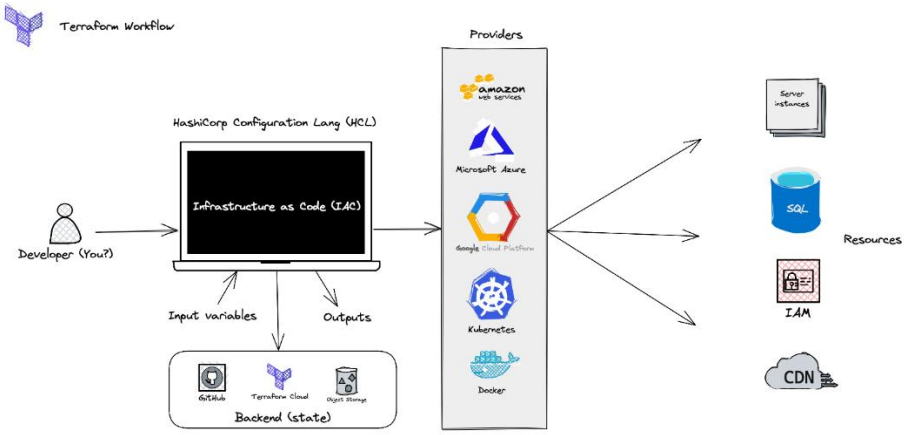

Figura 22. Flujo de trabajo Terraform

2.7.2 Pulumi

Pulumi [37] adopta un método distinto, al posibilitar describir la infraestructura con lenguajes de programación de uso general (como TypeScript, Python o Go), lo que facilita la integración con librerías y herramientas de software convencionales.

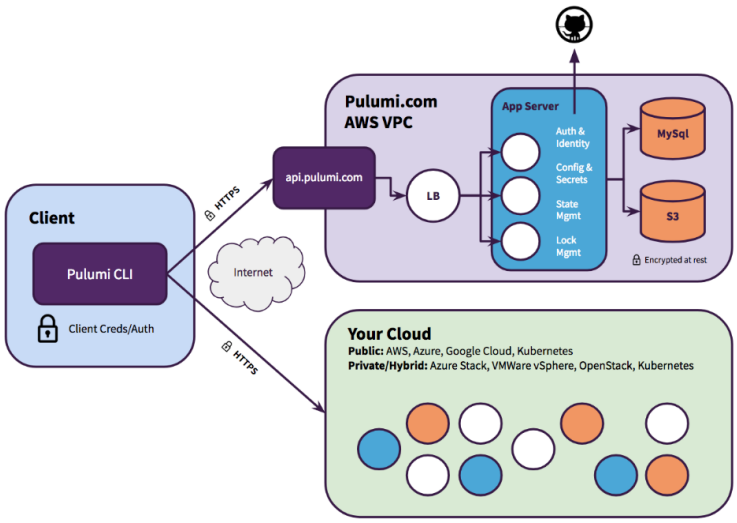

Figura 23. Pulumi

2.7.3 Ansible

Ansible [38] se caracteriza por ser una solución *agentless*, es decir, no requiere la instalación de agentes en las máquinas que va a administrar, sólo requieren tener Python instalado. Emplea *playbooks* escritos en YAML, donde se describen las tareas necesarias para configurar los sistemas y desplegar aplicaciones. Gracias a este enfoque, es posible manejar servidores de manera remota haciendo uso de protocolos de conexión estándar (como SSH), lo cual simplifica la gestión y reduce la complejidad de instalación.

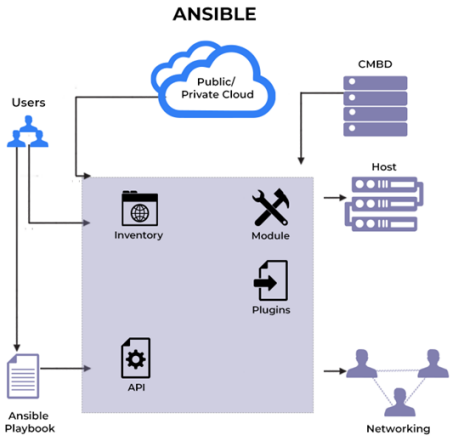

Figura 24. Ansible

Ansible resulta especialmente útil en entornos dinámicos y en infraestructuras de múltiples nubes o entornos híbridos, ya que su curva de aprendizaje es relativamente baja y permite automatizar procesos de forma ágil.

2.7.4 Chef

Chef [39] opera habitualmente en un modelo cliente-servidor, donde los nodos (o máquinas gestionadas) se comunican con un servidor Chef para recibir las configuraciones deseadas. Dichas configuraciones se describen en *cookbooks* y *recipes* escritos en un DSL (lenguaje específico de dominio) basado en Ruby. El enfoque de Chef se centra en definir el estado objetivo de cada sistema, permitiendo realizar cambios de manera automática y consistente.

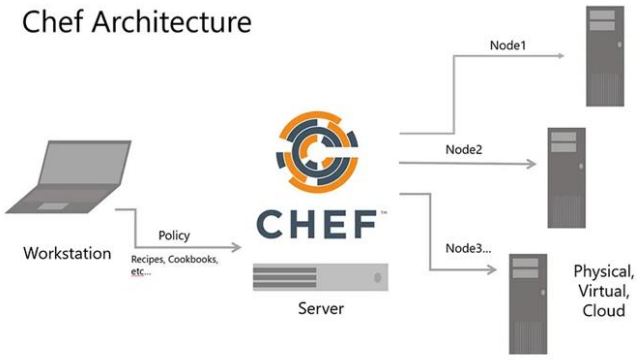

Figura 25. Arquitectura Chef

Aunque requiere un conocimiento más profundo de la infraestructura y del lenguaje Ruby, Chef ofrece gran flexibilidad y un robusto ecosistema de *cookbooks* creados por la comunidad, lo que facilita la integración con múltiples plataformas y servicios.

2.7.5 Puppet

Al igual que Chef, Puppet [40] funciona mayormente con una arquitectura cliente-servidor y se vale de manifiestos para describir el estado deseado de cada nodo. Este lenguaje declarativo propio de Puppet indica cómo debe configurarse el sistema, y el agente de Puppet se encarga de evaluar y aplicar los cambios necesarios para alcanzar ese estado.

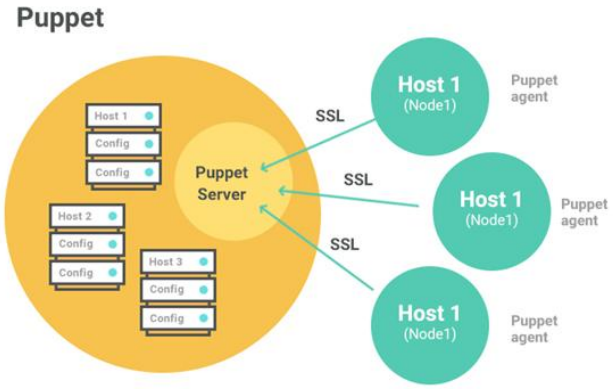

Figura 26. Puppet

Puppet destaca por ofrecer un modelo de configuración idempotente: aplicar el mismo manifiesto varias veces no debería alterar el estado final del sistema. Además, cuenta con una amplia comunidad y un extenso catálogo de módulos predefinidos para diferentes servicios y aplicaciones.

2.8 Comparativa general métodos autenticación y autorización

A continuación de forma generalizada, se presenta una tabla resumen con los principales métodos de autenticación y autorización mencionados en este segundo capítulo, describiendo sus características, ventajas, desventajas y usos más habituales.

Método	Origen / Año	Tipo	Mecanismo / Factores	Ventajas principales	Desventajas principales	Uso típico
PAP (Password Access Protocol)	MIT, CTSS (década de 1960)	Autenticación	Contraseña (algo que el usuario sabe)	– Muy sencillo de implementar – Compatibilidad universal	– Vulnerable a ataques de fuerza bruta y phishing – Reutilización de contraseñas – Gestión segura del almacenamiento crítica	Sistemas heredados, acceso básico a terminales
LDAP	ISO X.500 / DAP simplificado (1990s)	Autenticación y autorización	Contraseña + atributos en directorio centralizado	– Gestión centralizada de identidades – Integración con múltiples sistemas – Buena escalabilidad	– Sigue dependiendo de contraseñas – Requiere conocimientos específicos para su implementación y mantenimiento	Gestión de usuarios y permisos en entornos corporativos
MFA (Autenticación multifactor)	Finales de los 1990 / principios de 2000	Autenticación	Varios factores independientes (conocimiento, posesión, inherencia)	– Aumenta drásticamente la seguridad – Dificulta el acceso con credenciales comprometidas	– Coste adicional (tokens, aplicaciones móviles) – Mayor complejidad para el usuario	Acceso a servicios financieros, entornos críticos
SAML / SSO	OASIS (2002)	Autenticación y autorización	Aserciones XML intercambiadas entre proveedor de identidad (IdP) y proveedor de servicio (SP)	– Interoperabilidad entre dominios – Menos inicios de sesión repetidos, mejor experiencia de usuario	– Implementación compleja – Menor adecuación a arquitecturas de microservicios y móviles	Federaciones de identidad en empresas, aplicaciones web
OAuth 1.0 / OAuth 2.0	OAuth 1.0 (2007) / OAuth 2.0 (2012)	Autorización	Tokens de acceso delegados (flujos de concesión)	– Delegación de permisos sin compartir credenciales – Adaptable a distintos tipos de aplicación	– No define un mecanismo de autenticación de usuario – Peligro si se configura incorrectamente	Integración de aplicaciones con redes sociales y API
OpenID Connect (OIDC)	Fundación OpenID (2014)	Autenticación y autorización	Capa de identidad sobre OAuth 2.0 (ID Token en JWT)	– Añade autenticación estándar a OAuth 2.0 – Uso de JWT para transmitir identidad de forma segura – Amplia interoperabilidad	– Ligeramente más complejo que OAuth 2.0 puro – Gestión de tokens y validación de JWT	Inicio de sesión único en aplicaciones web y móviles
JSON Web Tokens (JWT)	RFC 7519 (2015)	Autenticación y autorización	Token compacto firmado (JSON) y sin estado	– Arquitectura sin estado en el servidor – Ligero y portable – Integridad garantizada criptográficamente	– Dificultad para revocar tokens – Exposición de claves si no se protegen adecuadamente	Microservicios, API REST, flujos de OIDC/OAuth
IDaaS (Identity as a Service)	SaaS en la nube (2010s)	Autenticación y autorización	Servicio en la nube compatible con OAuth, OIDC y SAML	– Gestión de identidades centralizada en la nube – Alta escalabilidad y flexibilidad – No requiere infraestructura local	– Dependencia del proveedor del servicio – Retos de cumplimiento normativo (GDPR) – Integración con sistemas heredados	Gestión de usuarios en entornos multicloud y SaaS

Figura 27. Tabla comparativa métodos de autenticación

Actualmente no existe un único método "ideal" de autenticación y autorización general para todos los casos. Cada tecnología aporta ventajas específicas (desde la simplicidad de PAP hasta la flexibilidad de IDaaS). La elección adecuada dependerá de los requisitos de seguridad, escalabilidad y experiencia de usuario, teniendo en cuenta el tipo de aplicación y los datos sensibles u operaciones que manejen. Cada vez se recomienda más y se opta por combinar varios enfoques (por ejemplo, OIDC con MFA y JWT) para obtener un equilibrio adecuado entre usabilidad y robustez.

2.9 Resumen del capítulo

En este capítulo se estudia cómo han evolucionado las diferentes estrategias de seguridad en aplicaciones y qué tecnologías han ido implementándose con el paso del tiempo, para reforzar las vulnerabilidades que han ido surgiendo. En primer lugar, se repasan los métodos tradicionales de autenticación y autorización, como los sistemas basados en contraseñas, LDAP y SAML, junto a la autenticación multifactor (MFA). A pesar de su solidez en cuanto a la gestión de identidades, presentan limitaciones importantes en términos de escalabilidad, usabilidad y seguridad.

Posteriormente aparecieron tecnologías como *OAuth 2.0*, *OpenID Connect* (OIDC) y *JSON Web Tokens* (JWT), las cuales permiten una gestión más ágil de cara al usuario y al administrador del sistema, ya que se unifican los procesos de autenticación y autorización. Como resultado de ello las credenciales de los usuarios de las distintas aplicaciones queden menos expuestas y por lo tanto menos vulnerables, reforzándose así su seguridad. También aparecen soluciones que adoptan una nueva arquitectura, una de ellas es conocida como IDaaS (*Identity as a Service*), donde gestión de identidades queda externalizada del entorno aplicativo, aunque esto deja posibles riesgos en manos de terceros de cara a la normativa vigente relacionada con la seguridad de la información y cumplimiento normativo.

Herramientas actuales y en las que se basa el actual trabajo son Keycloak, diseñada y utilizada para la administración centralizada de identidades y accesos, y Vault, que se centra en proteger y gestionar secretos así como otros datos sensibles. Ambas herramientas ayudan a cualquier aplicación ya sea que disponga una arquitectura monolítica o basadas en microservicios y contenedores, incluyéndose como una capa externa de seguridad.

Finalmente, se introducen metodologías de desarrollo como son DevOps y DevSecOps, que integran la seguridad a lo largo de todo el ciclo de vida del software, además de emplear entornos basados en contenedores orquestados con Kubernetes. En conjunto con la Infraestructura como Código (IaC) haciendo uso de herramientas como Terraform, Ansible, Chef o Puppet permiten desplegar sistemas escalables y seguros, optimizando el rendimiento y garantizando el cumplimiento de estándares y normativas de protección de datos.

Capítulo 03: Herramientas y tecnologías

En este capítulo se describen las herramientas y tecnologías empleadas en el proyecto, ofreciendo una visión detallada de cada componente y de cómo se integran dentro del sistema global. Entre las tecnologías clave se encuentran **Docker**, utilizado para la contenedorización, **Keycloak**, para la gestión de identidad y acceso, **Vault**, para la gestión de secretos, y **Terraform**, que permite el despliegue automatizado de la infraestructura.

3.1 Virtualización con Docker

Docker es una plataforma que permite empaquetar aplicaciones y sus dependencias en contenedores ligeros y portátiles, lo que agiliza su despliegue y escalabilidad. Al aislar las aplicaciones en contenedores, se obtiene un entorno homogéneo y coherente que se mantiene tanto en la fase de desarrollo como en la de producción, evitando diferencias debidas a configuraciones del sistema o a variaciones en librerías. Las principales características de Docker son las siguientes:

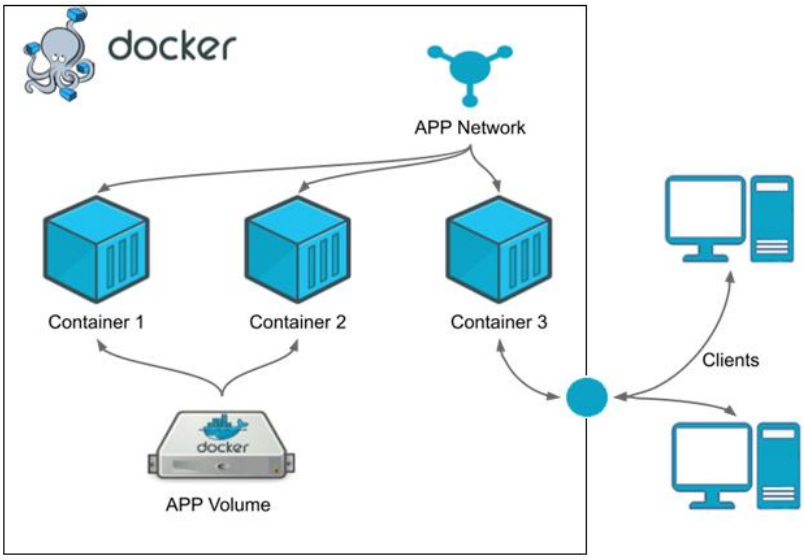

Figura 28. Virtualización con Docker

- **Portabilidad:** Los contenedores pueden ejecutarse en cualquier sistema que tenga Docker instalado, sin importar las diferencias en la configuración o en el sistema operativo subyacente.
- **Eficiencia de recursos:** Dado que comparten el kernel del sistema anfitrión, los contenedores son más ligeros que las máquinas virtuales, reduciendo el consumo de recursos de hardware.
- **Aislamiento:** Cada contenedor posee su propio sistema de archivos, procesos y red, lo que evita interferencias entre aplicaciones.
- **Rapidez de despliegue:** Los contenedores pueden iniciarse en segundos, lo que acelera significativamente los procesos de desarrollo, prueba y puesta en producción.

La creación de imágenes en Docker se fundamenta en la idea de contar con plantillas reutilizables que incluyan tanto la aplicación como todas las dependencias necesarias para su ejecución. El proceso

suele comenzar con una imagen base, que puede ser oficial o de confianza, y sobre la cual se añaden las capas que conforman el entorno requerido por la aplicación

Para llevar a cabo esta construcción se emplea el Dockerfile, un archivo de texto que especifica, mediante una serie de instrucciones, cómo configurar y preparar ese entorno dentro del contenedor. Por ejemplo, se puede establecer la versión del lenguaje a utilizar, instalar dependencias de sistema, copiar el código fuente y definir el comando de inicio.

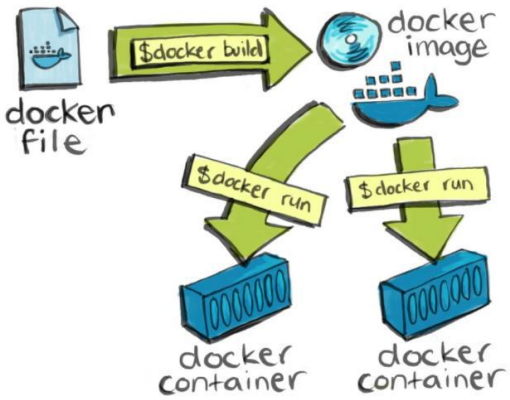

Figura 29. Comandos Docker

El comando "*docker build*" es el encargado de traducir esas instrucciones a capas de imagen, generando una plantilla final que se puede distribuir y desplegar en cualquier sistema que tenga Docker instalado. Cada capa se cachea para optimizar las construcciones posteriores y agilizar los procesos de integración continua. De esta forma, se logra una estandarización del entorno de ejecución de la aplicación, lo que facilita la portabilidad y reduce los errores originados por diferencias en la configuración del servidor.

En Docker, las redes virtuales permiten comunicar contenedores entre sí y con el exterior, garantizando un mayor control sobre la topología de red y el tráfico de datos. La configuración de red por defecto se basa en la creación de una red bridge, donde cada contenedor recibe una dirección IP interna y puede comunicarse con otros contenedores que pertenezcan a la misma red. Sin embargo, Docker ofrece otras opciones de configuración, como la red host, donde el contenedor comparte la pila de red del sistema anfitrión, o las redes *overlay*, que facilitan la comunicación entre contenedores en distintos servidores, lo cual resulta esencial en entornos distribuidos u orquestados, como Docker Swarm o Kubernetes.

Para la gestión de redes, Docker proporciona comandos que permiten crear y administrar redes personalizadas, asignar contenedores a diferentes redes y definir políticas de acceso. De esta manera, cada servicio puede colocarse en la red que mejor se ajuste a sus requisitos de seguridad y conectividad, controlando la exposición de los puertos al exterior y limitando el alcance de cada contenedor.

El uso de contenedores implica que, por defecto, todo lo que se almacena dentro de un contenedor se pierde cuando este se elimina o se detiene. Para solventar esta limitación, Docker implementa el concepto de volúmenes, que permiten que los datos existan más allá del ciclo de vida

del contenedor y sean accesibles para otros contenedores o servicios. Existen dos enfoques principales: por un lado, están los volúmenes administrados por Docker, donde la plataforma gestiona la ubicación y el acceso a los datos. Por otro lado, se encuentran los *bind mounts*, que vinculan un directorio o archivo del sistema anfitrión con el contenedor, permitiendo compartir y editar ficheros de manera sencilla.

Este mecanismo de persistencia es crucial para bases de datos y servicios que requieren conservar información entre reinicios, actualizaciones o despliegues. Además, facilita el respaldo y la migración de los datos, dado que los volúmenes pueden copiarse o trasladarse de forma independiente al ciclo de vida de los contenedores. Asimismo, diversos contenedores pueden acceder al mismo volumen, posibilitando la colaboración entre servicios, la compartición de archivos y la centralización de ciertos recursos. De esta forma, Docker no solo simplifica la ejecución de aplicaciones, sino también el almacenamiento de la información que dichas aplicaciones necesitan.

3.2 Docker Compose

Docker Compose es una herramienta que permite definir y gestionar entornos formados por múltiples contenedores de Docker a través de un único archivo de configuración en formato YAML con el nombre de **docker-compose.yml**. En este archivo se especifican los servicios que componen la aplicación, así como sus respectivas configuraciones: imagen a utilizar, variables de entorno, volúmenes (para persistencia de datos o configuración), puertos expuestos, redes y dependencias entre contenedores.

Para el arranque en primera instancia o inicial de algunos servicios es necesario hacer apoyo sobre contenedores auxiliares conocidos como ***init-containers***. Su propósito es realizar tareas de preparación necesarias para que el servicio principal pueda iniciar y funcionar correctamente. A diferencia de los contenedores principales, los *init-containers* se ejecutan de forma **temporal y secuencial**: cada uno debe completarse correctamente para que comience el siguiente, y solo cuando todos han terminado se inicia el contenedor principal. Entre sus usos más comunes se encuentran:

- La inicialización de configuraciones sensibles.
- La creación o migración de bases de datos.
- La validación de dependencias externas.
- La generación de claves, certificados u otros secretos.

En entornos donde se requiere una preparación previa al arranque, como ocurre con Vault, los *init-containers* ofrecen una forma controlada de automatizar este proceso.

Cada servicio definido en el archivo funciona como una instancia aislada de un contenedor, pero interconectada con los demás a través de redes internas gestionadas por Docker. Esta comunicación se realiza utilizando nombres de servicio como identificadores de red (por ejemplo, postgres o vault), lo que evita la necesidad de gestionar direcciones IP de forma manual.

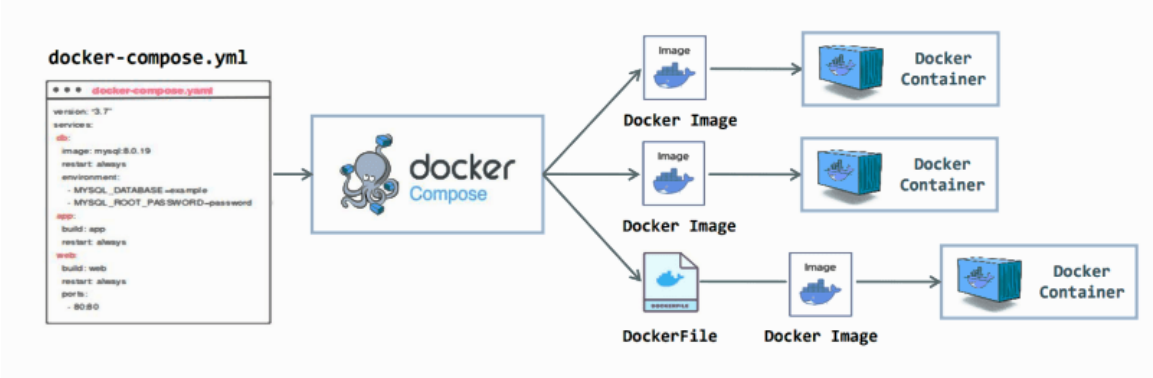

Figura 30. Docker Compose

En el caso concreto de este Trabajo Fin de Máster, Docker Compose se ha utilizado para levantar de forma coordinada los siguientes servicios:

- Keycloak, como solución de gestión de identidad y autenticación, configurado para usar una base de datos externa.
- PostgreSQL, que actúa como base de datos para Keycloak, configurada con un volumen persistente para mantener los datos tras reinicios.
- Vault, como gestor de secretos y credenciales, cuya configuración requiere una fase previa de inicialización.
- Un init-container de inicialización de Vault, diseñado para ejecutarse una vez y automatizar los pasos necesarios de configuración inicial como son la generación de claves de desbloqueo y tokens de acceso.

Docker Compose permite también establecer relaciones de dependencia mediante la instrucción "*depends_on*", lo que define un orden de arranque entre contenedores. No obstante, este mecanismo no garantiza que el servicio esté completamente operativo, por lo que en entornos más exigentes se pueden emplear *healthchecks* y scripts de espera activa para asegurar que un servicio no continúe hasta que sus dependencias estén listas.

Otro aspecto técnico relevante es la definición de volúmenes para persistencia de datos, y redes personalizadas para garantizar una comunicación interna segura y controlada entre los servicios. Gracias a estas configuraciones, es posible crear un entorno de trabajo replicable, coherente y estable, tanto en entornos locales como en despliegues automatizados.

Además, Docker Compose permite realizar tareas de mantenimiento de forma sencilla, como reiniciar servicios concretos, escalar instancias de un servicio o regenerar todo el entorno desde manteniendo los datos críticos.

En resumen, Docker Compose ofrece una solución eficiente, modular y altamente reproducible para orquestar entornos de desarrollo y pruebas complejos. Su uso en este proyecto ha permitido levantar una infraestructura compuesta por servicios interdependientes (Keycloak, Vault y PostgreSQL) de forma automatizada, garantizando su correcta inicialización, comunicación y persistencia de datos con el mínimo esfuerzo manual.

3.3 Keycloak

Keycloak es una solución de código abierto para la gestión de identidad y acceso (IAM) que ofrece autenticación y autorización centralizadas a aplicaciones y servicios modernos. Su objetivo principal es unificar la gestión de usuarios, roles y permisos, de modo que distintas aplicaciones o entornos puedan integrar sus procesos de autenticación sin necesidad de manejar sus propias credenciales de forma aislada. Para lograrlo, se hace uso de conceptos como *realms*, que funcionan como espacios de seguridad independientes donde se configuran usuarios y credenciales, así como roles y mapeos de usuarios, los cuales permiten asignar privilegios o permisos de manera precisa.

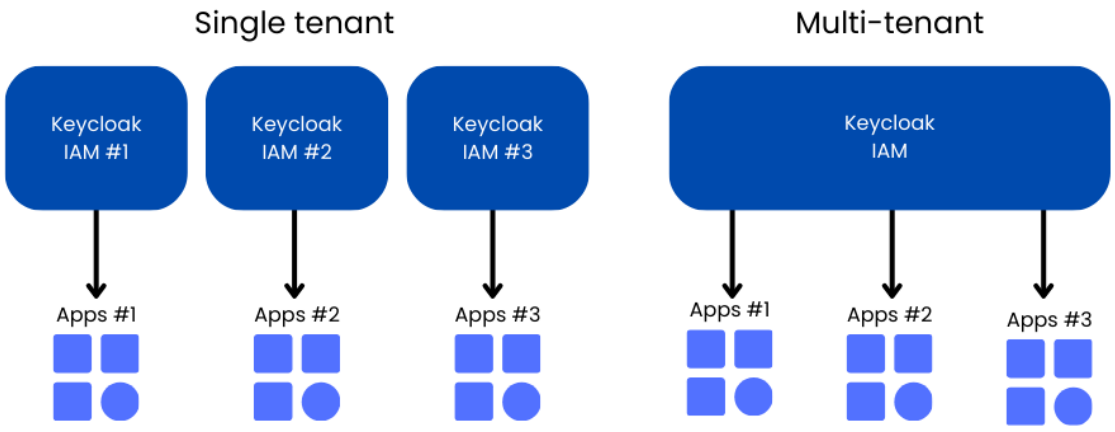

Figura 31. Arquitectura de aplicaciones con Keycloak

Al mismo tiempo, se puede integrar con fuentes externas como LDAP o bases de datos, centralizando así la autenticación y la gestión de atributos de cada usuario.

La instalación y configuración inicial de Keycloak puede llevarse a cabo de distintas formas. Es posible instalarlo manualmente en un servidor de aplicaciones Java o, de manera más sencilla y portable, desplegarlo dentro de un contenedor Docker. Al iniciarse por primera vez, se crea un usuario administrador encargado de configurar el sistema y de definir las aplicaciones (clientes) que necesitarán autenticación, ya sea mediante *OAuth2*, *OpenID Connect* u otros protocolos compatibles. Cada cliente se registra con información como el tipo de aplicación o las direcciones de redirección, y puede utilizar distintos flujos de autenticación según las necesidades (aplicaciones web, móviles o de página única).

Para asegurar una integración fluida con las aplicaciones, Keycloak proporciona adaptadores y librerías que simplifican la implementación de autenticación y autorización. Esto facilita que los desarrolladores se centren en la lógica de negocio en lugar de tener que desarrollar soluciones de seguridad desde cero. Asimismo, el servidor de Keycloak, una vez en marcha, puede administrar todos los inicios de sesión y cierres de sesión, mantener la sesión del usuario y asignar o revocar roles y permisos de forma centralizada.

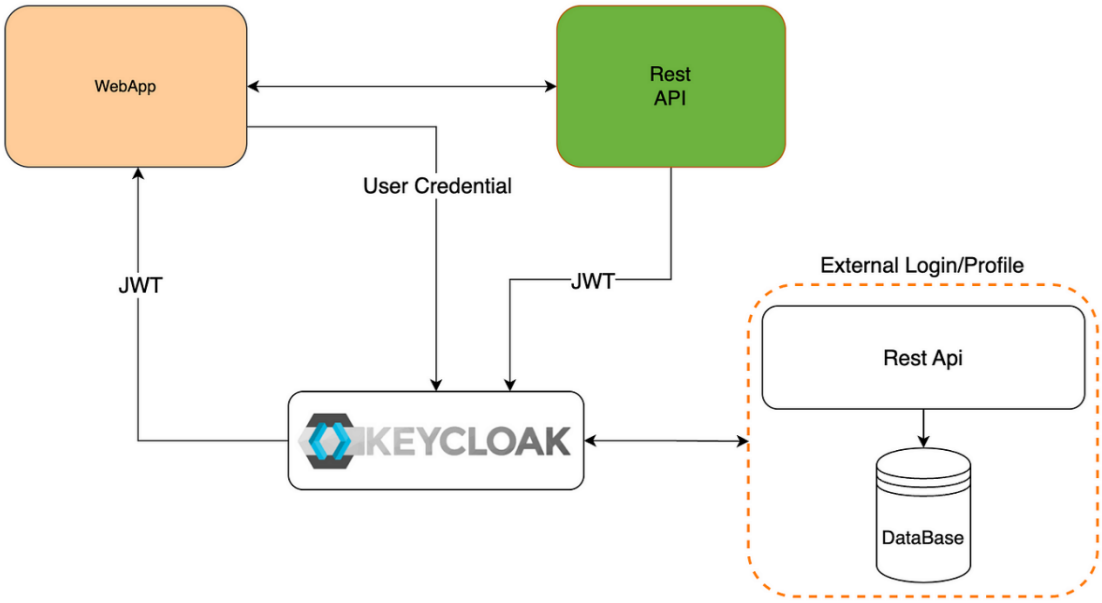

Figura 32. Keycloak como servicio

Al ejecutar Keycloak como un servicio Docker, se obtienen beneficios adicionales en términos de portabilidad, facilidad de despliegue y escalabilidad. El proceso de instalación se reduce a arrancar un contenedor con la imagen oficial de Keycloak y configurar ciertas variables de entorno para especificar aspectos como el usuario administrador, la contraseña y los puertos de red. Esto agiliza enormemente la puesta en marcha, ya que no es necesario instalar ni mantener un servidor de aplicaciones independiente. Además, se puede replicar o escalar el servicio con solo lanzar instancias adicionales del contenedor, ajustándose así a las fluctuaciones de carga y garantizando alta disponibilidad. El aislamiento proporcionado por Docker también minimiza conflictos con otras aplicaciones que se ejecuten en el mismo servidor, dado que cada contenedor opera en su propio entorno. Si se necesita integrar Keycloak en un flujo de orquestación más complejo, puede funcionar sin problemas con Kubernetes u otras herramientas de administración de contenedores, conservando la misma experiencia de instalación y uso en cualquier plataforma compatible con Docker.

3.4 Vault

Vault es una herramienta de código abierto enfocada en la gestión de secretos, diseñada para controlar de manera segura y centralizada el acceso a tokens, contraseñas, certificados y claves de cifrado. Su funcionamiento gira en torno a varios conceptos clave.

En primer lugar, maneja *secrets*, que pueden ser valores estáticos (almacenados de forma cifrada para que solo entidades autorizadas tengan acceso) o secretos dinámicos, generados temporalmente para servicios y aplicaciones con el fin de limitar el tiempo durante el cual las credenciales resultan válidas. Otro pilar fundamental son las políticas de acceso, que establecen qué operaciones pueden llevar a cabo usuarios y aplicaciones, permitiendo un control muy granular sobre los permisos. Además, Vault se integra con múltiples métodos de autenticación (como tokens, AppRole

o LDAP), de modo que, cuando las entidades inician sesión, se les asignan las políticas que determinan qué acciones pueden o no realizar.

Para almacenar y proteger la información de manera persistente, Vault utiliza diferentes *backends*, como Consul, etcd o archivos locales con su configuración. Una vez configurado el backend, la herramienta debe iniciarse mediante un proceso de inicialización, en el que se generan las claves maestras y un token raíz. Vault permanece "sellado" hasta que dichas claves se proporcionan nuevamente, un mecanismo que blinda el acceso a los datos cifrados y evita la filtración de secretos. Un aspecto especialmente destacado es la posibilidad de gestionar credenciales dinámicas, ya que las aplicaciones pueden solicitar credenciales temporales solo cuando las necesitan, reduciendo así la ventana de exposición a brechas de seguridad.

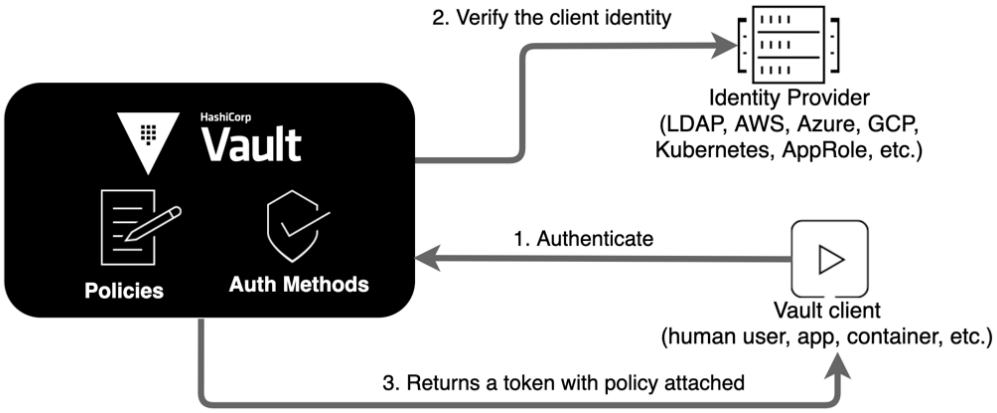

Figura 33. Vault como servicio

La ejecución de Vault como un servicio Docker ofrece numerosas ventajas. El despliegue se simplifica, puesto que basta con lanzar el contenedor con la imagen oficial de Vault y configurar las variables necesarias para indicar la configuración del entorno *backend* de almacenamiento, la ruta de sellado y otros ajustes. Esto acorta drásticamente el tiempo de instalación y reduce los pasos manuales, aprovechando la portabilidad inherente a los contenedores. Además, el aislamiento que proporciona Docker minimiza los posibles conflictos con otras aplicaciones, ya que cada contenedor funciona de manera independiente. De esta forma, se combina la robustez y versatilidad de Vault con la facilidad de despliegue y la escalabilidad que ofrece la tecnología de contenedores, resultando en una solución más ágil y segura para la gestión de secretos en diferentes entornos y escenarios de producción.

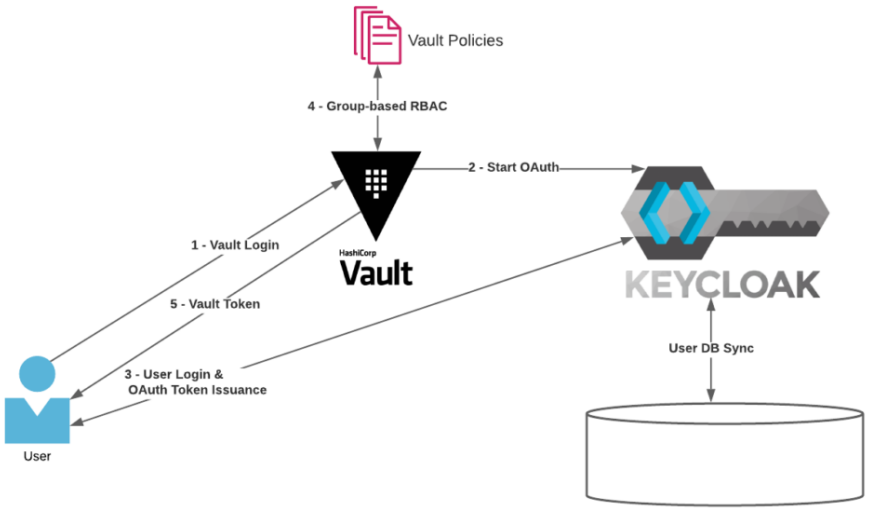

Figura 34. Funcionamiento conjunto Keycloak y Vault

3.5 Terraform

Terraform es una herramienta de infraestructura como código que permite definir, aprovisionar y gestionar la infraestructura de forma eficiente y reproducible mediante archivos de configuración declarativos. Con este enfoque, se especifica el estado deseado de la infraestructura y Terraform se encarga de aplicar los cambios necesarios para alcanzarlo, lo que brinda claridad y reduce errores al automatizar tareas que antes se realizaban de manera manual o con múltiples herramientas. Además, su soporte para diferentes proveedores y servicios lo convierte en una opción versátil para entornos locales y en la nube.

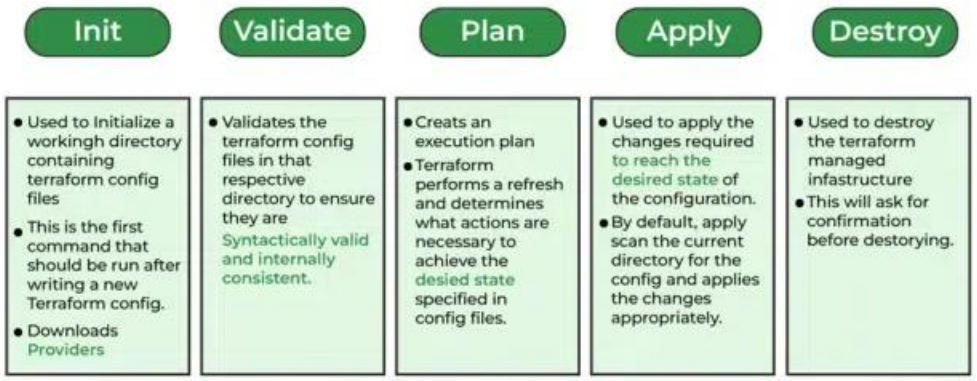

Figura 35. Comandos Terraform

Aunque Terraform es frecuentemente asociado con la gestión de proveedores como AWS, Azure o GCP, también resulta útil para manejar recursos locales. Por ejemplo, puede utilizarse para orquestar contenedores Docker, redes y volúmenes requeridos para servicios como Keycloak y Vault

en un entorno local. Esta capacidad de crear, actualizar y destruir recursos de forma controlada facilita el mantenimiento y la evolución de los sistemas, asegurando coherencia y repetibilidad en cada despliegue. Otra funcionalidad destacable es la etapa de planificación previa, donde es posible visualizar los cambios antes de aplicarlos, lo que minimiza el riesgo de errores y otorga mayor confianza a los equipos de desarrollo y operaciones.

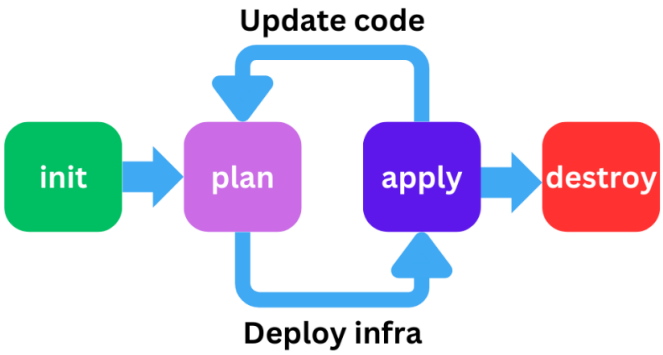

Figura 36. Flujo de comandos Terraform

La principal ventaja de utilizar Terraform es su consistencia y reproducibilidad, ya que al definir la infraestructura como código se garantiza que cada despliegue sea idéntico, independientemente del entorno donde se realice. Sumado a esto, la herramienta dispone de un ecosistema muy amplio de proveedores y de una comunidad activa que comparte módulos reutilizables, acelerando la adopción de buenas prácticas. Esta integración fluida con sistemas de control de versiones como Git y con pipelines de CI/CD permite mantener un historial de cambios y colaborar de manera más eficiente, posibilitando la implementación de una infraestructura inmutable y robusta.

3.5 Resumen del capítulo

En este capítulo se presentan las principales herramientas y tecnologías utilizadas en el proyecto, destacando cómo se integran y complementan para lograr un entorno seguro y automatizado. En primer lugar, se describe la virtualización con Docker, que permite aislar aplicaciones en contenedores ligeros y portátiles, optimizando el despliegue y la escalabilidad. También se hace mención a Docker Compose como herramienta fundamental para orquestar y desplegar de forma conjunta los distintos servicios que conforman el entorno desarrollado. Su utilización ha permitido definir, automatizar y coordinar el arranque de todos los servicios sin necesidad de invadir con gran impacto sobre el código fuente original de las aplicaciones. Seguidamente, se aborda la implementación de Keycloak como servicio para la gestión centralizada de identidades y accesos, y la de Vault para la protección y administración de secretos (credenciales, claves, certificados…). Ambas soluciones se integran como servicios haciendo uso de contenedores Docker, aislando esta capa de seguridad y permitiendo una integración más fácil, evitando modificar mínimamente el código fuente de la aplicación.

Finalmente, se explica el uso de Terraform como herramienta de Infraestructura como Código (IaC), la cual automatiza la creación y configuración de estos servicios. Se define de manera declarativa

la infraestructura a desplegar. La combinación de estas tecnologías proporciona una base sólida para construir y gestionar entornos de aplicaciones haciéndolas más escalables, al tiempo que simplifica el control de cambios y la colaboración de diferentes trabajos de desarrollo que modificasen cualquier ámbito de la aplicación.

Capítulo 04: Descripción de la solución desarrollada

Una vez explicado los distintos protocolos y medidas de seguridad y herramientas que han ido surgiendo con el paso del tiempo, así como de las diferentes tecnologías de las que se compone este proyecto, el siguiente paso es explicar el desarrollo realizado para el presente proyecto. Todo el código fuente se encuentra en GitHub, en el enlace adjunto a continuación.

- Repositorio Git: KevBerja/TFM: Trabajo Final de Máster - Integración servicio de autentificación y gestión de usuarios para aplicaciones sin login

4.1 Descripción de la solución

El alcance de la solución desarrollada se ha llevado a cabo en las siguientes fases:

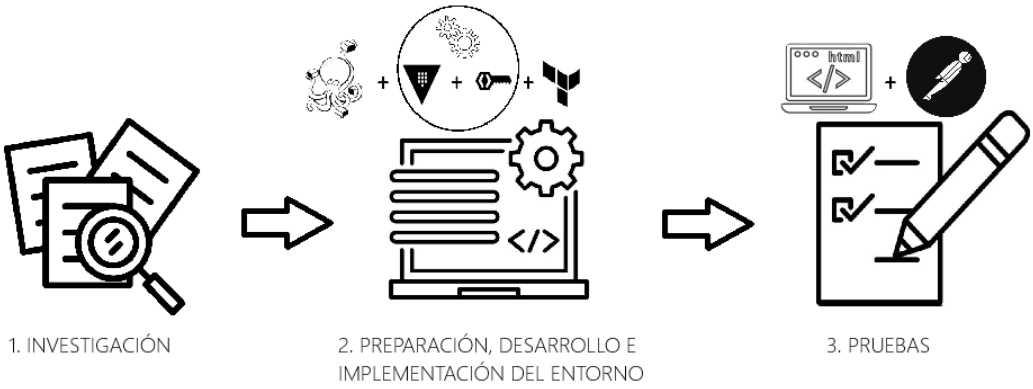

Figura 37. Fases de desarrollo de la solución

1. Investigación. En esta fase se lleva a cabo el estudio del protocolo OIDC y gestor de identidades y secretos y de las tecnologías disponibles.
2. Preparación, desarrollo e implementación del entorno. Terminada la fase de investigación y elección de tecnologías y entornos donde aplicarse, se realiza el desarrollo de la configuración e implementación de la capa de autenticación y autorización, así como su integración para las aplicaciones ejemplo y despliegue automatizado de toda la infraestructura.
3. Pruebas. Se realizan pruebas funcionales sobre esta capa en las aplicaciones ejemplo federadas y gestión de secretos vía web como vía API con POSTMAN.

Figura 38. Fase 1 - Investigación

En esta fase inicial, se han investigado las diferentes opciones existentes en el mercado para poder abordar el problema inicial.

En primer lugar, se ha llevado a cabo un análisis previo del protocolo de autorización OIDC (OpenID Connect), así como de las principales soluciones de gestión de identidades. Tras este estudio, se ha decidido hacer uso de Keycloak como proveedor de identidad OIDC y HashiCorp Vault como gestor centralizado de secretos, teniendo en cuenta la disponibilidad de imágenes oficiales para su despliegue mediante contenedores Docker y su gran comunidad, y así disponerlos ambos como servicios externos, ya que el objetivo es que se dispongan de éstos para un conjunto determinado de aplicaciones.

La elección de estas tecnologías permite una arquitectura en la que la gestión de identidades y la gestión de secretos queden claramente diferenciadas, pero perfectamente integradas. En este sistema, Keycloak emite tokens JWT firmados tras la autenticación del usuario, gestionando de forma centralizada usuarios, roles y permisos; mientras que Vault consume esos tokens JWT, los valida y, en función de ellos, emite tokens internos vinculados a políticas concretas, lo que permite acceder a secretos o generar credenciales dinámicas de forma segura y controlada.

Para la orquestación y despliegue de todos los servicios la herramienta que facilita esto en un entorno de desarrollo local es Docker Compose, el cual permite definir una estructura modular fácilmente replicable. Se ha tenido en cuenta la necesidad del desarrollo de scripts de inicialización para automatizar tareas clave como el des-sellado y configuración inicial de Vault o la espera de disponibilidad de la base de datos de Keycloak (PostgreSQL).

En cuanto a la gestión declarativa de la configuración de la infraestructura, se ha optado por el uso de Terraform frente a otras alternativas como Ansible. Aunque inicialmente se había valorado el uso de Ansible para la fase de configuración y Terraform solo para la infraestructura, la necesidad de definir la configuración de Keycloak en el momento mismo de su creación condicionó la decisión final de centralizarlo todo sobre Terraform. Cabe destacar que los límites entre infraestructura y configuración están cada vez más difusos, ya que herramientas como éstas están incorporando

funcionalidades cruzadas sin abandonar sus propósitos originales. Esta evolución refleja la naturaleza cambiante y competitiva del mercado DevOps actual, donde la flexibilidad y adaptabilidad de las herramientas son factores decisivos.

Tras esta fase de investigación y estudio previo se ha llegado a las siguientes conclusiones y estructura de la solución:

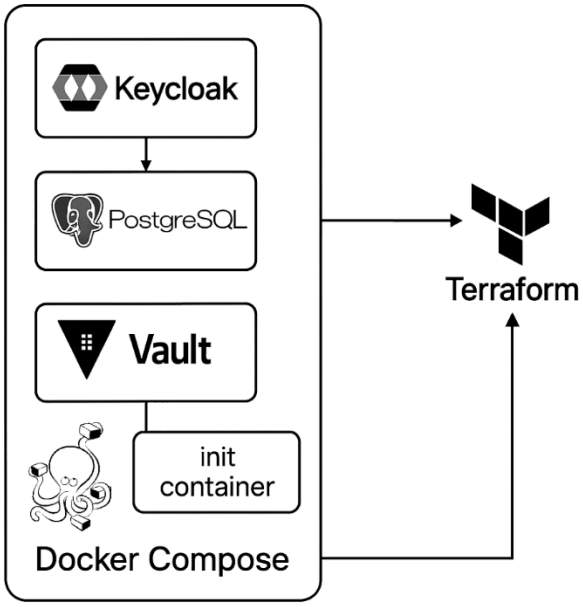

Figura 39. Estructura solución propuesta

1. El uso Keycloak junto con Vault ofrece una separación clara de roles (identidad vs. secretos) manteniendo una integración fluida.
2. Utilizar Docker Compose para la orquestación de servicios Docker combinado con scripts e *init-containers* permitiría, en un despliegue real, levantar el entorno de forma reproducible y sin intervención manual.
3. Automatizar la configuración previa con Terraform de toda la infraestructura de forma declarativa (la creación de realms, clientes OIDC y políticas, etc.), garantizando idempotencia, trazabilidad y fácil adaptación en los entornos de desarrollo y productivos.

4.1.1 Debilidades detectadas

Tras analizar el escenario de la solución, se han detectado las siguientes debilidades que podrían afectar al funcionamiento conjunto tanto de la solución, como de las aplicaciones que hace uso de ella:

1. En la configuración actual todos los componentes (Keycloak, PostgreSQL, Vault y su *init-container*) se levantan con Docker Compose en una única instancia de cada uno, sin replicación ni balanceo de carga. Si cualquiera de estos contenedores se detiene o falla, toda la capa de identidad o de gestión de secretos deja de estar disponible, incluyéndose consigo que los enlaces protegidos quedarán inaccesibles hasta que los servicios se restablezcan.

2. Al no estar configurado un mecanismo de auto-unseal (por ejemplo, mediante un KMS), tras cada reinicio Vault permanece "sellado" y rechaza todas las peticiones (HTTP 503) hasta que se ejecute manualmente el proceso de unseal.
3. PostgreSQL guarda sus datos en un volumen local de Docker. Si falla o se pierde el volumen, se pierden credenciales, configuraciones de realms y usuarios, inutilizando Keycloak y con ello y en consecuencia el resto de aplicaciones federadas.
4. Para simplificar, Vault escucha en HTTP sin TLS (tls_disable = true) y se utiliza disable_mlock = true en el contenedor. Esto expone los secretos en entornos no controlados, siendo solo idóneo en casos de entornos de desarrollo.
5. Docker Compose puede sufrir conflictos de puertos entre contenedores o problemas de permisos en los volúmenes, que requieren ajustes manuales cada vez que se reconstruye el entorno.

4.1.2 Plan de mitigación

Ante las disintinas debilidades que se han valorado en el apartado anterior, un posible plan de contingencia para cada una de ellas sería el siguiente:

1. Garantizar la disponibilidad de los servicios de Keycloak, PostgreSQL y Vault y que nunca se conviertan en un único punto de fallo. Para ello, se desplegarían en clústeres con réplicas activas y balanceo de carga. Keycloak tendría varias instancias redundantes conectadas a una base de datos PostgreSQL en configuración maestro-esclavo o gestionada, mientras que Vault utilizaría almacenamiento integrado replicado o Consul en modo cluster y auto-unseal mediante un KMS. Todo ello sería orquestado con Kubernetes.
2. Realizar copias de seguridad y/o réplicas de la base de datos PostgreSQL para asegurar una recuperación rápida tras cualquier incidente de Keycloak.
3. En entornos productivos toda comunicación con Vault y Keycloak quedaría cifrada mediante TLS, usando certificados firmados por una CA interna o Let's Encrypt. Además, se reactivaría la paulatina protección de memoria (mlock) donde esté permitido, para evitar que claves y secretos sensibles sean paginados a disco, mejorando así la confidencialidad y la integridad de los datos.
4. Cada servicio dispondría de sondas de liveness y readiness configuradas en Kubernetes para detectar y auto-recuperar fallos a nivel de pod, y se definirían alertas que avisen ante cualquier degradación de servicio.
5. Se podrían almacenar en la memoria caché local los tokens JWT válidos durante su ciclo de vida, reduciendo el impacto de cortes breves y mejorando la experiencia de usuario cuando ocurran fallos puntuales.

4.2 Preparación del entorno

Una vez se ha realizado la primera fase de investigación, se procede a llevar a cabo la preparación, desarrollo y despliegue del entorno de la solución.

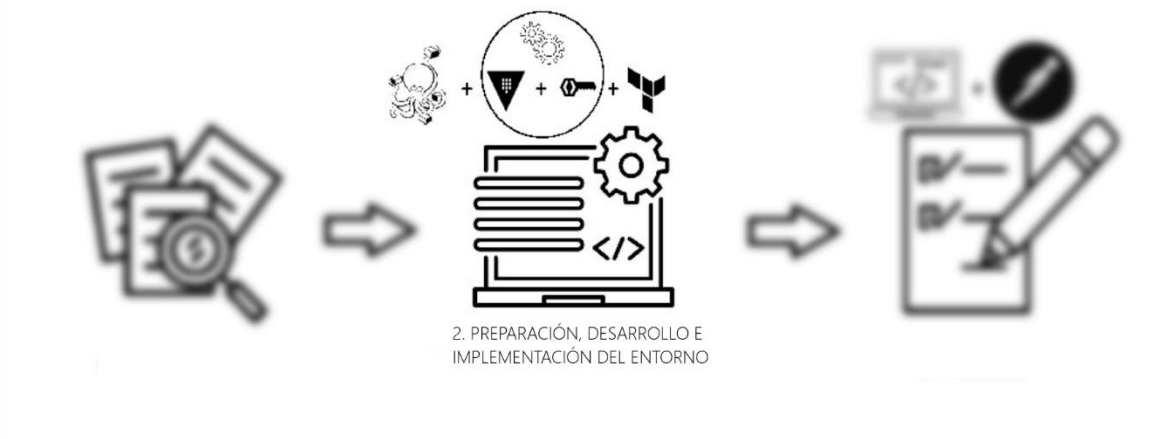

Figura 40. Fase 2. Preparación, desarrollo e implementación del entorno

El primer paso consiste en iniciar los servicios de Keycloak y Vault. Como primer punto en el objetivo de desarrollo de este sistema se trató de conseguir una configuración en la que Vault permitiera mediante la creación de credenciales almacenadas en él autentificarse por OIDC por medio del proveedor Keycloak, y entender con ello su lógica de funcionamiento. El entorno está compuesto por los siguientes servicios:

- Keycloak (proveedor de identidad).
- PostgreSQL (base de datos para Keycloak).
- Vault (gestor de secretos).
- vault-init (script automático de inicialización y des-sellado de Vault).

La siguiente imagen muestra la configuración del servicio keycloak y keycloak-db:

```yaml
services:

  keycloak:
    image: quay.io/keycloak/keycloak:26.1.3
    container_name: keycloak
    hostname: keycloak
    user: root
    environment:
      KC_BOOTSTRAP_ADMIN_USERNAME: admin
      KC_BOOTSTRAP_ADMIN_PASSWORD: admin
      KC_DB: postgres
      KC_DB_URL_HOST: keycloak-db
      KC_DB_URL_PORT: 5432
      KC_DB_URL_DATABASE: keycloak
      KC_DB_USERNAME: kcv239
      KC_DB_PASSWORD: inlumine.ual.es
    ports:
      - "8080:8080"
      - "8443:8443"
    networks:
      - tfm-net
    volumes:
      - ./keycloak/volume/data:/opt/keycloak/data
      - ./keycloak/init-keycloak.sh:/keycloak/init-scripts/init-keycloak.sh
      - ./keycloak/providers:/opt/keycloak/providers
    entrypoint: ["/bin/sh", "/keycloak/init-scripts/init-keycloak.sh"]
    depends_on:
      - keycloak-db
    restart: on-failure

  keycloak-db:
    image: postgres:13.3
    container_name: keycloak-db
    hostname: keycloak-db
    environment:
      POSTGRES_DB: keycloak
      POSTGRES_USER: kcv239
      POSTGRES_PASSWORD: inlumine.ual.es
    volumes:
      - ./db/volume/postgresql/data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    networks:
      - tfm-net
    restart: on-failure
```

Figura 41. Configuración servicio Keycloak

(Enlace al archivo GitHub: [TFM/docker-compose.yml at main · KevBerja/TFM])

Para el contenedor **keycloak** se usa la imagen de **quay.io/keycloak/keycloak:26.1.3** y se configuran variables de entorno que definen la conexión a la base de datos PostgreSQL para el almacenamiento de las distintas configuraciones, usuario y contraseña, así como el puerto de escucha de la base de datos, scripts de inicialización y resto de dependencias para su inicialización.

El contenedor **keycloak-db** es creado a partir de una imagen de PostgreSQL, con credenciales que coincidirán con las definidas en el contenedor keycloak para el almacenamiento de datos y configuraciones y, también utilizando volúmenes para persistencia.

Este es el script que automatiza la inicialización del entorno de Keycloak:

```bash
#!/bin/bash

# Crear el directorio de proveedores si no existe
mkdir -p /opt/keycloak/providers

# Definir la ruta y versión del controlador JDBC
JDBC_DRIVER_VERSION="42.2.23"
JDBC_DRIVER_FILE="postgresql-${JDBC_DRIVER_VERSION}.jar"
JDBC_DRIVER_PATH="/opt/keycloak/providers/${JDBC_DRIVER_FILE}"

# Verificar si el controlador ya existe
if [ -f "${JDBC_DRIVER_PATH}" ]; then
  echo "El controlador JDBC de PostgreSQL ya está presente en ${JDBC_DRIVER_PATH}."
  chmod a+r "${JDBC_DRIVER_PATH}"
else
  echo "Error: No se ha cargado el controlador JDBC de PostgreSQL."
  exit 1
fi

echo "Esperando a que Postgresql esté disponible en el puerto 5432..."
while ! (echo > /dev/tcp/keycloak-db/5432) 2>/dev/null; do
  sleep 1
done
echo "Postgresql está listo. Iniciando Keycloak..."

# Iniciar Keycloak
exec /opt/keycloak/bin/kc.sh start-dev
```

Figura 42. Script inicialización servicio Keycloak

(Enlace al archivo GitHub: [TFM/keycloak/init-keycloak.sh at main · KevBerja/TFM](#))

- Se crea el directorio de proveedores para alojar el driver JDBC.
- Se comprueba si el controlador JDBC de PostgreSQL está disponible.
- Queda en espera la disponibilidad del servicio PostgreSQL antes de lanzar Keycloak en modo desarrollo mediante **kc.sh start-dev**, para asegurar que Keycloak no intente arrancar antes de que la base de datos esté lista.

A continuación, se muestra la configuración del servicio Vault y su init-container:

```yaml
  vault:
    image: hashicorp/vault:1.18
    container_name: vault
    hostname: vault
    ports:
      - "8200:8200"
    volumes:
      - ./vault/config:/vault/config
      - ./vault/volume/data:/vault/data
      - ./vault/volume/logs:/vault/logs
    command: ["sh", "-c", "vault server -config=/vault/config/config.hcl"]
    cap_add:
      - IPC_LOCK
    networks:
      - tfm-net
    restart: on-failure

  vault-init:
    image: hashicorp/vault:1.18
    container_name: vault-init
    depends_on:
      - vault
    volumes:
      - ./vault/init-vault.sh:/init-vault.sh
      - ./vault/volume/data:/vault/data
    networks:
      - tfm-net
    entrypoint: ["sh", "-c", "apk add --no-cache curl jq && sh /init-vault.sh"]
    restart: on-failure
```

Figura 43. Configuración servicio Vault

(Enlace al archivo GitHub: [TFM/docker-compose.yml at main · KevBerja/TFM](#))

- El contenedor **vault-init** ejecuta un script de inicialización **init-vault.sh**. Está configurado para depender del servicio vault, de modo que espera a que este esté completamente operativo antes de proceder. Además, se añade la capacidad **IPC_LOCK** para proteger las claves en memoria. Vault queda expuesto en el puerto 8200.

  Este es el script de inicialización init-vault.sh:

```
#!/bin/sh

export VAULT_ADDR='http://vault:8200'

# Esperar a que Vault este disponible
until curl -s $VAULT_ADDR/v1/sys/health > /dev/null; do
  echo "Esperando a que Vault esté disponible..."
  sleep 1
done

# Verificar si Vault esta inicializado
if [ "$(vault status -format=json | jq -r '.initialized')" = "false" ]; then
  echo "Inicializando Vault..."
  vault operator init -key-shares=1 -key-threshold=1 -format=json > /vault/data/keys.json

  UNSEAL_KEY=$(jq -r '.unseal_keys_b64[0]' /vault/data/keys.json)
  ROOT_TOKEN=$(jq -r '.root_token' /vault/data/keys.json)

  # Des-sellar Vault
  vault operator unseal $UNSEAL_KEY

  echo "Vault inicializado y des-sellado."
else
  echo "Vault ya está inicializado. Des-sellando..."
  UNSEAL_KEY=$(jq -r '.unseal_keys_b64[0]' /vault/data/keys.json)
  vault operator unseal $UNSEAL_KEY
  echo "Vault des-sellado."
fi
```

Figura 44. Script inicialización servicio Vault

(Enlace al archivo GitHub: TFM/vault/init-vault.sh at main · KevBerja/TFM)

Este script realiza la configuración inicial de arranque llevando a cabo las siguientes instrucciones:

1. Establece la dirección de Vault (**http://vault:8200**).
2. Espera hasta que el servicio esté accesible.
3. Comprueba si Vault ya está inicializado:
   a. Si no lo está, lo inicializa con una única clave de des-sellado.
   b. Guarda la clave de des-sellado y el token raíz en un fichero JSON.
4. Des-sella Vault automáticamente. En caso de que ya esté inicializado, simplemente ejecuta el des-sellado usando la clave existente.

Esta imagen muestra el contenido del archivo config.hcl, que define la configuración principal del servicio Vault.

```
storage "file" {
  path = "/vault/data"
}

listener "tcp" {
  address     = "0.0.0.0:8200"
  tls_disable = true
}

plugin_directory = "/vault/plugins"

api_addr = "http://0.0.0.0:8200"
ui = true

disable_mlock = true
```

Figura 45. Archivo config.hcl Vault

(Enlace al archivo GitHub: TFM/vault/config/config.hcl at main · KevBerja/TFM)

- **storage "file":** Define la ruta de almacenamiento como archivos locales. Los datos persistirán en el directorio /vault/data dentro del contenedor. Esta opción es adecuada para entornos de desarrollo o pruebas, aunque para entornos de producción reales lo ideal es hacer apoyo sobre alguna plataforma cloud o servicio externo *Active Directory* (AD).
- **listener "tcp":** Configura el servicio para escuchar conexiones entrantes a través del protocolo TCP en todas las interfaces (0.0.0.0) y en el puerto 8200. Se desactiva TLS (**tls_disable = true**) para simplificar el entorno en desarrollo. En producción, se recomienda habilitar HTTPS. desactiva el uso de TLS (*Transport Layer Security*), permitiendo que Vault acepte conexiones no cifradas a través del protocolo HTTP.
- **plugin_directory:** Indica el directorio donde Vault buscará *plugins* adicionales, como autenticadores o backends de secretos personalizados. En este caso, es /vault/plugins.
- **api_addr y ui = true:** Se establece la dirección pública de la API de Vault como http://0.0.0.0:8200 y se habilita la interfaz web (UI) para poder acceder desde el navegador.
- **disable_mlock = true:** Desactiva el uso de mlock, una función de seguridad que impide que el contenido de la memoria se intercambie al disco. Esta opción está desactivada por motivos de compatibilidad en contenedores, donde normalmente no se permite el uso de mlock.

El desarrollo de la automatización y configuración de la infraestructura con Terraform incluye la creación de realms, usuarios, clientes OIDC y políticas de acceso en Vault.

Se configuran los proveedores con variables externas que incluyen URLs, credenciales de administración y tokens:

- El proveedor de Keycloak usa el cliente, usuario y contraseña administrativos.
- El proveedor de Vault se configura con la dirección del servidor y un token generado en la inicialización.

En esta imagen se define la configuración inicial de Terraform, incluyendo los proveedores necesarios:

```hcl
terraform {
  required_providers {
    keycloak = {
      source  = "mrparkers/keycloak"
      version = ">= 3.6.0"
    }
    vault = {
      source  = "hashicorp/vault"
      version = ">= 3.14.0"
    }
  }
}

provider "keycloak" {
  url       = var.keycloak_url
  client_id = var.keycloak_client_id
  username  = var.keycloak_username
  password  = var.keycloak_password
  realm     = var.keycloak_realm
}

provider "vault" {
  address = var.vault_address
  token   = local.vault_token
}
```

Figura 46. Fichero init.tf – Providers

(Enlace al archivo GitHub: TFM/init.tf at main · KevBerja/TFM)

- **mrparkers/keycloak** permite gestionar recursos en Keycloak como realms, usuarios y clientes.
- **hashicorp/vault** se utiliza para definir autenticación, políticas y secretos en Vault. Además, se configura el proveedor keycloak con las credenciales administrativas y el proveedor vault con la URL del servidor y el token correspondiente.

```hcl
# Realm "tfm"
resource "keycloak_realm" "tfm" {
  realm   = var.keycloak_realm_name
  enabled = var.keycloak_realm_enabled
}

# Cliente "vault"
resource "keycloak_openid_client" "vault" {
  realm_id                = keycloak_realm.tfm.id
  client_id               = var.vault_client_id
  name                    = var.vault_client_name
  enabled                 = var.vault_client_enabled
  standard_flow_enabled   = var.vault_standard_flow_enabled
  access_type             = var.vault_access_type
  service_accounts_enabled = var.vault_service_accounts_enabled
  client_secret           = var.vault_client_secret
  valid_redirect_uris     = var.vault_valid_redirect_uris
  web_origins             = var.vault_web_origins
}

# Crear el usuario "kcv239"
resource "keycloak_user" "kcv239" {
  realm_id = keycloak_realm.tfm.id
  username = var.kcv239_username
  enabled  = var.kcv239_enabled

  initial_password {
    temporary = var.kcv239_temporary_password
    value     = var.kcv239_password_value
  }
}
```

Figura 47. Fichero init.tf – Configuración cliente OIDC Keycloak

(Enlace al archivo GitHub: TFM/init.tf at main · KevBerja/TFM)

Esta imagen muestra la configuración que automatiza la creación de los componentes necesarios para el cliente OIDC en Keycloak: el realm "tfm", el cliente "vault" y el usuario del cliente vault.

- Realm "tfm": se declara con el recurso keycloak_realm, especificando el nombre y si está habilitado. Este realm actuará como contenedor lógico para gestionar la autenticación centralizada de todas las aplicaciones del proyecto.
- Cliente "vault": Representa al servicio vault como cliente OpenID Connect dentro del realm "tfm". La configuración incluye:
  - ID y nombre del cliente.
  - Activación del flujo estándar de autenticación.
  - Habilitación de cuentas de servicio.
  - URIs válidas de redirección y orígenes web permitidos.
  - Clave secreta del cliente (client_secret).
- Usuario para cliente "vault": se crea como recurso keycloak_user, asociado al mismo realm. Se especifican el nombre de usuario, su activación y su contraseña inicial mediante el bloque initial_password, indicando si es temporal y su valor concreto (parametrizado por variables externas). Con este usuario es con el que se identifica una todo el sistema esté en funcionamiento.

```
87   # Configuracion OIDC usando vault_generic_endpoint
88   resource "vault_generic_endpoint" "oidc_config" {
89     depends_on   = [keycloak_realm.tfm]
90     path         = "auth/${vault_auth_backend.oidc.path}/config"
91     disable_read = false
92     data_json = jsonencode({
93       oidc_discovery_url = "${var.keycloak_url}/realms/tfm",
94       oidc_client_id     = var.vault_client_id,
95       oidc_client_secret = var.vault_client_secret,
96       default_role       = var.oidc_default_role
97     })
98   }
99
100  # Definicion del rol para OIDC vault
101  resource "vault_generic_endpoint" "oidc_role" {
102    path = "auth/${vault_auth_backend.oidc.path}/role/default"
103    data_json = jsonencode({
104      allowed_redirect_uris = var.oidc_allowed_redirect_uris,
105      user_claim            = var.oidc_user_claim,
106      role_type             = var.oidc_role_type,
107      oidc_scopes           = var.oidc_scopes,
108      bound_issuer          = "${var.keycloak_url}/realms/tfm",
109      policies              = var.oidc_policies,
110      ttl                   = var.oidc_ttl
111    })
112  }
113
114  # Politica "default" vault
115  resource "vault_policy" "default" {
116    name   = var.vault_policy_name
117    policy = var.vault_policy_content
118  }
```

Figura 48. Configuración OIDC Vault

(Enlace al archivo GitHub: TFM/init.tf at main · KevBerja/TFM)

Mediante vault_generic_endpoint, se configura Vault para autenticar mediante OIDC:

- Se define el *endpoint* auth/<oidc_path>/config, donde se especifican:
  - URL de descubrimiento de Keycloak.
  - ID y secreto del cliente configurado previamente.
  - Rol por defecto a asignar a los usuarios autenticados.

Posteriormente, se crea el rol OIDC llamado "default" en Vault:

- Se definen los URIs de redirección válidos (por ejemplo, localhost, Vault UI o Postman).
- Se especifican los scopes OIDC y el issuer (Keycloak).
- Se establecen las políticas que se aplicarán al usuario autenticado. Se crea una política en Vault llamada "default", con capacidades completas (create, read, update, delete, list) sobre todos los *paths*. Esta política se asocia al rol OIDC anteriormente definido, permitiendo que los usuarios autenticados tengan permisos adecuados dentro de Vault.

Como resultado de ello, se ha logrado disponer de los servicios clave (Keycloak con su base de datos PostgreSQL, Vault y su *init-container*) orquestados en Docker Compose listos para arrancar y comunicarse entre sí, con scripts automáticos que garantizan el arranque ordenado y controlado de éstos, y una infraestructura declarativa en Terraform que crea de forma reproducible los realms, clientes OIDC y políticas en Keycloak y Vault, estableciendo así una base automatizada y replicable para las pruebas de flujos OIDC y el acceso controlado a secretos.

4.3 Desarrollo e implementación del entorno

A continuación, y tras la preparación del entorno, el siguiente paso es desarrollar e implementar esta capa de autenticación y autorización en aplicaciones que no la disponen.

Se ha escogido como escenario de ejemplo tres aplicaciones web Flask con Python que ofrecen la posibilidad de realizar entrenar tres tipos distintos de modelos Machine Learning (uno por aplicación) y realizar predicciones simples o conjuntas para los conjuntos de datos en los que se hayan entrenado, así como inicializarlos para trabajar con un conjunto de datos distinto.

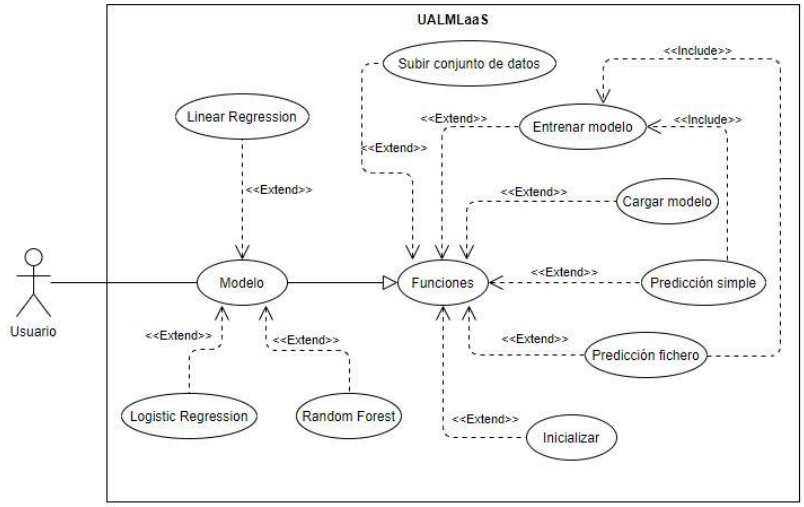

Figura 49. Diagrama de casos de uso app tfg

El usuario puede, una vez que ha escogido uno de los 3 modelos (Linear Regression, Logistic Regression o Random Forest) realizar lo siguiente:

1. Subir un conjunto de datos en formato .csv cuyo nombre sea train.csv, ya que será el fichero con ese nombre que reconocerá el modelo para posteriormente ser entrenado.
2. Entrenar modelo posteriormente de haber subido el dichero con el conjunto de datos y atribuirle un nombre. Cuando finalice el entrenamiento generará una los archivos que componen el motor de machine learning del modelo creando un directorio y una referencia con el nombre establecido.
3. Realizar una predicción simple con un conjunto de datos de entrada en formato JSON.
4. Realizar una predicción masiva registro a registro sobre un fichero de datos adjuntado como entrada en formato .csv.
5. Consultar y elegir dentro de un listado de modelos entrenados el que quiera utilizarse.
6. Inicializar el modelo, limpiando el modelo entrenado actual en uso.

La pantalla principal que contiene las 3 aplicaciones Python es la siguiente:

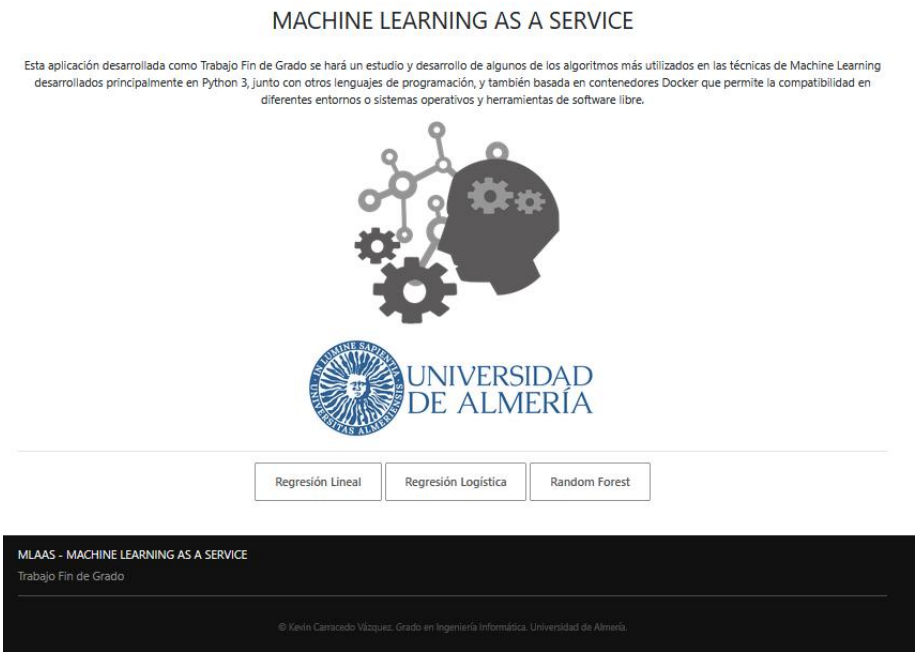

Figura 50. Inferfaz app UALMLaaS

En esta pantalla se muestra el conjunto de opciones descritas anteriormente que aparecerían para cada modelo y que se restringirán su acceso a aquellos usuarios que se identifiquen y estén autorizado, y en donde este título cambiaría para que el usuario sepa en qué aplicación se encuentra.

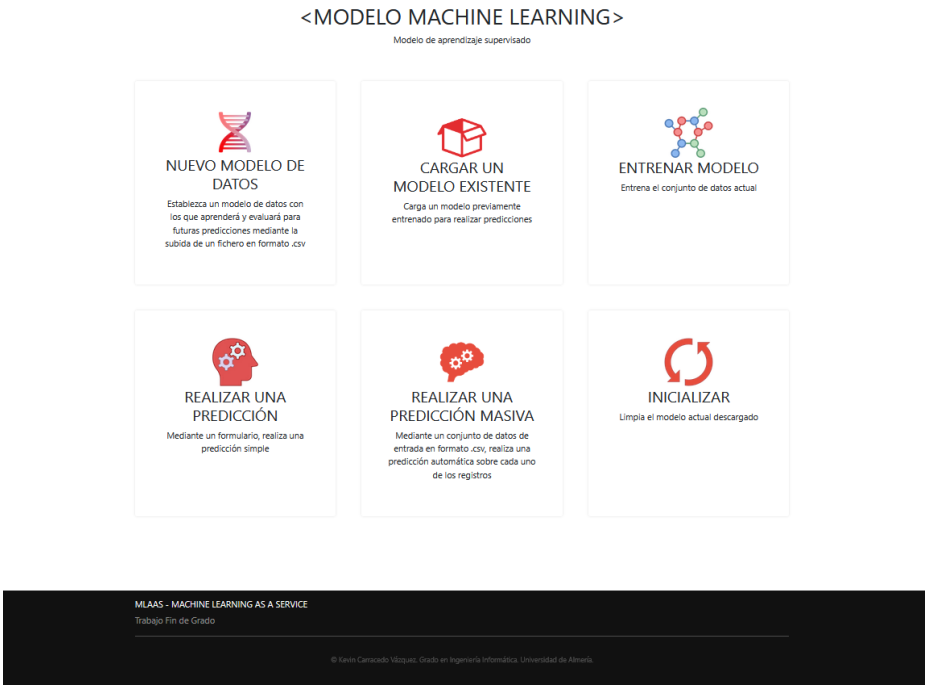

Figura 51. Página principal modelo ML

La sección del fichero docker-compose.yml que contiene estas tres aplicaciones en las que se aplica la solución de este trabajo es el siguiente:

```
118     linear-regression:
119       container_name: linear-regression
120       hostname: linear-regression
121       build:
122         context: ./linear-regression
123         dockerfile: Dockerfile
124       ports:
125         - "5000:5000"
126       networks:
127         - tfm-net
128       restart: on-failure
129
130     logistic-regression:
131       container_name: logistic-regression
132       hostname: logistic-regression
133       build:
134         context: ./logistic-regression
135         dockerfile: Dockerfile
136       ports:
137         - "5001:5001"
138       networks:
139         - tfm-net
140       restart: on-failure
141
142     random-forest:
143       container_name: random-forest
144       hostname: random-forest
145       build:
146         context: ./random-forest
147         dockerfile: Dockerfile
148       ports:
149         - "5002:5002"
150       networks:
151         - tfm-net
152       restart: on-failure
153
154   networks:
155     tfm-net:
156       driver: bridge
```

Figura 52. Servicios app tfg docker-compose.yml

(Enlace al archivo GitHub: TFM/docker-compose.yml at main · KevBerja/TFM)

Para federar estas aplicaciones y que utilicen el flujo OIDC para la autenticación ha sido necesario crear un cliente en Keycloak para que puedan identificarse como federadas, así como un fichero en formato JSON para cada una con las propiedades de conexión de dicho cliente para que una vez esté en funcionamiento los usuarios puedan autenticarse y acceder a los distintos *endpoints* de las aplicaciones que estén protegidos.

Se define un nuevo cliente "tfg" OpenID en el fichero init.tf de Terraform para Keycloak sobre el realm "tfm", que será con el que identificaremos la aplicación para el uso del *login* federado por OIDC.

```
47  # Cliente "tfg"
48  resource "keycloak_openid_client" "tfg" {
49    realm_id                = keycloak_realm.tfm.id
50    client_id               = var.tfg_client_id
51    name                    = var.tfg_client_name
52    enabled                 = var.tfg_client_enabled
53    standard_flow_enabled   = var.tfg_standard_flow_enabled
54    access_type             = var.tfg_access_type
55    service_accounts_enabled = var.tfg_service_accounts_enabled
56    valid_redirect_uris     = var.tfg_valid_redirect_uris
57    client_secret           = var.tfg_client_secret
58    web_origins             = var.tfg_web_origins
59  }
```

Figura 53. Configuración cliente tfg OIDC Keycloak Terraform

(Enlace al archivo GitHub: TFM/init.tf at main · KevBerja/TFM)

Esta imagen muestra el fichero client_secrets.json que define los parámetros de configuración para la autenticación OpenID Connect en una de las aplicaciones que compone el cliente tfg. En él ha de ir informados el client_id, client_secret, URI del emisor (issuer), endpoints de autorización y URI para el token, así como las URIs válidas de redirección. Estos datos permiten que la aplicación web se comunique correctamente con el servidor Keycloak y gestione el proceso de *login* OIDC federado, es decir, la aplicación estará federada.

```
1
2   {
3     "web": {
4       "issuer": "http://keycloak:8080/realms/tfm",
5       "auth_uri": "http://keycloak:8080/realms/tfm/protocol/openid-connect/auth",
6       "client_id": "tfg",
7       "client_secret": "inlumine.ual.es",
8       "redirect_uris": ["http://localhost:5000/*"],
9       "token_uri": "http://keycloak:8080/realms/tfm/protocol/openid-connect/token",
10      "userinfo_uri": "http://keycloak:8080/realms/tfm/protocol/openid-connect/userinfo"
11    }
12  }
```

Figura 54. Fichero client_secrets.json

(Enlace al archivo GitHub: [TFM/linear-regression/client_secrets.json at main · KevBerja/TFM](#))

Para algunas aplicaciones es necesario importar una librería para poder utilizar el login federado OIDC. En este caso es necesario importar el módulo OpenIDConnect desde flask_oidc. Este fichero es el que identificará a la aplicación federada. En ella se especifica:

- La ruta al archivo client_secrets.json, que contiene los parámetros de conexión con el proveedor de identidad Keycloak.
- La desactivación del uso obligatorio de cookies seguras (OIDC_ID_TOKEN_COOKIE_SECURE: False) para facilitar pruebas locales en HTTP.
- El uso del *scope* básico openid, necesario para identificar al usuario.
- El método de autenticación del endpoint de introspección como client_secret_post, conforme a las recomendaciones del flujo estándar.

Finalmente, se inicializa la instancia oidc que permitirá aplicar la autenticación a rutas protegidas de la aplicación, haciendo uso del decorador @oidc.require_login.

```
18  app = Flask(__name__)
19  app.secret_key = 'kcv239LinearRegression'
20
21  # Configuracion de OIDC
22  app.config.update({
23      'OIDC_CLIENT_SECRETS': './client_secrets.json',
24      'OIDC_ID_TOKEN_COOKIE_SECURE': False,
25      'OIDC_SCOPES': ['openid'],
26      'OIDC_INTROSPECTION_AUTH_METHOD': 'client_secret_post'
27  })
28
29  oidc = OpenIDConnect(app)
```

Figura 55. OIDC para aplicaciones Flask

(Enlace al archivo GitHub: [TFM/linear-regression/api.py at main · KevBerja/TFM](#))

En esta sección de código Python se observa una parte del backend de la aplicación desarrollada, implementado con Flask y decoradores de autenticación mediante OIDC. Las rutas como /login, /logout, /loadInitCSV y /uploadInitCSV están protegidas, lo que garantiza que solo usuarios autenticados puedan acceder.

```python
@app.route('/login')
@oidc.require_login
def login():
    # Si se ha iniciado sesion redirige a la pagina de inicio de la app
    return redirect(url_for('home'))

@app.route('/logout')
def logout():
    oidc.logout()
    flash("Sesión cerrada")
    return redirect(url_for('home'))

@oidc.require_login
@app.route('/loadInitCSV', methods=['GET'])
def upload_file():
    return render_template('subida_fichero.html')

@app.route('/uploadInitCSV', methods=['POST'])
def uploader():
    if request.method == 'POST':
        file_form = request.files['file_request']
        file = secure_filename(file_form.filename)

        filename = file.split('.')
        f_name = filename[0]
        f_extension = filename[-1]

        if (f_name != "train" and f_extension != "csv"):
            flash("ERROR - Archivo no válido. El fichero de datos de entrenamiento debe estar en formato .csv con el nombre --train.csv--")

            return redirect('/loadInitCSV')


        os.makedirs('./static/model_temp', exist_ok=True)
        file_form.save(os.path.join('', 'static/model_temp/' + file))

        flash("Archivo de datos subido con éxito")

        return redirect('/loadInitCSV')
```

Figura 56. Protección endpoints

(Enlace al archivo GitHub: [TFM/linear-regression/api.py at main · KevBerja/TFM](#))

4.4 Pruebas

En el siguiente apartado se describen las pruebas realizadas de todas las configuraciones realizadas.

Figura 57. Fase 3 - Pruebas

En esta última fase, se han realizado varias pruebas sobre el trabajo realizado que van desde el *login* por protocolo OIDC con Keycloak desde Vault hasta el funcionamiento e integración con una aplicación de ejemplo y explotación del token vía POSTMAN.

4.4.1 Pruebas Web

En esta primera imagen se muestra la minitoma de Keycloak cuando se trata de autenticar en el servicio de Keycloak desde Vault. Se puede observar el login OIDC personalizado hacia el realm "tfm".

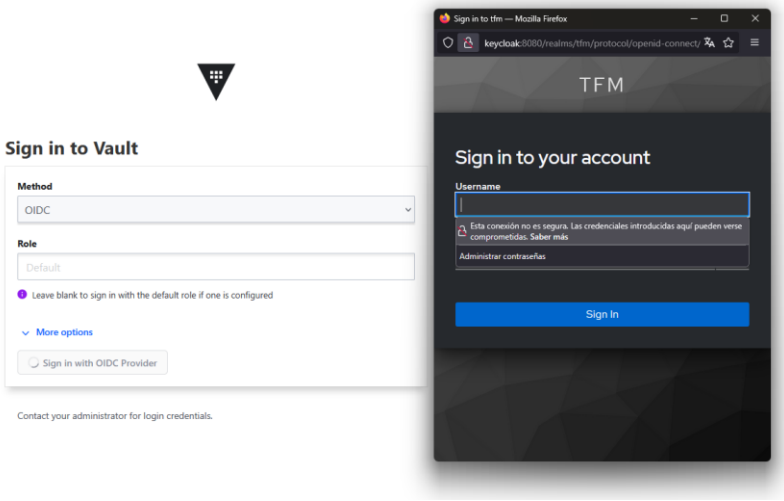

Figura 58. Login OIDC Vault

Una vez el login se lleva a cabo con éxito, se accede al dashboard de Vault del usuario logueado. Como se puede observar en el panel izquierdo al pulsar sobre el icono del usuario, aparece el token del usuario actual, así como otras opciones rápidas como renovar vida del token, vencerlo o cerrar la sesión.

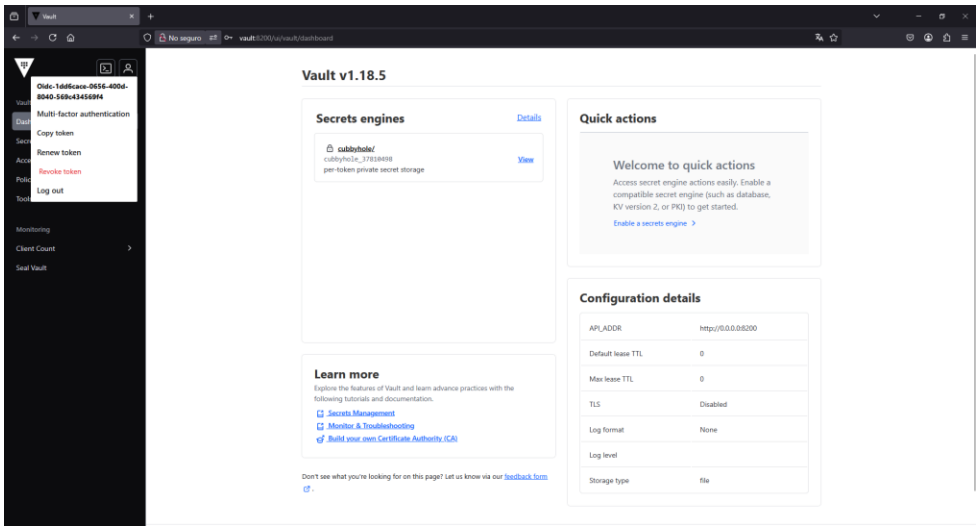

Figura 59. Dashboard cliente Vault

Dentro de la opción "Authentication Methods" se puede comprobar que la configuración aplicada con Terraform se ha realizado correctamente, teniendo el path "/oidc" donde estarán los tokens de los distintos usuarios autenticados por este método.

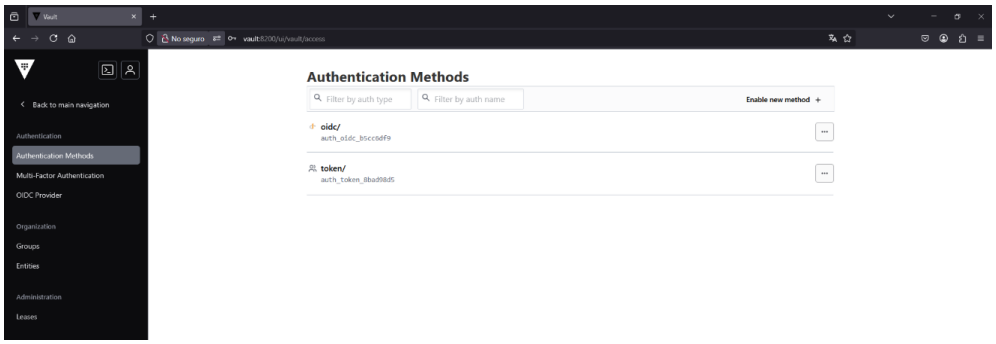

Figura 60. Path protocolos autentificación Vault

Para ver más detalle del token en cuestión se accede a la opción "Entities" del menú. En la sección mostrada, se evidencia la existencia del token generado dinámicamente para el usuario autenticado, reflejando con ello la integración entre los servicios Keycloak y Vault.

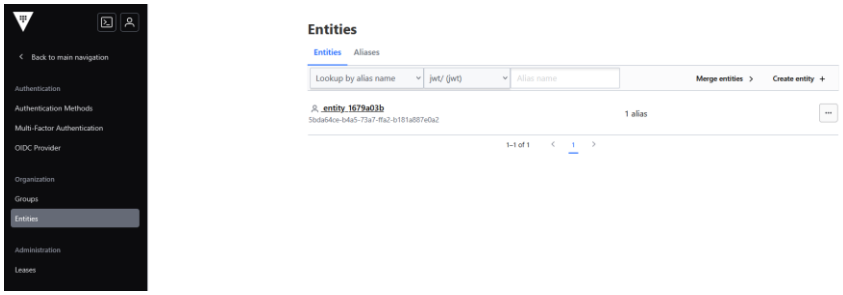

Figura 61. Entities

Aquí se muestran los tokens validados vía OIDC. Se puede observar cuándo se creó, así como la fecha de su última actualización, la política aplicada y los *paths* accedidos, cumpliendo con uno de los objetivos de seguridad marcados para el desarrollo de la solución.

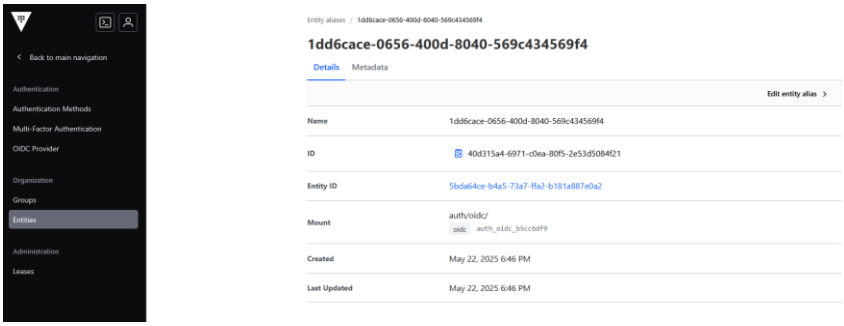

Figura 62. Entities Details

En el subapartado Metadata del token se puede ver el rol definido "default" durante la configuración de Vault. Este comportamiento valida la aplicación de roles y permisos asignados a los usuarios configurados.

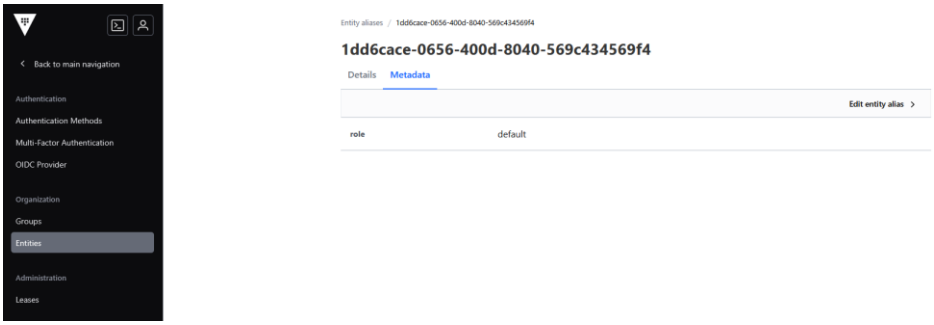

Figura 63. Entities Metadata

Por defecto y como prueba se establece la vida útil del token de una hora, mostrando una alerta informativa al usuario del tiempo restante.

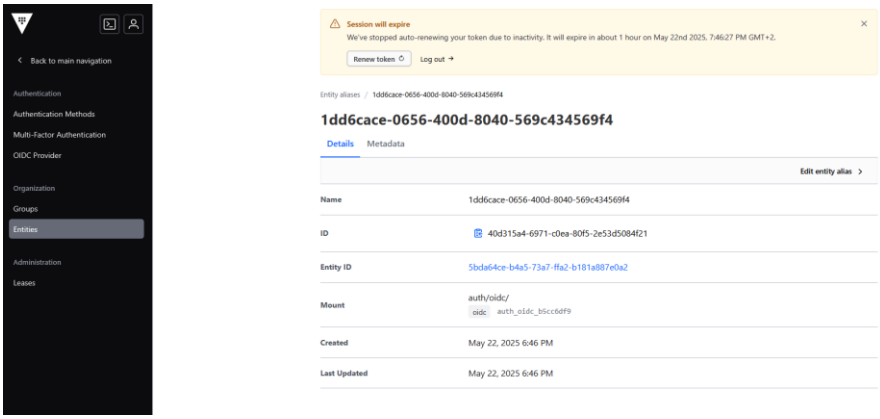

Figura 64. TTL (Token Time Life)

Por la parte de Keycloak, se accede a su interfaz web de administración con el usuario admin para comprobar y validar la configuración aplicada con Terraform. En esta captura se muestra el listado de clientes configurados dentro del realm "tfm". Dentro de los múltiples clientes que se listan para el protocolo OpenID Connect, entre ellos se encuentra "vault", con el que se registra al servicio de Vault como usuario para la autentificación por OIDC.

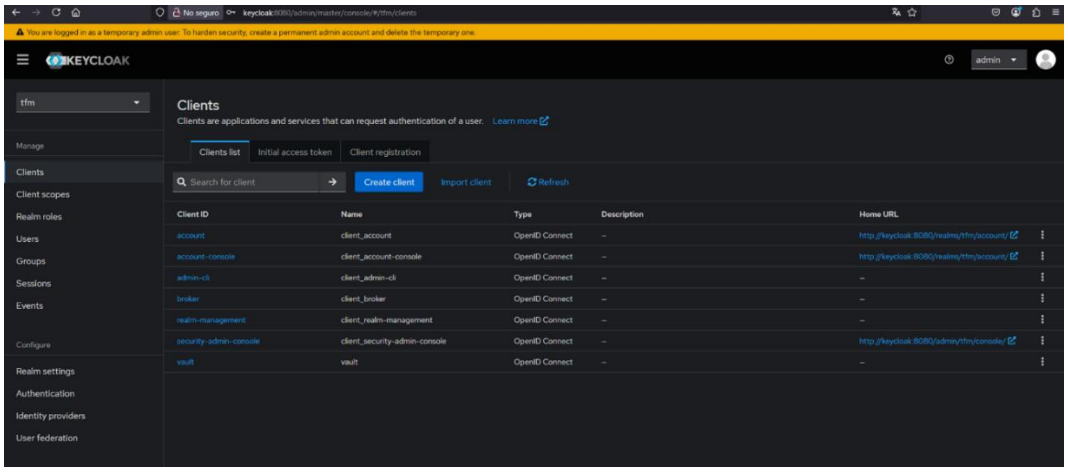

Figura 65. Clientes realm tfm Keycloak

Dentro de la sección de "Users", aparece el usuario con el que se había iniciado sesión con anterioridad a través de Vault y con el que se ha validado su autentificación.

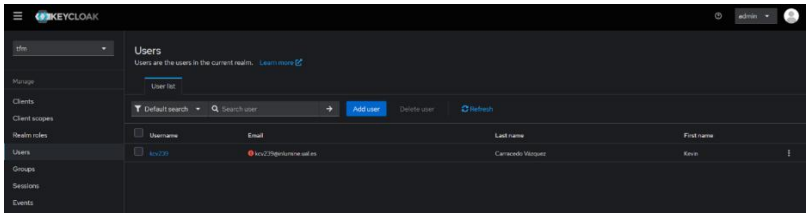

Figura 66. Usuarios realm tfm

En el cliente "vault", en el apartado Settings se encuentra la configuración realizada del Client ID y los URI de redirección válidos. Estos parámetros son esenciales para permitir que Vault utilice el flujo de autenticación OpenID Connect. El puerto 8200 es utilizado para la API de Vault, UI y autenticación OIDC, mientras que el puerto 8250 se utiliza para el redireccionamiento del OIDC hacia el entorno autenticado.

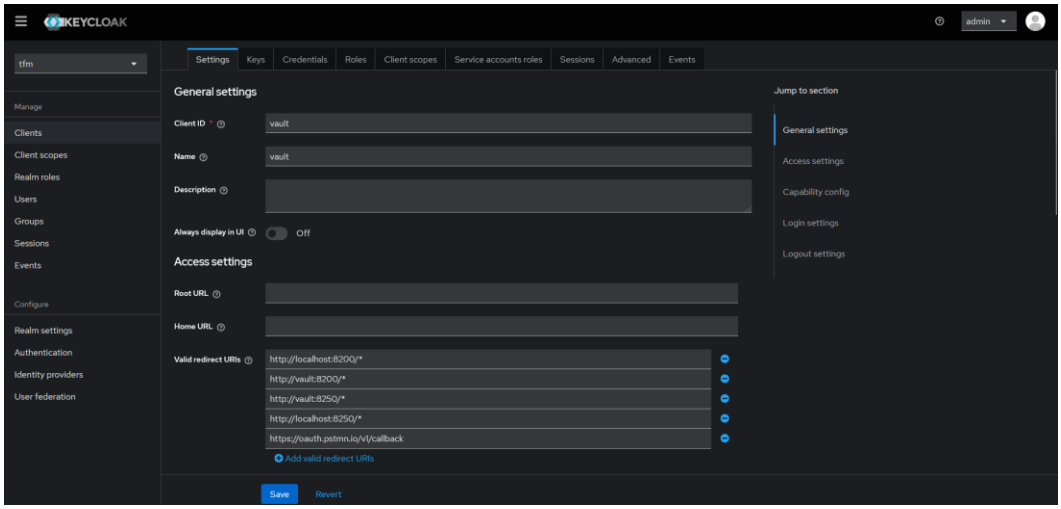

Figura 67. Configuración cliente vault Keycloak – 1

Avanzando en esta misma sección aparece habilitado el Client Authentication, así como los flujos Standard flow y Service accounts roles, necesarios para el funcionamiento del inicio de sesión OIDC y la obtención de tokens JWT válidos. Esta configuración garantiza que Vault pueda autenticar usuarios mediante Keycloak, con políticas definidas basadas en roles y servicios, como la definida anteriormente como "default".

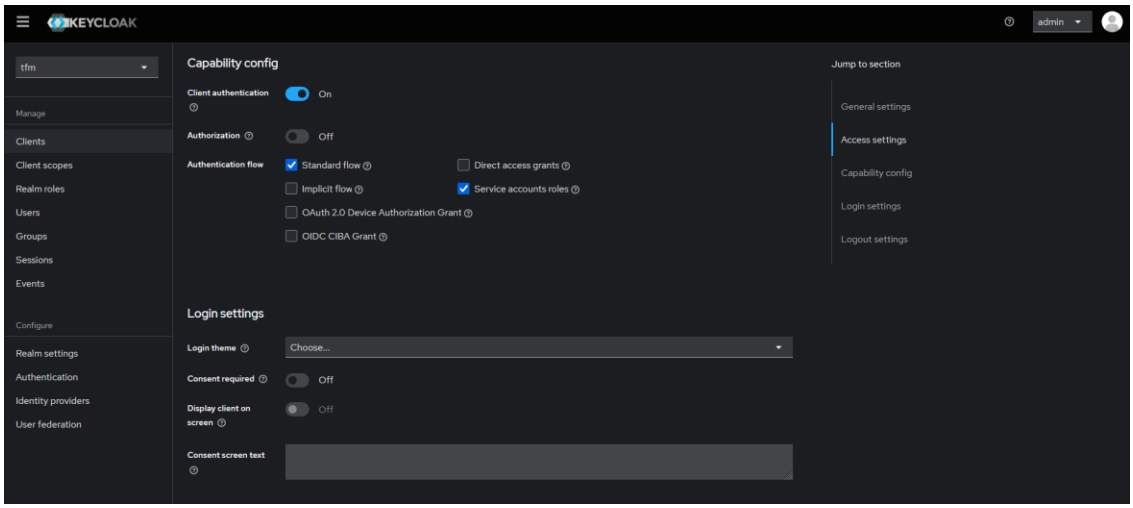

Figura 68. Configuración cliente vault Keycloak – 2

A continuación, se muestra la aplicación tfg compuesta por 3 aplicaciones Python, una en cada contenedor Docker y que escuchan en diferentes puertos.

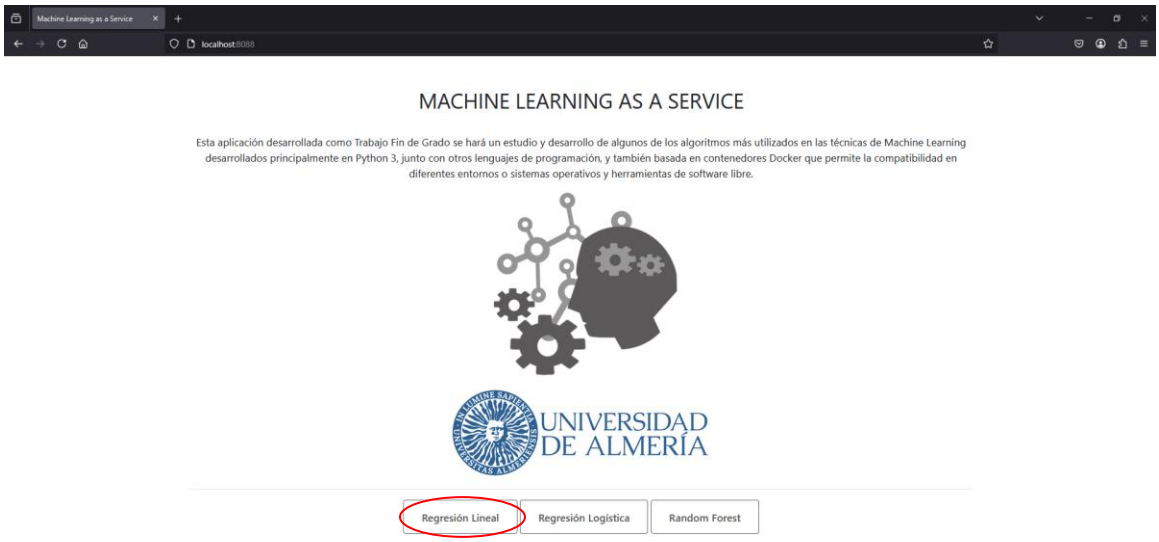

Figura 69. App web tfg

Si se intenta acceder a una de ellas, por ejemplo, a "Regresión Lineal" se pedirá la autentificación del usuario, ya que el *endpoint* que se está tratando de acceder de esa aplicación está

protegido. Incluso si el usuario conoce la ruta directa igualmente de le pedirá que se identifique a no ser que ya lo haya hecho previamente y su token no haya caducado o lo haya revocado.

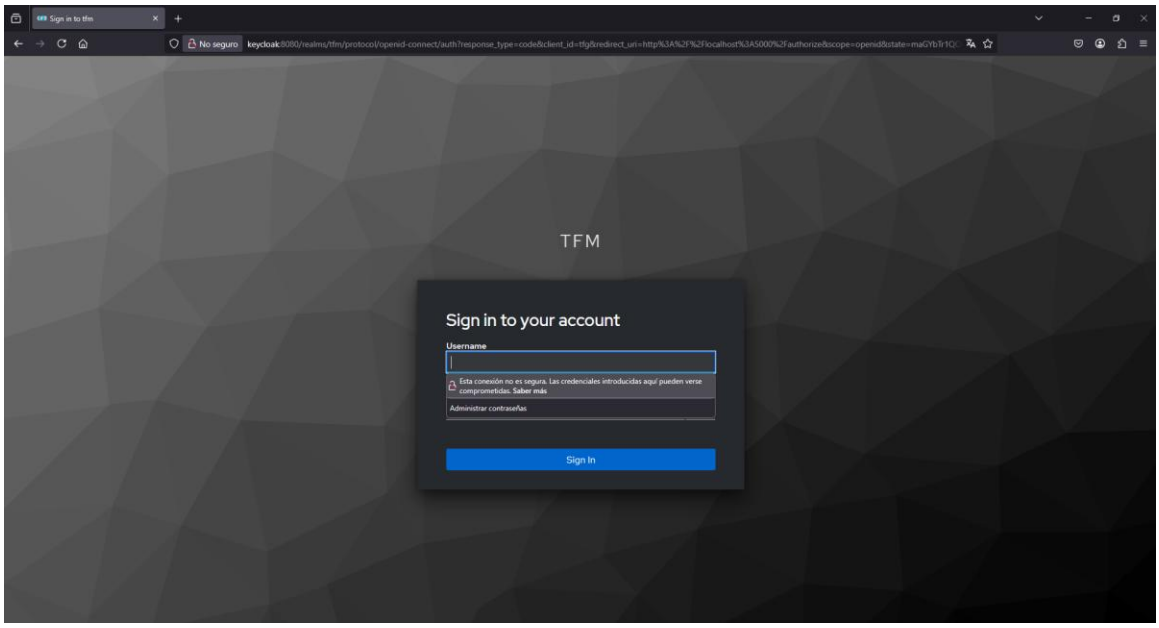

Figura 70. Login OIDC app tfg

4.4.2 Pruebas POSTMAN

En estas pruebas se trata mediante POSTMAN autenticarse por OIDC para obtener el token del usuario y hacer uso de él a modo de ejemplo en los *endpoints* protegidos de la aplicación, así como tratar de explotar la aplicación protegida sin permisos o con un token no válido o caducado.

En esta petición se lanza una petición de autenticación OIDC con el siguiente *body*, que corresponde a la autenticación realizada en el aparatado anterior con un usuario ya configurado para el real "tfm":

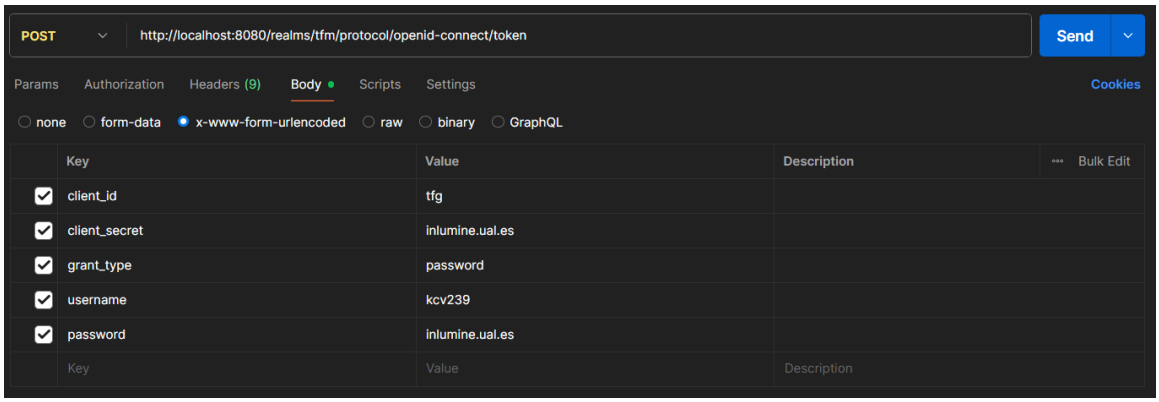

Figura 71. Body login token OIDC

Como respuesta se obtiene el *access_token*, el *refresh_token*, así como la caducidad de estos.

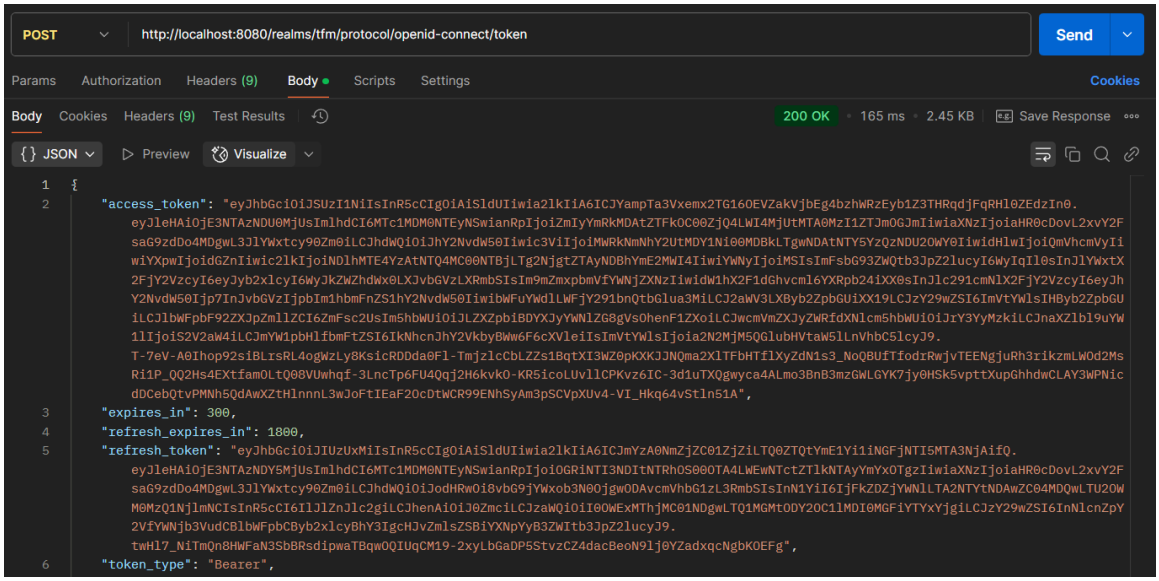

Figura 72. Endpoint login token OIDC

La siguiente prueba consiste en intentar obtener un token, pero con un usuario que no está registrado, obteniendo como respuesta de Keycloak una respuesta con el error "*unauthorizen_client*".

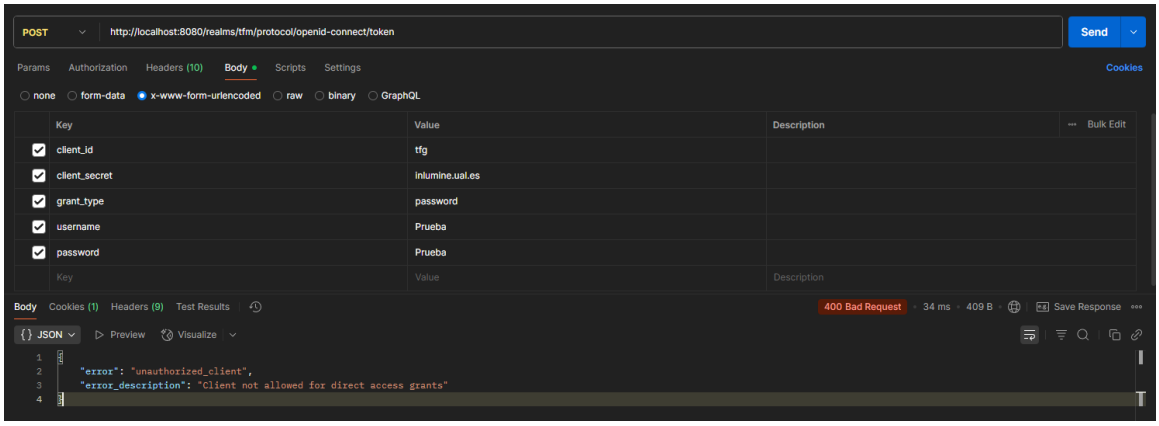

Figura 73. Endpoint login token OIDC erróneo

Con un token válido se llama a un *endpoint* de la aplicación federada obteniendo como respuesta un *HTTP 200 OK* con la página cargada.

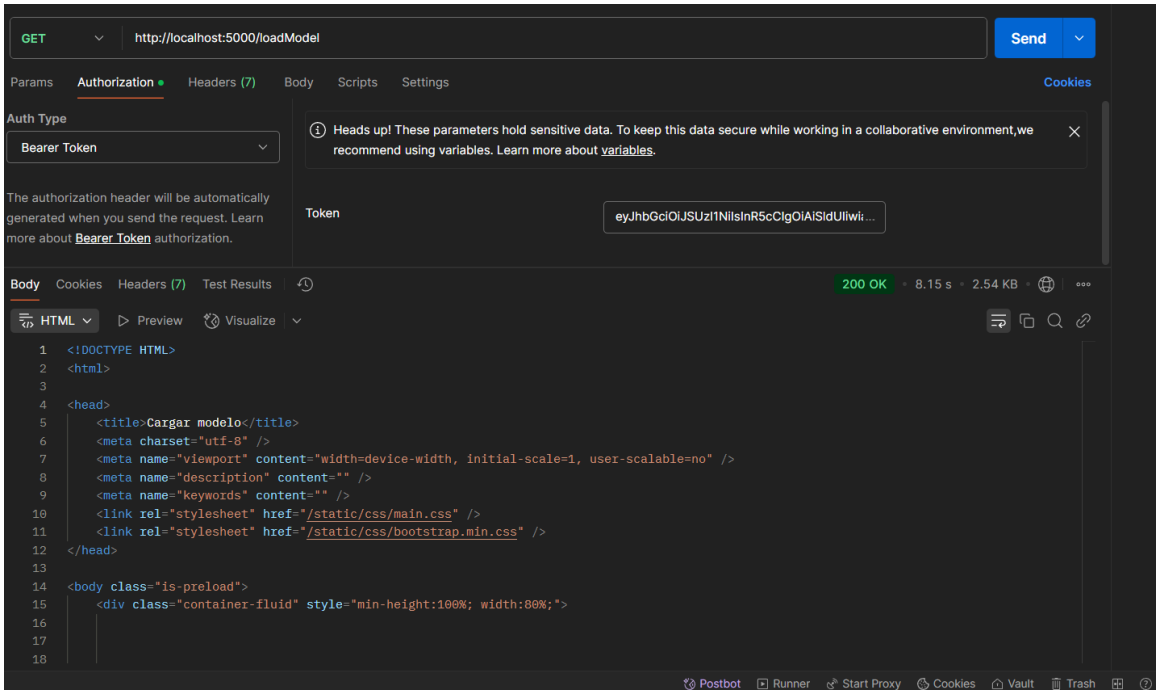

Figura 74. Endpoint con token válido

Si se trata de hacer lo mismo con otro *endpoint* diferente para el cual el usuario autenticado del token no tenga permisos se recibe como respuesta un *HTTP ERROR 402 UNAUTHROIZED*, impidiendo el acceso.

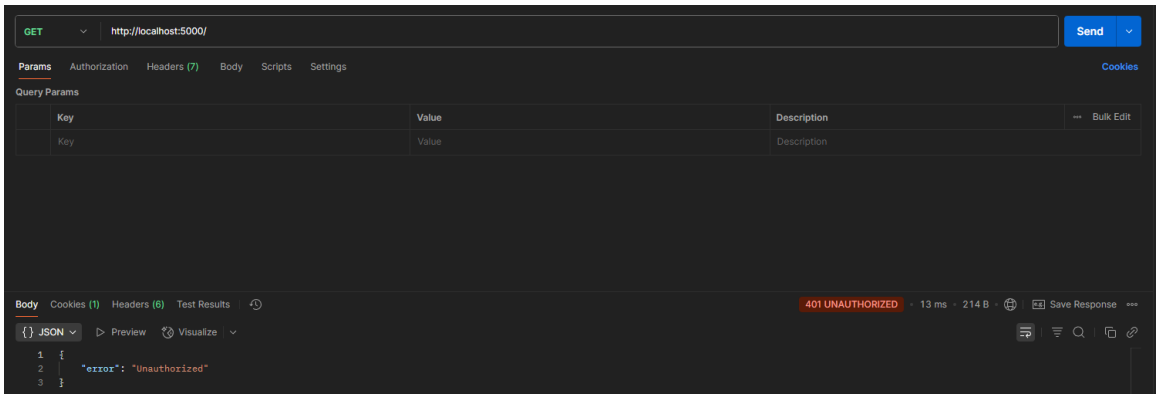

Figura 75. Endpoint usuario sin permisos

Si se trata de acceder a un *endpoint* de la aplicación una vez la vida útil del token emitido expira, se recibe como respuesta el error HTTP 401 UNAUTHORIZED "*token_expired*", siendo necesario autenticarse de nuevo por OIDC.

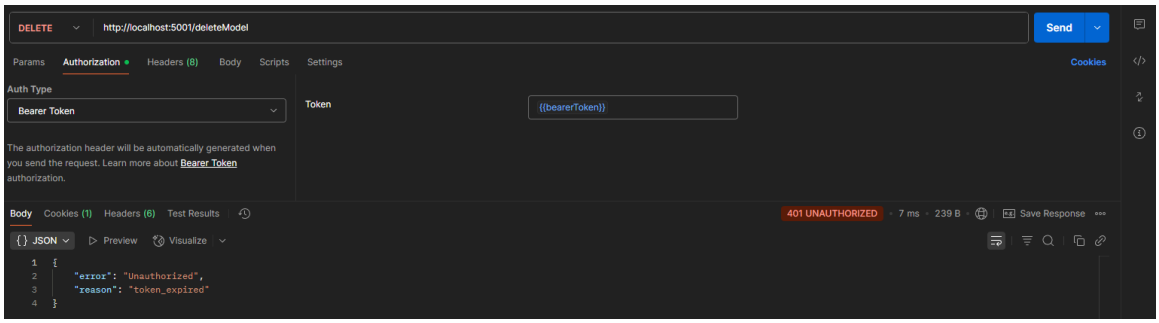

Figura 76. Endpoint token caducado

4.5 Problemas comunes del uso de Keycloak y Vault

En entornos de producción en donde soluciones como la que se aplica en este trabajo, resulta imprescindible no sólo conocer los fallos más frecuentes de los servicios de autenticación y gestión de secretos, sino también aplicar medidas concretas que permitan resolverlos con rapidez y minimizar el impacto en la operación. A continuación, se presentan, los problemas más comunes y habituales en entornos que utilizan Keycloak y Vault y cómo poder gestionarlos, ya que la pérdida temporal o total de estos servicios podrían llevar a la total inoperatividad de los usuarios al no poder identificarse correctamente.

4.5.1 Problemas comunes con Keycloak

- **Pérdida de conectividad con la base de datos.** Interrupciones de red, expiración de credenciales o saturación de conexiones provocan errores HTTP 500 y excepciones JDBCCommunicationsException. Para evitarlo, conviene desplegar la base de datos con réplicas o en modo clúster, ajustar el tamaño de los pools de conexión y aplicar patrones de reintentos en segundo plano continuos en el cliente. La monitorización continua del número de conexiones activas permite detectar picos anómalos antes de que se sature el servicio y anticiparse a este posible escenario.
- **Desincronización de sesiones en clúster.** La falta de replicación de sesiones impide que un nodo recién arrancado conozca las sesiones activas, obligando a los usuarios a reautenticarse. La solución consiste en almacenar las sesiones en un repositorio compartido (o Redis, por ejemplo) y verificar periódicamente la coherencia entre nodos tras reinicios o escalados.
- **Expiración de certificados TLS.** El vencimiento del certificado SSL/TLS del proxy o del Ingress bloquea el acceso por alertas de "sitio no seguro". Se debe automatizar la emisión y renovación (por ejemplo, con Cert-Manager y Let's Encrypt) para garantizar que los proxies se actualicen sin intervención manual.
- **Problemas de rendimiento por configuración de JVM insuficiente.** Parámetros de *heap* y *metaspace* inadecuados generan pausas prolongadas de Garbage Collector, produciendo *timeouts como consecuencia*. Para evitar esto, es preciso dimensionar la JVM según la carga real, ajustar el recolector (G1 o ZGC) y emplear herramientas APM (Prometheus + Grafana, New Relic) que alerten sobre fugas de memoria o saturación de CPU.

- **Errores en la sincronización LDAP/Active Directory.** Cambios en el esquema LDAP o cuentas bloqueadas impiden la federación de usuarios, lo que provoca fallos al importar o actualizar identidades. Se recomienda validar los cambios en un entorno de preproducción, implementar *health checks* específicos sobre el endpoint LDAP y notificar automáticamente al equipo de seguridad ante incidencias de autenticación.

4.5.2 Problemas comunes con Vault

- **Nodo "sealed" tras reinicio sin auto-unseal.** Si no está configurado el auto-unseal, el servicio permanece sellado tras cada arranque y rechaza todas las solicitudes con una respuesta del servidor HTTP 503. La habilitación de auto-unseal mediante un KMS (AWS KMS, Azure Key Vault o Google KMS) permite el desbloqueo automático. Para esto conviene validar cada cierto tiempo los permisos sobre la clave maestra y, en entornos on-premise, contar con scripts seguros que utilicen tokens de recuperación almacenados de forma cifrada.
- **Caducidad o revocación de la clave de auto-unseal en el KMS.** La expiración de la clave maestra o la revocación de permisos en el proveedor KMS bloquea el proceso de auto-unseal. Es esencial definir políticas de ciclo de vida y rotación de claves con alertas prematuras, así como auditar los permisos asignados al principal de desbloqueo.
- **Caída de rendimiento por saturación del backend de almacenamiento.** Elevados volúmenes de lecturas/escrituras pueden sobrecargar Consul o el *Integrated Storage*, provocando latencias y errores de I/O. La respuesta pasa por dimensionar adecuadamente el backend, y si es posible implementar una caché de lectura local para reducir la carga sobre el almacenamiento central.
- **Revocación o modificación accidental de políticas de acceso.** Cambios erróneos en las *policies* pueden denegar permisos legítimos, generando respuestas 403 Forbidden. La práctica recomendada consiste en versionar las políticas en un entorno o repositorio de control de versiones como Git, automatizar su despliegue con pipelines CI/CD y ejecutar pruebas de acceso antes de aplicar estos cambios en un entorno productivo.

4.6 Resumen del capítulo

En este capítulo se ha detallado el diseño e implementación de una solución de autenticación, autorización y gestión segura de credenciales tras haber llevado a cabo una fase de análisis e investigación previa de las diferentes opciones disponibles, donde se ha llegado decidido hacer uso de Keycloak como proveedor de identidad y Vault como gestor de secretos. A modo de conocimiento, se ha configurado Vault para autenticar usuarios mediante el protocolo OIDC con Keycloak, validando así la integración entre ambos servicios. Posteriormente, se ha aplicado esta arquitectura a una aplicación cliente de ejemplo, la cual fue configurada para utilizar autenticación federada mediante OIDC, protegiendo así sus rutas sin necesidad de modificar su lógica interna.

Todos los servicios de estos escenarios han sido orquestados con Docker Compose, incluyendo los servicios de Keycloak, Vault y PostgreSQL, siendo este último utilizado como base de datos para almacenar la configuración y los datos persistentes de Keycloak. Además, se desarrollaron scripts personalizados y un *init-container* para Vault para automatizar la inicialización y puesta en marcha de los servicios.

También se ha utilizado Terraform como herramienta de infraestructura como código para crear y gestionar de forma declarativa y automatizada los recursos necesarios para configurar Keycloak y Vault, como *realms*, usuarios, clientes OIDC y políticas de acceso, facilitando su modularización, escalabilidad y mantenimiento.

Como punto final se exponen los errores más habituales en Keycloak como son las caídas de la base de datos, desincronización de sesiones, certificados TLS caducados, o fallos en LDAP/AD, así como también los de Vault, como son nodos sellados, claves de *auto-unseal* caducadas, sobrecarga de I/O y errores en las políticas. Para cada caso se proponen soluciones que garantizan una rápida recuperación y mantienen la disponibilidad y seguridad de los servicios y con ello la operatividad y funcionamiento correcto de las distintas aplicaciones que los utilicen.

Capítulo 05: Conclusiones y trabajo futuro

En el siguiente capítulo se elaborarán las conclusiones obtenidas tras la investigación e implementación realizadas, así como posibles oportunidades de mejora para seguir evolucionando y trabajando sobre este proyecto.

5.1 Conclusiones

En el presente Trabajo Fin de Máster, la aportación primordial radica en la externalización de los procesos de autenticación y autorización mediante la adopción de un sistema especializado. La centralización de la gestión de identidades en Keycloak permite a las aplicaciones cliente prescindir de implementar sus propios mecanismos de seguridad, lo que reduce la complejidad del desarrollo y homogeneiza el control de accesos. De forma análoga, Vault suministra un marco sólido para la administración dinámica de secretos (contraseñas, certificados y tokens) y sus políticas definidas. La clara distinción entre la gestión de identidades gobernada por Keycloak y la gestión de secretos gestionados por Vault, unida a la aplicación estricta del principio de mínimos privilegios, refuerza de manera sustancial la seguridad y facilita la integración de nuevas aplicaciones que carecen o quieren externalizar la autenticación y autorización.

Por otra parte, la descripción de la infraestructura como código, mediante la combinación de Docker Compose para la orquestación local y Terraform para el aprovisionamiento en entornos de nube, asegura la reproducibilidad y la homogeneidad de los entornos de desarrollo, prueba y producción. Este enfoque posibilita la automatización integral de los despliegues, que abarca desde la base de datos hasta la configuración de Keycloak y Vault, traduciéndose en una notable reducción de la carga operativa y de los errores manuales asociados.

Por último, la contenedorización garantiza portabilidad y escalabilidad: el sistema se puede replicar en cualquier proveedor de nube o en máquinas locales, y la parametrización modular de Terraform permite incorporar nuevos ámbitos de uso o servicios sin rehacer la base del proyecto.

Sin embargo, este enfoque también conlleva una serie de consecuencias. En primer lugar, la curva de aprendizaje para los usuarios que mantienen y hacen uso de estas tecnologías es elevada, pues es necesario tener conocimiento técnico y funcional de Docker, Keycloak, Vault y Terraform, lo que implica un esfuerzo inicial considerable. Además, para aplicaciones sencillas, desplegar todo el stack (base de datos, servidor de identidades, almacén de secretos y contenedor de inicialización) puede resultar excesivo y generar complejidad adicional. El mantenimiento de versiones y la compatibilidad entre las distintas herramientas haría necesario realizar revisiones periódicas de los scripts y de la configuración, ya que cambios en las APIs o en los formatos de datos pueden dejar corruptos flujos ya establecidos. Por último, aunque Docker aísla los servicios, siguen existiendo posibles conflictos de puertos, permisos de carpetas que requieren ajustes específicos, y poner en marcha soluciones de alta disponibilidad para garantizar cero interrupciones o recuperación inmediata del servicio.

5.2 Trabajo futuro

Para garantizar la alta disponibilidad y el escalado horizontal, se podría desplegar Keycloak y Vault en modo clúster utilizando un orquestador como Kubernetes. Esta aproximación permitiría que las instancias de Keycloak compartieran un backend de base de datos PostgreSQL configurado en modo maestro-esclavo, mientras que Vault podría emplear un almacenamiento distribuido como Consul o Integrated Storage para replicar su estado interno. De este modo, se aseguraría la continuidad del servicio ante fallos puntuales y se repartiría la carga de peticiones entre las distintas réplicas, mejorando tanto la resiliencia como el rendimiento del sistema.

Desde una perspectiva funcional, esta configuración no solo mejoraría la infraestructura, sino que aportaría beneficios claros a nivel de operación y mantenimiento: un sistema más robusto frente a caídas, menor latencia de respuesta en momentos de alta demanda y una gestión más ágil y automatizada de identidades y secretos para entornos de producción reales. Además, integrar ambos servicios en un clúster Kubernetes permitiría sacar partido del mecanismo de autenticación nativo de Vault para Kubernetes (auth/kubernetes), así como del Secret Injector en modo sidecar. Esto permitiría que las aplicaciones desplegadas en el clúster pudieran obtener credenciales dinámicas directamente desde Vault a través de sus propias Service Accounts.

Más allá de los flujos básicos de autenticación mediante OIDC, se propone habilitar en Vault motores de generación de secretos dinámicos y PKI (infraestructura de clave pública). A nivel funcional, esto permitiría, por ejemplo, generar usuarios de base de datos con credenciales efímeras, emitir tokens de acceso temporales para plataformas en la nube como AWS, GCP o Azure, o emitir certificados TLS válidos solo durante cortos periodos de tiempo para los distintos microservicios de la arquitectura. Estas capacidades refuerzan la seguridad del sistema, ya que limitan de forma estricta el tiempo de vida de cada credencial y minimizan el impacto de una posible filtración.

En términos de monitorización, trazas y alertas, resultaría muy útil integrar Prometheus y Grafana para capturar métricas clave de Keycloak (tasa de autenticaciones, latencia de login) y de Vault (número de peticiones de autenticación, tiempos de respuesta, etc.). Asimismo, centralizar logs con ELK o Loki y, configurando alertas en caso de errores de autenticación o picos inusuales de tráfico cuyo funcionamiento de las aplicaciones alimentadas por estos servicios pudieran verse afectadas.

5.3 Resumen del capítulo

En este capítulo se han sintetizado los aprendizajes y resultados clave del presente TFM, poniendo de relieve cómo la separación de responsabilidades entre Keycloak y Vault refuerza la seguridad de las aplicaciones que delegan e implementan de forma externa los procesos de autenticación y autorización mediante el principio de privilegios mínimos y la rotación automática de credenciales. La adopción de Docker Compose y Terraform para describir la infraestructura como código garantiza despliegues reproducibles, coherentes y auditablemente trazables. También se han identificado y planteado propuestas de evolución orientadas a dotar al sistema de tolerancia a fallos, escalado en Kubernetes, generación de secretos dinámicos y certificados y monitorización de los servicios donde con estas conclusiones y líneas de trabajo futuro se pretenda seguir las buenas prácticas de DevSecOps.

Capítulo 06: Bibliografía

[1] F. J. Corbató y V. A. Vyssotsky, "Introduction and overview of the Multics system", Proceedings of the Fall Joint Computer Conference, pp. 185-196, 1965.

[2] T. Hunt, "Passwords are broken, and nobody seems to care", IEEE Security & Privacy, vol. 15, no. 5, pp. 70-74, 2017.

[3] Verizon, "2016 Data Breach Investigations Report", Verizon Enterprise, 2016.

[4] A. O. Udo, "Phishing: A growing challenge for internet users", Journal of Information Security and Applications, vol. 21, pp. 1-9, 2015.

[5] W. Yeong, T. Howes, y S. Kille, "Lightweight Directory Access Protocol", RFC 1777, 1995.

[6] S. Furnell, "Tokens and Beyond: Multi-Factor Authentication for the Masses", *Computer Fraud & Security*, vol. 2005, no. 8, pp. 12-16, 2005.

[7] National Institute of Standards and Technology, "Electronic Authentication Guideline", NIST Special Publication 800-63-2, 2013.

[8] P. Madsen, "A Guide to Understanding SAML", *Sun Microsystems*, 2003.

[9] OASIS, "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard, 2005.

[10] D. R. Kuhn, "An overview of single sign-on technology", *Proceedings of the 24th NIST-NCSC National Information Systems Security Conference*, pp. 27-31, 2001.

[11] A. K. Das, N. Kumar, y J. J. P. C. Rodrigues, "Authentication protocols and schemes for smart mobile devices: security vulnerabilities, attacks, countermeasures, and open issues", *Telecommunication Systems*, vol. 67, no. 2, pp. 249-274, 2018.

[12] E. Hammer-Lahav, "The OAuth 1.0 Protocol", RFC 5849, 2010.

[13] D. Hardt, "The OAuth 2.0 Authorization Framework", RFC 6749, 2012.

[14] N. Sakimura et al., "OpenID Connect Core 1.0", The OpenID Foundation, 2014.

[15] M. Jones, "JSON Web Token (JWT)", RFC 7519, 2015.

[16] Gartner, "Magic Quadrant for Identity as a Service, Worldwide", Gartner Research, 2019.

[17] Red Hat, "Keycloak Documentation". Disponible en https://www.keycloak.org/documentation. Consultado el 10 de junio de 2025.

[18] Red Hat, "Server Developer Guide: Service Provider Interfaces (SPI)", Keycloak Documentation, 2023. Disponible en https://www.keycloak.org/docs/latest/server_development/#_providers. Consultado el 10 de junio de 2025.

[19] HashiCorp, "Vault: Secrets Management", Vault by HashiCorp, 2023. Disponible en https://www.vaultproject.io/docs. Consultado el 10 de junio de 2025.

[20] K. Jackson, "Managing Secrets in Microservices with Vault", ACM Queue, vol. 18, no. 5, pp. 34–47, 2020.

[21] M. Merkow y J. Breithaupt, "Virtualization and Containers: A Guide to Software Isolation and Security", 2.ª ed. Boston, MA, EE. UU.: Pearson, 2021.

[22] Docker Inc., "Docker Overview", Docker Documentation, 2023. Disponible en https://docs.docker.com/get-started/overview. Consultado el 10 de junio de 2025.

[23] Docker Inc., "Best practices for writing Dockerfiles", Docker Documentation, 2023. Disponible en https://docs.docker.com/develop/develop-images/dockerfile_best-practices. Consultado el 10 de junio de 2025.

[24] The Kubernetes Authors, "Kubernetes Documentation", Kubernetes.io, 2023. Disponible en https://kubernetes.io/docs/home. Consultado el 10 de junio de 2025.

[25] The Kubernetes Authors, "Pod Security Standards", Kubernetes Documentation, 2023. Disponible en https://kubernetes.io/docs/concepts/security/pod-security-standards. Consultado el 10 de junio de 2025.

[26] G. Kim, J. Humble, P. Debois y J. Willis, "The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations", 2.ª ed. Portland, OR, EE. UU.: IT Revolution Press, 2021.

[27] N. Zubair, "DevSecOps: Integrating Security into DevOps", IEEE Software, vol. 37, no. 3, pp. 20–27, 2020.

[28] OWASP, "Testing Guide", OWASP Foundation, 2023. Disponible en https://owasp.org/www-project-web-security-testing-guide. Consultado el 10 de junio de 2025.

[29] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg y I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility", Future Generation Computer Systems, vol. 25, núm. 6, pp. 599–616, 2009.

[30] P. Mell y T. Grance, "The NIST Definition of Cloud Computing", NIST Special Publication 800-145, Gaithersburg, MD, USA, 2011.

[31] T. Erl, R. Puttini y Z. Mahmood, "Cloud Computing: Concepts, Technology & Architecture". Upper Saddle River, NJ, USA: Prentice Hall, 2013.

[32] M. Armbrust et al., "A View of Cloud Computing", Communications of the ACM, vol. 53, núm. 4, pp. 50–58, 2010.

[33] M. Cusumano, A. Gawer y D. B. Yoffie, "The Business of Platforms: Strategy in the Age of Digital Competition, Innovation, and Power". New York, NY, USA: Harper Business, 2019.

[34] K. Morris, "Infrastructure as Code: Managing Servers in the Cloud". Sebastopol, CA, USA: O'Reilly Media, 2016.

[35] P. M. Fitts, S. L. Atkinson y J. Watson, "Infrastructure as Code: Principles and Practices of Immutable Infrastructure," en Proceedings of the 2019 IEEE Conference on Software Engineering (ICSE), Montreal, Canada, 2019, pp. 123–130.

[36] M. Duffy, "Infrastructure as Code with Terraform: Provisioning Cloud Infrastructure in a Modern and Reproducible Way". Sebastopol, CA, USA: O'Reilly Media, 2020.

[37] Pulumi, "Pulumi Documentation". Disponible en https://www.pulumi.com/docs. Consultado el 10 de junio de 2025.

[38] Red Hat, "Ansible Documentation". Disponible en https://docs.ansible.com. Consultado el 10 de junio de 2025.

[39] Progress Software, "Chef Documentation". Disponible en https://docs.chef.io. Consultado el 10 de junio de 2025.

[40] Puppet, "Puppet Documentation". Disponible en https://puppet.com/docs. Consultado el 10 de junio de 2025.