UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

Stateless Authentication Microservice

Erick Andre Ñauñay Cantos

Ingeniería en Ciencias de la Computación

Trabajo de fin de carrera presentado como requisito
para la obtención del título de
INGENIERO EN CIENCIAS DE LA COMPUTACIÓN

Quito, 11 de diciembre de 2020

UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

HOJA DE CALIFICACIÓN
DE TRABAJO DE FIN DE CARRERA

Stateless Authentication Microservice

Erick Andre Ñauñay Cantos

Nombre del profesor, Título académico **Daniel Fellig, MS**

Quito, 11 de diciembre de 2020

© DERECHOS DE AUTOR

Por medio del presente documento certifico que he leído todas las Políticas y Manuales de la Universidad San Francisco de Quito USFQ, incluyendo la Política de Propiedad Intelectual USFQ, y estoy de acuerdo con su contenido, por lo que los derechos de propiedad intelectual del presente trabajo quedan sujetos a lo dispuesto en esas Políticas.

Asimismo, autorizo a la USFQ para que realice la digitalización y publicación de este trabajo en el repositorio virtual, de conformidad a lo dispuesto en la Ley Orgánica de Educación Superior del Ecuador.

Nombres y apellidos: Erick Andre Ñauñay Cantos

Código: 00136877

Cédula de identidad: 1724461957

Lugar y fecha: Quito, 06 de noviembre de 2020

ACLARACIÓN PARA PUBLICACIÓN

Nota: El presente trabajo, en su totalidad o cualquiera de sus partes, no debe ser considerado como una publicación, incluso a pesar de estar disponible sin restricciones a través de un repositorio institucional. Esta declaración se alinea con las prácticas y recomendaciones presentadas por el Committee on Publication Ethics COPE descritas por Barbour et al. (2017) Discussion document on best practice for issues around theses publishing, disponible en http://bit.ly/COPETheses.

UNPUBLISHED DOCUMENT

Note: The following capstone project is available through Universidad San Francisco de Quito USFQ institutional repository. Nonetheless, this project – in whole or in part – should not be considered a publication. This statement follows the recommendations presented by the Committee on Publication Ethics COPE described by Barbour et al. (2017) Discussion document on best practice for issues around theses publishing available on http://bit.ly/COPETheses.

RESUMEN

Desde el nacimiento de la web y los sistemas digitales la autenticación ha sido un tema de investigación y de relevancia, uno de los casos de uso más grandes de autenticación son los negocios digitales que necesitan almacenar la información de sus usuarios. En la mayoría de las aplicaciones web o móviles la lógica de autorización y autenticación se encuentra embebida en el mismo backend de las reglas del negocio, a esta solución se le conoce como monolítica y no es escalable ni segura. Dado esto, el proyecto desarrolla un microservicio independiente de autenticación, el cual mediante una REST API expone la interfaz de conexión donde se implementan los casos de uso para que cualquier solución que necesite de autenticación pueda conectarse al servicio y así disminuir el tiempo de desarrollo. En el presente documento se encuentra la documentación de la implementación del servicio. La principal característica del servicio es que no guarda un estado del cliente, es decir que las sesiones son manejadas solo por el cliente web mas no por el servidor de autenticación. El manejo de sesiones usa JWT como token de autorización a los recursos. Además, el proyecto fue diseñado e implementado siguiendo estándares de calidad mediante los patrones de diseño orientado al domino y el patrón de desarrollo orientado a pruebas. Finalmente, el proyecto cuenta con una aplicación cliente y una interfaz gráfica de la REST API que permiten visualizar el flujo del uso del servicio además de probar sus puntos de conexión.

Palabras clave: autenticación, autorización, microservicio, JWT, API, REST, recursos, puntos de conexión, servidor, cliente, usuario.

ABSTRACT

Since the birth of the web and digital systems, authentication has been a topic of research and relevance, one of the largest use cases of authentication is digital businesses that need to store the information of their users. In most web or mobile applications, the authorization and authentication logic is embedded in the same backend of the business rules, this solution is known as monolithic and it is not scalable or secure. Given this, the project develops an independent authentication microservice, which through a REST API exposes the authorization interface part where the use cases are implemented so that any solution that needs authentication can connect to the service and thus reduce development time. This document contains the documentation for the implementation of the service. The main characteristic of the service is that it does not save a state of the client, that is, the sessions are handled only by the web client but not by the authentication server. Session handling uses JWT as authorization token to resources. Furthermore, the project was designed and implemented following quality standards through the domain-oriented design patterns and the test-oriented development pattern. Finally, the project has a client application and a graphical interface of the REST API that allow to visualize the flow of the use of the service in addition to testing its endpoints.

Key words: authentication, authorization, JWT, API, REST, resources, endpoints, server, client, user.

TABLA DE CONTENIDOS

Introducción **10**
Objetivos generales y específicos 11
Objetivos generales 11
Objetivos específicos 12
Desarrollo del tema **13**
Lógica de negocio 13
Estructura de la aplicación 16
Tecnologías utilizadas para el desarrollo del servicio 17
Golang 18
SQLite 19
Domain-driven design (DDD) 19
Test-driven development (TDD). 19
Arquitectura de la aplicación 20
Capa de infraestructura 23
Servicio de email 24
Servicio de persistencia de información (Base de Datos) 25
Capa de negocio 27
Capa de aplicación 31
Pruebas y resultados de la funcionalidad 34
Cobertura de pruebas 34
Resultados 35
Registro de usuario 36
Inicio de sesión 37
Validación del token 37
Cambio y reinicio de contraseña 38
Dificultades encontradas y sus soluciones 40
Mejoras conocidas 40
Trabajo futuro 41

Conclusiones 42

Referencias bibliográficas 43

ANEXO A: DIAGRAMA DE LOS MÓDULOS PRINCIPALES...............................45

ANEXO B: DIAGRAMA DE FLUJO DEL JWT...46

ANEXO C: DIAGRAMA DE FLUJO DEL CASO DE USO: REGISTRO DE USUARIO.47

ANEXO D: DIAGRAMA DE FLUJO DEL CASO DE USO: INICIO DE SESIÓN 47

ANEXO E: DIAGRAMA DE FLUJO DEL CASO DE USO: CAMBIO DE CONTRASEÑA 48

ANEXO F: DIAGRAMA DE FLUJO DEL CASO DE USO: REINICIO DE CONTRASEÑA...49

ÍNDICE DE FIGURAS

Figura 1. Diagrama de casos de uso 14

Figura 2. Ciclo de acción del JWT 15

Figura 3. Estructura lógica y de directorios del proyecto 17

Figura 4. Declaración de las dependencias del servicio. 18

Figura 5. Directorio donde se implementan las entidades de la capa de negocio 20

Figura 6. Archivo de configuración .env.example 22

Figura 7. Archivo Make de automatización 23

Figura 8. Interfaz del servicio de email definida en la capa de negocio 24

Figura 9. Estructura de la implementación del servicio de proveedores de email 25

Figura 10. Interfaz del servicio de base de datos definida en la capa de negocio 26

Figura 11. Estructura de la base de datos 26

Figura 12. Estructura de la implementación del servicio base de datos 27

Figura 13. Estructura de directorios de la capa de negocio 28

Figura 14. Interfaz que define al Authenticator 29

Figura 15. Interfaz que define al Password Manager 30

Figura 16. Interfaz que define al User Manager 30

Figura 17. Estructura de directorios de la capa de aplicación. 32

Figura 18. Documentación de la API en swagger, rutas del Authenticator 33

Figura 19. Documentación de la API en swagger, rutas del Healthcheck y User Manager 33

Figura 20. Pruebas de integración de los Authenticator endpoints de la REST API 35

Figura 21. Resultado registro de usuario en la aplicación cliente 36

Figura 22. Resultado inicio de sesión en la aplicación cliente 37

Figura 23. Resultado validación token en la aplicación cliente 38

Figura 24. Resultado cambio de contraseña en la aplicación cliente 39

Figura 25. Resultado inicio del reinicio de la contraseña en la aplicación cliente 39

Figura 26. Resultado final del reinicio de la contraseña en la aplicación cliente 39

INTRODUCCIÓN

La autenticación es el proceso de verificación de la identidad de un usuario, persona o dispositivo. Este concepto ha formado parte desde el comienzo de la computación, pero sin duda con el nacimiento de la web, su evolución a través del tiempo y el desarrollo de los negocios digitales, la idea de una identidad que persiste en el tiempo con información se ha vuelto un caso de uso intrínseco en la mayoría de las aplicaciones web y móviles (Richardson, 2008). El proceso de acceder a esta información y poder manipularla se le llama autorización. Los procesos de autenticación y autorización son ampliamente implementados en la seguridad y/o lógica de negocio de la mayoría de los sistemas web/móviles.

La mayoría de las soluciones de software en la web que requieren lógica de autenticación y autorización las implementan en el mismo código base de la solución general, esto se le conoce como una arquitectura monolítica (Thönes, 2015). Este proyecto construye una solución de autenticación como un servicio independiente o microservicio. Un microservicio es una unidad independiente de código fuente entre otras unidades, permitiendo la comunicación entre procesos, estas unidades son parte fundamental de la arquitectura de microservicios (Thönes, 2015). Un servicio de autenticación como microservicio provee beneficios en rendimiento, es una solución escalable, mantenible y confiable siempre que el código fuente siga estándares de calidad.

El mecanismo de autenticación más implementado en los negocios digitales permite a los usuarios ingresar sus credenciales, se genera un identificador único para el usuario conocido como sessionId, el servidor de autenticación lo almacena y devuelve al cliente web, a este proceso se lo conoce como stateful (Boyd, 2016). El proyecto es la construcción de un servicio de autenticación que usa una base de datos relacional para almacenar la información de los

usuarios como: nombres, apellidos, email y contraseña; el servicio genera un token único cada vez que un usuario inicia sesión, el token no es almacenado en el servidor de autenticación, esta es una estrategia stateless (Boyd, 2016). El proceso stateless hace uso de JWT (JSON Web Tokens) como método de transmisión de información de forma segura entre entidades. El JWT almacena cierta información del usuario de forma encriptada y dado que es la aplicación cliente quien almacena el token en lugar del servidor, el servidor de autenticación se encarga de la creación del JWT y de la verificación de la validez de un token provisto por el cliente.

La solución se diseñó siguiendo los conceptos: Diseño basado en el Dominio (Domain-driven design) y Diseño basado en pruebas (Test-driven design), además el código fuente cuenta con pruebas unitarias y de integración en un script. El servicio de autenticación está estructurado en tres capas: negocio, infraestructura y aplicación. Finalmente, la funcionalidad del proyecto se encuentra expuesta mediante una RESTful API.

Objetivos generales y específicos

Objetivo general.

- Implementar un servicio que permita autenticar y autorizar usuarios como un servicio aislado e independiente de cualquier lógica de negocio que requiera la solución.

Objetivos específicos.

- Diseñar la estructura y arquitectura del servicio mediante la implementación de los patrones de diseño adecuados que permitan al código ser escalable, mantenible y confiable.
- Exponer los casos de uso del servicio mediante una API REST la cual deberá estar documentada y contener pruebas de integración.

DESARROLLO DEL TEMA

Lógica de negocio

La solución implementada es un servicio de autenticación y autorización que puede formar parte del backend de cualquier aplicación como un microservicio. Los microservicios forman parte de un estilo de organizar el código fuente de una aplicación en elementos más pequeños e independientes entre sí, sin embargo, estos pueden comunicarse para compartir información o lanzar eventos (Thönes, 2015). La solución fue desarrollada como un servicio independiente el cual no forma parte de una arquitectura monolítica, donde todo se encuentra en el mismo código fuente. En cambio, esta solución es más modular, el servicio únicamente se encargará del proceso de autenticación de forma independiente sin que afecte el modelo y lógica de negocio externa al microservicio.

El servicio de autenticación se encuentra estructurado en 3 capas principales: infraestructura, negocio y aplicación. En la capa de infraestructura se encuentra la base de datos y el servicio de email, en la capa de negocio se definen las entidades y los casos de uso, finalmente la capa de aplicación expone los casos de uso ya previamente definidos al público mediante una API REST.

El microservicio usa las especificaciones RESTful para exponer los casos de uso para que cualquier solución pueda conectarse y hacer uso de este; el servicio está pensado para ser usado por cualquier proyecto sin importar su arquitectura ni diseño. La figura 1 muestra el diagrama de casos de uso del servicio de autenticación. Como se evidencia en la figura 1 el microservicio de autenticación expone mediante su REST API seis casos de uso principales:

1. *Creación de cuenta*: Permite la creación y registro de un usuario en la base de datos, por otro lado, si un usuario crea una cuenta es necesario que para acceder por primera vez use el caso de uso de inicio de sesión.
2. *Inicio de sesión*: El inicio de sesión mediante el ingreso de credenciales como: email y contraseña permite autorizar a un usuario para acceder a su información.
3. *Validación del token*: Validar el token de un usuario para verificar que es quien dice ser.
4. *Reinicio de contraseña*: Permite restaurar la contraseña de un usuario en 2 pasos, el primero inicia el proceso de reinicio enviando un código al email del usuario y el segundo permite cambiar la contraseña usando el código proporcionado.
5. *Cambio de contraseña*: Un usuario puede cambiar su contraseña en cualquier instante de tiempo.
6. *Operaciones CRUD*: El servicio de back-end que se conecta al servicio debe poder realizar operaciones de lectura, escritura, actualización y eliminación de información.

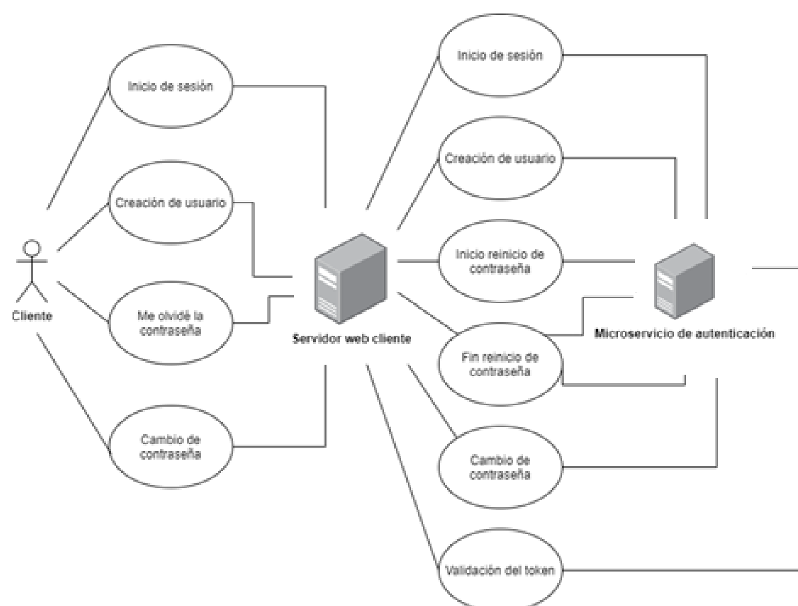


Figura 1. Diagrama de casos de uso

El microservicio implementa la estrategia stateless, la cual no necesita del almacenamiento del estado del cliente para administrar las sesiones, sino genera un token al cual se le puede validar y verificar su autenticidad, este token es el Json Web Toklen (JWT).

La figura 2 muestra el flujo del JWT. El proceso inicia cuando un usuario inicia una sesión. La aplicación cliente llama al servidor de autenticación y este le devuelve el JWT solo si el usuario se autentico correctamente. Después, el JWT es almacenado por el cliente. Finalmente, cuando el usuario decide acceder a recursos privados este debe verificar su identidad mediante una llamada al servidor de autenticación para la validación del token, si este no ha sido modificado el cliente recibe el acceso a dichos recursos.

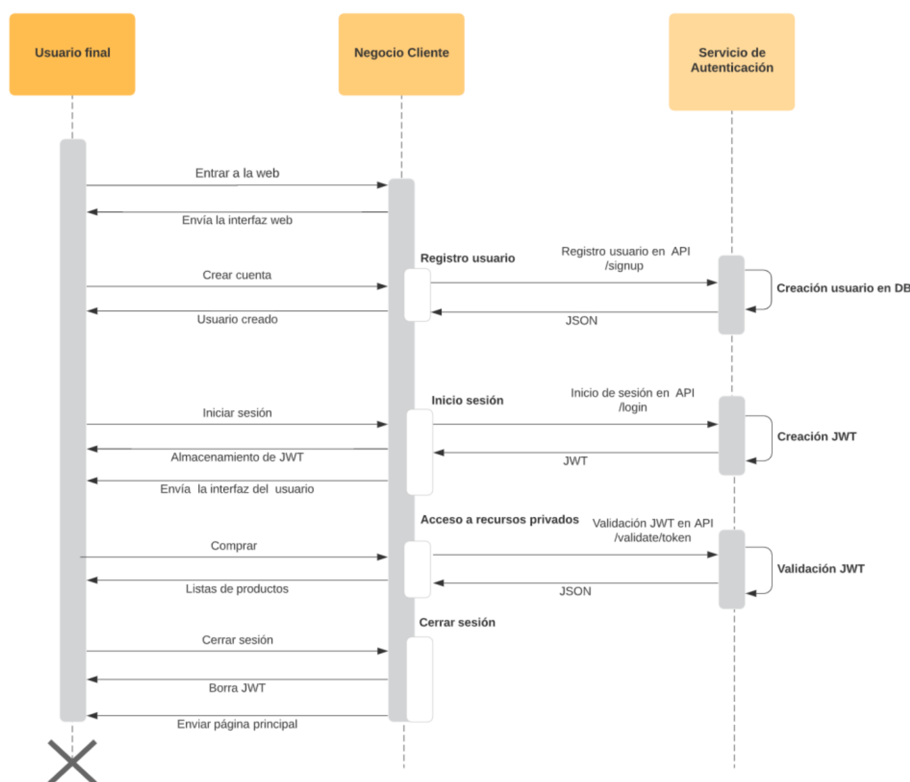


Figura 2. Ciclo de acción del JWT

Existen varias tecnologías en el mercado que brindan estos servicios, la mas grande Auth0. Auth0 es una empresa que provee un kit de herramientas de desarrollo para administrar la identidad de usuarios mediante su infraestructura. Por otro lado, Amazon Web Services también tiene una solución dentro de su plataforma AWS.

Estructura de la aplicación

Como lo explicado anteriormente, el código fuente del microservicio se encuentra estructurado en tres capas principales: aplicación, infraestructura y lógica de negocio como lo muestra la figura 3. En la figura 3 se puede evidenciar el directorio *authenticator/* que almacena las implementaciones de los casos de uso respectivos a cada entidad del modelo y lógica de negocio, en el directorio *email/* y *database/* se encuentran las implementaciones de los servicios de infraestructura de email transaccional y de sistema de persistencia de información, finalmente el directorio *api/* contiene el código que implementa la capa de exposición del servicio de autenticación, el cual expone los casos de uso mediante una REST API. Una API es una interfaz de programación de aplicaciones, define un grupo de reglas que permite a los programas hablar entre sí; *Representational State Transfer*, también conocido como REST es una estrategia que determina como la API está estructurada (Rodriguez, 2008). Los directorios *test/* y *tools/* contienen implementaciones de utilidades necesarias para las capas de desarrollo; adicionalmente, directorios como *tmp/* y *bin/* forman parte de la construcción y despliegue automatizado del fichero Makefile.

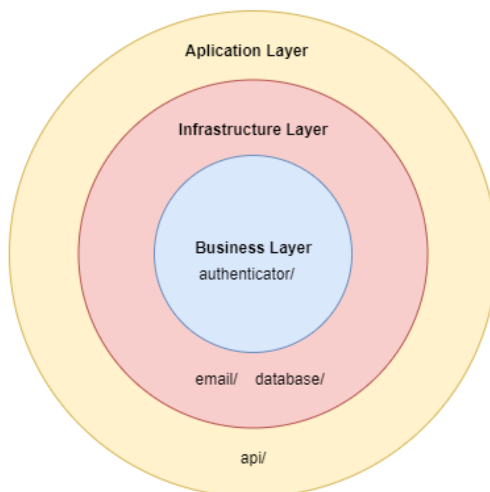


Figura 3. Estructura lógica y de directorios del proyecto

El código fuente cuenta con ficheros en segundo plano como go.sum y go.mod que son usados por el compilador de Golang para la inyección de dependencias. Adicionalmente, el proyecto cuenta con herramientas de desarrollo como: formateador y linterna, las cuales pueden ser ejecutadas por el fichero coordinador de la construcción y despliegue del servicio de autenticación, Makefile es el nombre del fichero. La herramienta de linterna usa el shell script deps.sh para ser descargada en el proyecto, además de crear un fichero que almacena la documentación de la REST API. Estas herramientas ayudan a mantener una estructura del código limpio, con un formato consistente a través de los ficheros y con menos errores de sintaxis.

Tecnologías utilizadas para el desarrollo del servicio

El proyecto se encuentra desarrollado con el lenguaje de programación Golang. Como sistema de persistencia se utilizó una base de datos relacional ligera, SQLite en este caso. El proyecto fue estructurado siguiendo los patrones de diseño Domain-Driven Design y Test-Driven Development.

Golang.

También conocido como Go, es un lenguaje de programación diseñado y mantenido por un equipo de Google. Golang es un lenguaje compilado y de tipeado estático, además tiene una sintaxis similar a la de C. Sin embargo, Go posee un colector de basura que permite tener un control seguro de memoria (Andrawos, 2017). El proyecto necesita mínimo la versión 1.11 del compilador, además se usa el framework Gonic para el desarrollo del servidor web mediante una REST API. La figura 4 muestra las versiones de las dependencias del código fuente.

```
require (
    github.com/alecthomas/template v0.0.0-20190718012654-fb15b899a751
    github.com/bixlabs/authentication v0.4.0-alpha
    github.com/caarlos0/env v3.5.0+incompatible
    github.com/cosmtrek/air v1.12.1 // indirect
    github.com/creack/pty v1.1.11 // indirect
    github.com/cucumber/godog v0.10.0
    github.com/dgrijalva/jwt-go v3.2.0+incompatible
    github.com/fatih/color v1.9.0 // indirect
    github.com/franela/goblin v0.0.0-20180407132755-cd5d08fb4ede
    github.com/gin-gonic/gin v1.6.3
    github.com/go-chi/chi v4.0.2+incompatible // indirect
    github.com/go-openapi/spec v0.19.12 // indirect
    github.com/go-openapi/swag v0.19.11 // indirect
    github.com/google/uuid v1.1.2
    github.com/imdario/mergo v0.3.9 // indirect      You, 2 months ago • feature: auth use ca
    github.com/jinzhu/gorm v1.9.16
    github.com/joho/godotenv v1.3.0
    github.com/kr/pty v1.1.5 // indirect
    github.com/mailgun/mailgun-go/v3 v3.6.4
    github.com/mailru/easyjson v0.7.6 // indirect
    github.com/mattn/go-colorable v0.1.7 // indirect
    github.com/mattn/go-sqlite3 v2.0.3+incompatible
    github.com/mitchellh/go-homedir v1.1.0
    github.com/mitchellh/mapstructure v1.2.2 // indirect
    github.com/onsi/gomega v1.10.2
    github.com/pelletier/go-toml v1.8.0 // indirect
    github.com/pkg/errors v0.9.1
    github.com/satori/go.uuid v1.2.0 // indirect
    github.com/sendgrid/rest v2.4.1+incompatible // indirect
    github.com/sendgrid/sendgrid-go v3.6.3+incompatible
    github.com/sethvargo/go-password v0.2.0
    github.com/sirupsen/logrus v1.6.0
    github.com/spf13/afero v1.2.2 // indirect
    github.com/spf13/cobra v0.0.7
    github.com/spf13/jwalterweatherman v1.1.0 // indirect
    github.com/spf13/viper v1.7.1
    github.com/stretchr/objx v0.2.0 // indirect
    github.com/swaggo/gin-swagger v1.2.0
    github.com/swaggo/swag v1.6.9
    github.com/urfave/cli/v2 v2.3.0 // indirect
    golang.org/x/crypto v0.0.0-20200622213623-75b288015ac9
    golang.org/x/net v0.0.0-20201031054903-ff519b6c9102 // indirect
    golang.org/x/text v0.3.4 // indirect
    golang.org/x/tools v0.0.0-20201105220310-78b158585360 // indirect
    gopkg.in/alexcesaro/quotedprintable.v3 v3.0.0-20150716171945-2caba252f4dc // indirect
    gopkg.in/gomail.v2 v2.0.0-20160411212932-81ebce5c23df
)
```

Figura 4. Declaración de las dependencias del servicio.

SQLite.

SQLite es una librería escrita en C, la cual implementa un motor de base de datos SQL. Este motor es rápido, pequeño, confiable y completo. Este motor es el más usado en el mundo debido a su versatilidad y poco consumo de recursos (Owens, 2010). El proyecto usa GORM, que es una librería Object-Relational Mapping (ORM) de Golang. Esta librería permite el diseñar, acceder y consultar al sistema de persistencia mediante su técnica de mapeo relacional de objetos programáticos a entidades de la base de datos.

Domain-Driven Design (DDD).

DDD es una técnica de diseño de software el cual se basa en conectar profundamente la solución con el dominio y reglas del negocio (Vernon, 2013). El proyecto cuenta con las entidades Authenticator, Password Manager y User Manager que representan entidades del modelo de dominio, las cuales se encuentran definidas en la capa más profunda del proyecto, la capa de negocio.

Test-Driven Development (TDD).

TDD es una técnica de desarrollo de software, la cual especifica que siempre se debe empezar diseñando y desarrollando las pruebas por cada pequeña funcionalidad de la aplicación que se construya. Si la prueba falla se escribe más código de la funcionalidad y se vuelve a ejecutar las pruebas. Este proceso se continúa hasta que todas las pruebas pasen con éxito. Esta técnica no garantiza un código 100% libre de bugs, pero sí provee de seguridad y confiabilidad al proyecto (Janzen, 2005). El proyecto fue desarrollado usando esta técnica desde su comienzo, los ficheros que contiene pruebas tienen una nomenclatura especial, el nombre del fichero debe terminar en "_test" esto le permite al lenguaje automatizar la ejecución de estas. En la figura 5 se

muestra como cada implementación de las entidades de la capa de negocio tienen su fichero de pruebas unitarias.

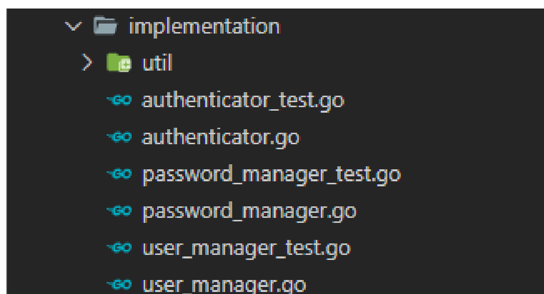


Figura 5. Directorio donde se implementan las entidades de la capa de negocio

Arquitectura de la aplicación

El servicio fue construido siguiendo la metodología 'The twelve-factor app' la cual tiene 12 factores primordiales al desarrollar software como un servicio (Wiggins, 2017). El código fuente de la aplicación se encuentra almacenado en Github, un sistema de control de versiones; existe un único repositorio del cual el código es clonado para desarrollar en un entorno local o para llevar la aplicación a un entorno de producción, a esta última tarea se la conoce como despliegue de aplicaciones. Las dependencias son declaradas explícitamente y son aisladas del ambiente. La configuración del sistema se encuentra en el ambiente de ejecución, esto se realiza mediante un archivo .env donde se encuentran todas las variables de configuración del servicio como se muestra en la figura 6. Se trata a los servicios de soporte como recursos agregados. La solución usa un servicio de email transaccional para enviar un código de 4 dígitos para autorizar el restablecimiento de contraseña el cual es independiente de la lógica principal. Las etapas de construcción, despliegue y ejecución se encuentran divididas en un fichero de automatización, el

proyecto cuenta con un archivo Make que contiene scripts para cada etapa de construcción del proyecto como se muestra en la figura 7.

Como muestra el anexo A, la aplicación cuenta con dos módulos principales que se encargan de dar soporte a los casos de uso definidos en la lógica de negocio, estos módulos son: email y base de datos. El modulo de email es usado únicamente cuando el usuario olvido su contraseña, por otro lado, la base de datos es el pilar del servicio ya que es la que almacena la información de los usuarios y maneja el flujo de información.

La solución fue elaborada usando técnicas patrones de diseño basados en el dominio y desarrollo de pruebas, también conocidos como DDD (Domain-driven design) y TDD (Testing-driven development). DDD especifica que la estructura y el lenguaje del código debe estar relacionado directamente al dominio del negocio; TDD es el proceso de un ciclo de desarrollo basado en las pruebas, donde antes de implementar la solución se empieza con las pruebas de esta (Janzen, 2005), esto beneficia en mejor calidad de código debido a que se capturan errores y excepciones lógicas en etapas desarrollo, esto no quiere decir que la posibilidad de errores lógicos en cualquier entorno sea cero. Finalmente, la estructura, organización de ficheros y directorios del proyecto están definidas por el patrón de diseño *Arquitectura Limpia* introducido por Robert Cecil Martin, el cual divide un proyecto en 4 capas principales: entidades, casos de uso, controladores e interfaces externas (Martin, 2018).

```
.env.example
...
1   ENV=
2   AUTH_SERVER_PORT=9000
3   AUTH_SERVER_TOKEN_EXPIRATION=3600
4   AUTH_SERVER_SECRET=
5
6   # reset password
7   AUTH_SERVER_RESET_PASSWORD_MAX=99999
8   AUTH_SERVER_RESET_PASSWORD_MIN=10000
9
10  # database
11  AUTH_SERVER_DATABASE_NAME=sqlite.s3db
12  AUTH_SERVER_DATABASE_USER=admin
13  AUTH_SERVER_DATABASE_PASSWORD=admin
14  AUTH_SERVER_DATABASE_SALT=salted
15
16  AUTH_SERVER_APP_ENV=dev
17
18  # email
19  AUTH_SERVER_EMAIL_PROVIDER=
20  AUTH_SERVER_EMAIL_FROM=
21  AUTH_SERVER_EMAIL_FROM_NAME=
22  AUTH_SERVER_EMAIL_TEMPLATE_PATH=
23
24  # mailgun
25  AUTH_SERVER_MAILGUN_DOMAIN=
26  AUTH_SERVER_MAILGUN_API_KEY=
27
28  # SMTP
29  AUTH_SERVER_SMTP_HOST=
30  AUTH_SERVER_SMTP_PORT=
31  AUTH_SERVER_SMTP_USERNAME=
32  AUTH_SERVER_SMTP_PASSWORD=
33
34  # sendgrid
35  AUTH_SERVER_SENDGRID_API_KEY=
```

Figura 6. Archivo de configuración .env.example

```
Makefile
You, seconds ago | 1 author (You)
all: deps lint

.PHONY: test clean format lint coverage coverage-html build

deps:
	./deps.sh

test:
	go test -v ./...

coverage:
	go test -covermode=count -coverprofile=coverage.out ./...

coverage-html:
	make coverage && go tool cover -html=coverage.out

format:
	go vet ./... && go fmt ./...

build: build-linux

build-linux: format api-docs
	go build --tags "sqlite_userauth" -o ./tmp/authenticator-server-linux ./api/main.go

clean:
	rm -r -f ./tmp

lint:
	./bin/golangci-lint run --enable-all \
		-D goimports \
		-D godox \
		-D wsl \
		-D godot \
		-D goerr113 \
		--timeout 2m0s

run:
	make api-docs && make format && go run api/main.go

api-docs:
	$(GOPATH)/bin/swag init --generalInfo  ./api/main.go --output ./api/docs

ci:
	make all build-linux
```

Figura 7. Archivo Make de automatización

Capa de infraestructura.

Infraestructura es todo servicio que no pertenezca al dominio principal de la lógica de negocio, en este caso, la capa de infraestructura del proyecto está compuesta por dos servicios: email y base de datos. Debido a que la capa de infraestructura está a un más alto nivel que la capa de negocio, la cual hace uso de estos servicios, es la capa de negocio quien define el contrato / interfaz de comunicación entre capas, y es en infraestructura donde estos servicios implementan las interfaces previamente definidas.

Servicio de email.

El servicio de email es usado únicamente en el proceso de petición de reinicio de contraseña como se puede observar en la figura 8, además de abstraer el concepto de proveedor para dar mayor versatilidad a la solución en ambientes diferentes. Los diferentes proveedores son implementados en la capa de infraestructura; sin embargo, la entidad Sender es implementada en la capa de negocio donde los casos de uso son definidos.

El servicio de email tiene dos interfaces principales: Sender y Provider. En la capa de infraestructura solo se implementa Provider, la figura 9 muestra la estructura de esta; por otro lado, la implementación de Sender se encuentra en la capa de negocio. La interfaz de Provider representa varios proveedores de servicio de email transaccional como: mailgun, sendgrid o protocolo SMTP. El servicio define cual proveedor de email a usar dependiendo de la configuración que se provea en la variable de ambiente AUTH_SERVER_EMAIL_PROVIDER en el archivo .env que configura el sistema. Finalmente, la interfaz Sender define la funcionalidad del servicio de email en la capa de aplicación, tiene un único método que envía un email con un código cuando un usuario se ha olvidado su contraseña y necesita recuperar el acceso a su información.

```
// Sender is how business layer uses emails.
type Sender interface {
    ForgotPasswordRequest(user structures.User, code string) error
}

// Provider represents different email manager platforms
type Provider interface {
    Send(emailMessage *message.Message) error
}
```

Figura 8. Interfaz del servicio de email definida en la capa de negocio

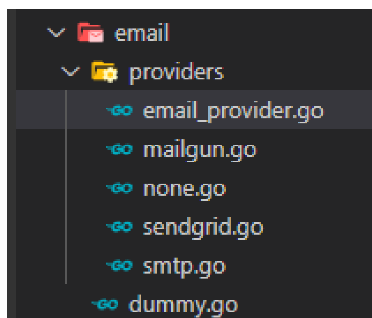


Figura 9. Estructura de la implementación del servicio de proveedores de email

Servicio de persistencia de información (Base de Datos).

El proyecto hace uso del patrón repositorio para comunicarse con la base de datos, el mismo que está definido mediante un contrato en la capa de negocio como muestra la figura 10, y es implementado en la capa de aplicación mediante un ORM con conexión al gestor de base de datos: SQLite. *Object Relational Mapping* conocido como ORM, es un modelo de programación que permite mapear las estructuras de las entidades de una base de datos relacional sobre una estructura de objetos y clases en la lógica de programación (Vernon, 2013). El servicio almacena una única tabla con la información de cada usuario. Debido a la simpleza en el diseño del modelo de la base de datos se decide usar una base de datos relacional que sea ligera y tenga buena velocidad de lectura y escritura. Esto nos permite implementar las operaciones como transacciones de forma que la concurrencia de procesos no afecte la integridad de la información y esta sea consistente y confiable.

El diseño de la interfaz sigue el patrón repositorio, este patrón define una interfaz única la cual debe encapsular la lógica necesaria para acceder a los recursos de la base de datos de manera eficiente, de forma que toda la funcionalidad de acceso a la información quede centralizada en un solo contrato, esto permite tener el código fuente mantenible y escalable (Vernon, 2013). Esta interfaz tiene los métodos CRUD para acceder a la información, estas

operaciones son: Crear (Create), Leer (Read), Actualizar (Update), Eliminar (Delete). Además de definir funcionalidad de las entidades del servicio de autenticación.

```
type Repository interface {
    Create(user structures.User) (structures.User, error)
    IsEmailAvailable(email string) (bool, error)
    GetHashedPassword(email string) (string, error)
    ChangePassword(email, newPassword string) error
    Find(email string) (structures.User, error)
    UpdateResetToken(email, resetToken string) error
    Delete(user structures.User) error
    Update(email string, user structures.User) (structures.User, error)
}
```

Figura 10. Interfaz del servicio de base de datos definida en la capa de negocio

Como muestra la figura 11, la base de datos cuenta con una única tabla de la entidad User, la cual tiene como datos: email, contraseña, nombres completos, token de reinicio y metadatos. La figura 3 muestra el fichero sqlites2.db que es aquel que contiene la base de datos de usuarios, su estructura e información.

User	
PK	ID
datetime	DeletedAt
datetime	CreatedAt
datetime	UpdatedAt
varchar	Email
varchar	Password
varchar	GivenName
varchar	SecondName
varchar	FamilyName
varchar	SecondFamilyName
varchar	ResetToken

Figura 11. Estructura de la base de datos

La capa de infraestructura que implementa la base de datos está conformada por la definición de la entidad User que será migrada a la base de datos por medio del ORM, la cual se encuentra en el directorio *model/*. El modelo es único para la base de datos por lo que en necesario crear un mapper que permita extrapolar una entidad User de la base de datos a una entidad User del modelo de negocio. Esta implementación se encuentra en *mappers/*, Finalmente, la implementación del repositorio definido en el contrato que muestra la figura 10 se encuentra en *user/*. La figura 12 muestra la estructura antes mencionada del servicio de la base de datos, la implementación en *memory/* usa una estructura de datos persistente durante el tiempo de ejecución del servicio mas no permanente. Este método fue implementado para optimizar tareas de depuración de código.

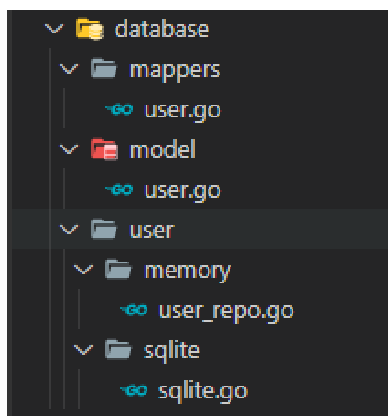


Figura 12. Estructura de la implementación del servicio base de datos

Capa de negocio.

La capa de negocio almacena las entidades del sistema, los casos de uso que implementa las reglas del negocio, en este caso del servicio de autenticación. El directorio principal es *authenticator/* como lo muestra la figura 13.

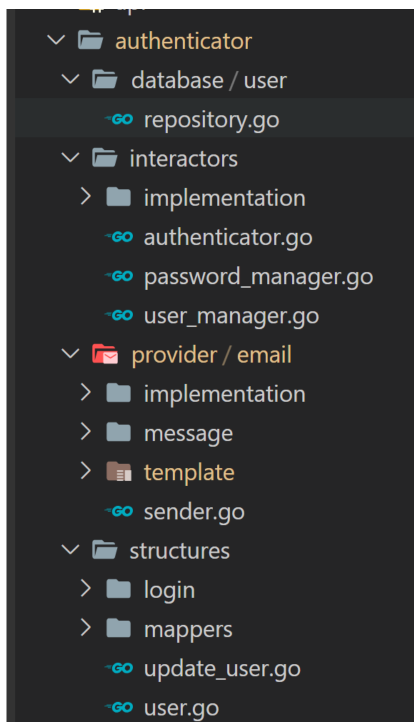


Figura 13. Estructura de directorios de la capa de negocio

La figura 13 muestra cuatro directorios internos los cuales representan capas internas, sin embargo, no están dispuestas en el orden que se muestra en la figura. La capa interna principal es la de entidades, llamada *structures/* la cual define los DTOs y la estructura de un Usuario. Un Data Transfer Object (DTO) es un objeto de transferencia de datos, es usado para transportar datos entre procesos (Vernon, 2013). Esta capa contiene de igual forma cualquier estructura de objeto que se necesite en la capa de casos de uso para su funcionamiento. Los casos de uso son administrados por tres entidades definidas en *interactors/*. Estas entidades son: Authenticator, Password Manager, User Manager. La primera define los casos de uso de registro, inicio de sesión y validación de token. El segundo administra las contraseñas de cada usuario, esta entidad define los casos de uso de reinicio y cambio de contraseña; finalmente, User Manager es aquella entidad que provee las operaciones CRUD de la base de datos.

```
7
8  type Authenticator interface {
9      Login(email, password string) (*login.Response, error)
10     Signup(user structures.User) (structures.User, error)
11     VerifyJWT(jwt string) (structures.User, error)
12 }
13
```

Figura 14. Interfaz que define al Authenticator

El autenticador como muestra la figura 14 contiene tres casos de uso principales; el de login, signup y verificación de token. La tarea de signup se encarga de ingresar un registro a la base de datos siguiendo las políticas del sistema que son: email válido y una contraseña de al menos 8 caracteres, los demás campos no son obligatorios. El registro de usuarios hace uso del repositorio de la base de datos, el mismo que para crear un usuario no guarda su contraseña en texto plano, en cambio se usa el algoritmo bcrypt el cual fue diseñado por Niels Provos y David Mazières. Bcrypt es capaz de mitigar los ataques de fuerza bruta mediante la combinación de la costosa configuración de la clave que usa el algoritmo Blowfish, con un número vasto de iteraciones que incrementan la carga de trabajo y duración de los cálculos del hash (Sriramya, 2015). Finalmente, la contraseña es almacenada en la base de datos luego de haberse realizado la operación de dispersión mediante el algoritmo bcrypt.

El anexo B muestra un diagrama de flujo del JWT token. La acción de login recibe como entrada el email y contraseña del usuario, estos datos son conocidos como las credenciales de autenticación del usuario, son validadas en el backend y luego se procede a realizar la operación de dispersión sobre la contraseña con el mismo algoritmo de encripción que se usó para el almacenamiento, si ambas cadenas de texto coinciden el sistema genera un JWT que autoriza a un cliente el acceso a su información. El JWT que se genera y retorna es un token que contiene

una carga de información del usuario registrado, además de firmar digitalmente el token de forma que se pueda verificar la integridad de la carga, a la carga también se le conoce como cuerpo. La firma que recibe el token es una cadena de caracteres que se obtienen después de realizar una operación de dispersión sobre la cabecera y el cuerpo del token. Para la firma se uso el algoritmo de dispersión HMAC SHA256.

Finalmente, el último caso de uso es el de verificación del token, este caso de uso verifica la integridad del JWT mediante la firma y el algoritmo de dispersión usado en la generación de este, el anexo B muestra el uso de estos métodos en un caso de uso real.

```
5   type PasswordManager interface {
6       ChangePassword(user structures.User, newPassword string) error
7       StartResetPassword(email string) (string, error)
8       FinishResetPassword(email string, code string, newPassword string) error
9   }
10
```

Figura 15. Interfaz que define al Password Manager

La figura 15 muestra la interfaz del Pasword Manager. La misma que define el reinicio de la contraseña en dos pasos: StartResetPassword y FinishResetPassword, el último método permite al usuario cambiar su contraseña. El servicio de email es usado para iniciar el proceso de reinicio de clave, se envía un email con un código que debe ser usado en el segundo paso para finalizar el proceso. Todos los casos de uso utilizan el contrato del repositorio el cual usa bcrypt en todas sus operaciones de campo contraseña.

```
7   type UserManager interface {
8       Create(user structures.User) (structures.User, error)
9       Delete(email string) error
10      Find(email string) (structures.User, error)
11      Update(email string, updateAttrs structures.UpdateUser) (structures.User, error)
12  }
```

Figura 16. Interfaz que define al User Manager

Finalmente, la última entidad es el User Manager, la cual provee de métodos CRUD para acceder a la información de la entidad User en el sistema de persistencia como muestra la figura 16. Los métodos de búsqueda usan el campo email que es una cadena de caracteres para las búsquedas. El método update usa una estructura propia para recibir los atributos a actualizar.

La capa de aplicación como se mencionó en la capa de infraestructura cuenta con las definiciones de los contratos de los servicios de email y base de datos como se muestra en la figura 10 en los directorios *database/user/* y *provider/sender*. La capa de infraestructura implementa el caso de uso de enviar email vía un proveedor como lo muestra la figura 9. Por otro lado, en la capa de negocio el servicio de email implementa la interfaz Sender con un solo caso de uso; enviar un email con un código numérico. Esto forma parte del primer paso de reinicio de contraseña, este evento es la petición de olvido de contraseña. Este caso de uso cuenta con una funcionalidad de poder configurar una plantilla como parte del cuerpo del email en formato HTML. La URL de la plantilla debe estar definida en el archivo de configuración con la variable AUTH_SERVER_EMAIL_TEMPLATE_PATH.

Capa de aplicación.

En la capa de aplicación se encuentra la REST API, la misma que expone los endpoints necesarios para cumplir con los casos de uso de los servicios. Esta se encuentra documentada con la tecnología swagger siguiendo el estándar de OpenAPI 3.0.

La figura 17 muestra la estructura de la capa de aplicación. Esta cuenta con tres actores principales: Authentication, Healthcheck y User Manager. Estos actores organizan los endpoints haciéndolos más descriptivos y accesibles.

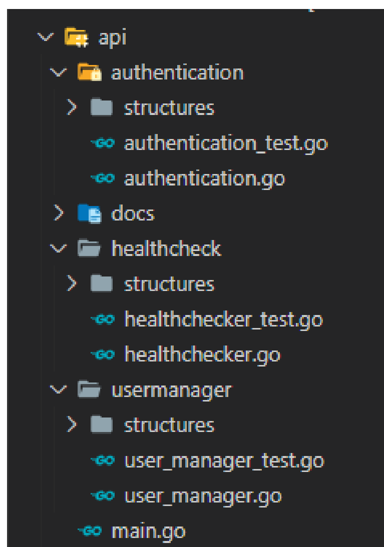


Figura 17. Estructura de directorios de la capa de aplicación.

Esta capa hace uso de un framework web ligero de golang, llamado Gin. El framework usa la tecnología de Swagger para documentar y estructurar RESTful APIs con nomenclatura JSON de forma declarativa (De, 2017). Con swagger se documentó las estructuras que recibe y retorna cada endpoint, la URL y características HTTP del punto de conexión. Toda esta documentación se encuentra en la ruta */documentation/index.html* que retorna una interfaz gráfica con un cliente HTTP que permite probar la API según lo descrito por los ficheros swagger. La capa está dividida en 3 directorios principales: *authentication/, healthcheck/* y *usermanager/,* donde cada uno representa la entidad que define e implementa un grupo de endpoints relacionados a los casos de uso de más alto nivel como lo son: autenticación, administración de usuarios y administrar el servicio. En este nivel cada endpoint que se define hace uso de al menos un caso de uso de la capa de negocio. Las figuras 17 y 18 muestran los endpoints de la REST API. Estos están divididos por los casos de uso macro.

Authentication

PUT /authentication/change-password Change password functionality

PUT /authentication/finish-reset-password Finish Reset password functionality

POST /authentication/login Login functionality

POST /authentication/signup Signup functionality

POST /authentication/start-reset-password Start reset password functionality

GET /authentication/token/validate Validates a JWT and returns the claims for it.

Figura 18. Documentación de la API en swagger, rutas del Autenticator

Healthcheck

GET /healthcheck Healthcheck functionality

User

PUT /users Update User functionality

POST /users Create one User functionality

DELETE /users Delete one User functionality

POST /users/find-by-email Find one User functionality

Figura 19. Documentación de la API en swagger, rutas del Healthcheck y User Manager

Como se puede evidenciar en las figuras 18 y 19, se muestran cómo están definidos todos los puntos de conexión con la estructura REST sobre HTTP. La etiqueta Authentication contiene todos los casos de uso de autenticación y autorización del servicio, se exponen seis rutas, las cuales son mapeadas a cada caso de uso de la entidad Authenticator en la capa de negocio. La etiqueta Healthcheck contiene una única ruta, la cual verifica que todos los servicios y entidades administradoras que usa la API estén activos y funcionales. Los componentes verificados son:

User Manager, Authenticator, proveedor de email y base de datos. Finalmente, el tag User agrupa los casos de uso CRUD que permite el acceso y manipulación de información del sistema de persistencia relacional, en este caso, SQLite.

Todos los endpoints de conexión a la API reciben y retornan objectos tipo JSON sobre HTTP. Las rutas de read y update de CRUD usan el email del usuario como identificador, mas no la llave primaria de la tabla User en la base de datos.

Pruebas y resultados de la funcionalidad

Cobertura de pruebas.

Debido al uso de TDD para el desarrollo del servicio, cada capa cuenta con pruebas unitarias y de integración. La figura 20 muestra los resultados de las pruebas de integración en la capa de aplicación. Estas pruebas hacen uso de los casos de uso de la capa de aplicación, y estos usan los servicios de infraestructura. Este tipo de pruebas son las más valiosas ya que proveen rendimiento y confiabilidad ante diferentes contextos sobre las características probadas. Las pruebas de integración verifican los casos de uso: registro, inicio de sesión, cambio de contraseña, reinicio de contraseña, generación y verificación del JWT.

```
Login rest handler
  ✓ should return 400 if email is invalid
  ✓ should return 400 if password is invalid
  ✓ should return 401 if credentials are wrong
  ✓ should return 200 if credentials are correct


4 tests complete (613 ms)

Reset password request rest handler
  ✓ should return 400 if email is invalid
  ✓ should return 500 if email doesn't exist
  ✓ should return 202 if everything goes well


7 tests complete (1099 ms)

Change Password process
  ✓ Should return 400 if email is not valid
  ✓ Should return 400 if password length is less than 8
  ✓ Should return 500 if we can't get the hashed password from db
  ✓ Should return 400 if user password is not valid
  ✓ should return 400 if newPassword is the same as the actual one
  ✓ Should return 200 if user provides the correct information


13 tests complete (1583 ms)

Sign up rest handler
  ✓ should return 400 if email is invalid
  ✓ should return 400 if password is invalid
  ✓ should return 400 if email is duplicated
  ✓ should return 201 if user is created successfully


17 tests complete (1724 ms)

Reset password rest handler
  ✓ should return 400 if email is invalid
  ✓ should return 400 if password length is not correct
  ✓ should return 400 if reset token is invalid
  ✓ should return 400 if newPassword is the same as the actual one
  ✓ should return 204 if password is changed successfully


22 tests complete (2472 ms)

Verify JWT rest handler
  ✓ should return 401 if the token is invalid
  ✓ should return 200 if the token is valid
```

Figura 20. Pruebas de integración de los Authenticator endpoints de la REST API

Resultados.

El servicio expone su funcionalidad a través de una REST API; sin embargo, para realizar una prueba de su funcionalidad se implementó un cliente web. El mismo que hace uso de todos los puntos de conexión referentes a la autenticación y autorización que expone la REST API. La interfaz gráfica del cliente web hace uso de varios modelos predefinidos de Bootstrap.

Bootstrap es un kit de herramientas de código abierto que facilita el desarrollo frontend para aplicaciones web o móvil (Krause, 2020).

La aplicación cliente se desarrolló con NodeJs como entorno en tiempo de ejecución en el servidor, ExpressJs como framework web y como motor de renderizado: PugJs. Esta aplicación usa el servicio para autenticar y autorizar a sus usuarios de forma básica. La aplicación cliente hace uso del módulo axios de NodeJs para hacer las llamadas HTTP al servicio.

Registro de usuario.

Si un usuario desea crear una cuenta debe ingresar sus datos al formulario como lo muestra la figura 21. Este formulario envía la información al endpoint del servicio mediante una petición POST a */authenticator/signup*. Si la respuesta del servicio es exitosa, entonces el usuario ha sido creado y este deberá iniciar una sesión a su nueva cuenta. En caso contrario, si existió un error, el cliente muestra el error de manera amigable, filtrado y sin exponer información del servicio. El anexo C muestra como fluye la información del cliente para este caso de uso.

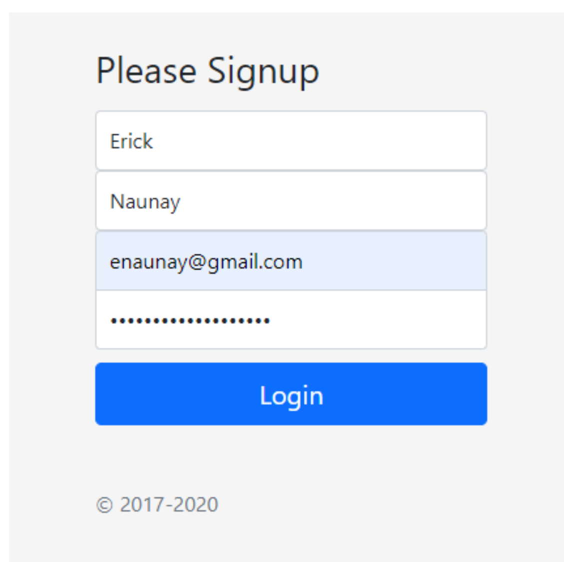


Figura 21. Resultado registro de usuario en la aplicación cliente

Inicio de sesión.

Cuando el usuario inicia una sesión, el cliente hace una petición POST a */authentication/login* con la acción del formualario de la figura 22. El servicio verifica las credenciales y retorna la sesión con el JWT token. De igual forma, si existe un error en la llamada HTTP el cliente se encarga de manejarlo. El flujo de informacion de este proceso se puede evidenciar en el anexo D.

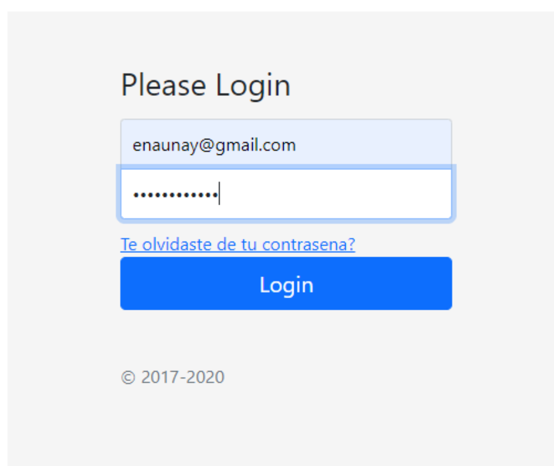


Figura 22. Resultado inicio de sesión en la aplicación cliente

Validación del token.

Una vez iniciada la sesión el usuario puede acceder a su perfil como lo muestra la figura 23. El cliente en cada recurso protegido verifica si el token almacenado en la cookie es válido y pertenece a un usuario creado. Si el token es inválido, ya sea porque fue modificado o no es un JWT se les niega el acceso a los recursos que sin autorización no son accesibles.

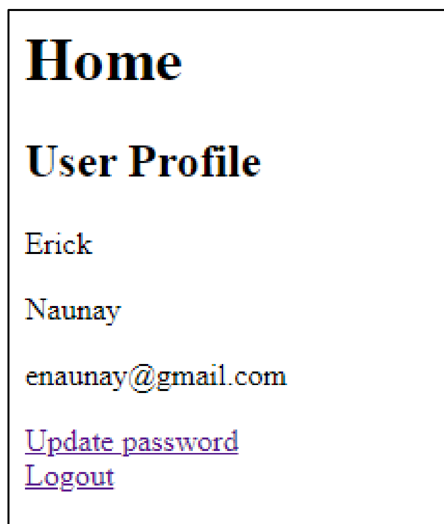


Figura 23. Resultado validación token en la aplicación cliente

Cambio y reinicio de contraseña.

Finalmente, si el usuario desea actualizar su contraseña necesita llenar los datos: email, contraseña antigua y nueva contraseña como se muestra en la figura 24. En el back-end, el cliente hace una petición PUT a la ruta */authentication/change-password* al enviar el formulario. El anexo E muestra el flujo de información para el caso de uso cambio de contraseña. Por otra parte, si el usuario no se acuerda de su contraseña, este puede cambiarla con un proceso de dos pasos. El primero se muestra en la figura 25. Se envía un email con un código que luego es requerido en el formulario del segundo paso para completar el reinicio de la contraseña. La segunda etapa del proceso usa el código provisto previamente: email y la nueva contraseña, como se observa en la figura 26. El primer paso hace uso del endpoint */authentication/start-reset-password* mediante una petición POST, en el siguiente paso se realiza la petición PUT a */authentication/finish-reset-password*. El anexo F muestra como la información fluye para completar el caso de uso: reinicio de contraseña.

Change password

enaunay@gmail.com

Change

© 2017-2020

Figura 24. Resultado cambio de contraseña en la aplicación cliente

Forgot yout password?

Email

Start

© 2017-2020

Figura 25. Resultado inicio del reinicio de la contraseña en la aplicación cliente

Reset Password

enaunay@gmail.com

345234

Finish

© 2017-2020

Figura 26. Resultado final del reinicio de la contraseña en la aplicación cliente

Dificultades encontradas y sus soluciones

La aplicación hace uso de un servicio de email externo que se encarga de entregar a los recipientes el email con el código para cambiar la clave. Muchos de estos servicios requieren de un dominio propio para hacer pruebas; sin embargo, otros usan un sandbox para usarse en desarrollo. Debido a la versatilidad en estos servicios se abstrajo el concepto de proveedor en la capa de infraestructura y se implementaron varios proveedores sobre la misma interfaz. El proyecto puede usar cualquiera de los proveedores sin importar el ambiente y la etapa en el que se encuentre el desarrollo, esto se puede configurar en el archivo .env.

Un servicio REST siempre debe contener documentación y pruebas. En etapas tempranas de desarrollo se analizó el posible uso de algún entorno de desarrollo de APIs para documentar y realizar las pruebas de integración; sin embargo, documentar de forma manual no es lo más optimo, por lo que se investigó e implemento la tecnología Swagger Docs, la cual permite generar documentación de la API automáticamente mediante ficheros de configuración JSON.

Mejoras conocidas

El proyecto cuenta con una base de datos embebida en el servidor, SQLite. Esta base de datos es de bajo consumo con buenas prestaciones en rendimiento y confiabilidad. Sin embargo, la autenticación al ser un servicio independiente, la base de datos también debería de serlo, pero con SQLite este no es el caso ya que la base de datos se encuentra como un fichero en el servidor. Si el servicio tuviese cientos de miles de usuarios y de operaciones en un periodo de tiempo corto, el servicio se vería afectado por el rendimiento en lectura y escritura en la base de datos, por lo tanto, lo ideal sería abstraer el concepto de proveedor de base de datos para así configurar desde el archivo .env el gestor de base de datos al que se va a conectar.

Por otro lado, la seguridad del servicio ante ataques cibernéticos como denegación de servicio y hombre en el medio dependerán de la calidad de protección que reciba el servidor de autenticación ante estos ataques.

Trabajo futuro

En la actualidad existen protocolos de comunicación que permiten acceder a información de aplicaciones de terceros si el usuario está autorizado a hacerlo. Esto permite que no sea necesario crear una cuenta en un dominio específico, sino se puede usar la información de otro dominio sobre el mismo usuario. El protocolo que permite esto es OAuth 2.0.

Como trabajo futuro se deberían abstraer los conceptos necesarios para poder integrar OAuth y permitir iniciar sesión con cualquier aplicación que tenga una API con soporte de OAuth 2.0. Además, el proyecto debería contar con un script en Docker para hacer el despliegue del servicio a un ambiente productivo con una sola línea de comando que construya la imagen y la ejecute.

Por último, el servicio debería tener visibilidad en todo momento, esto se consigue mediante la distribución de los logs a un sistema de persistencia en la nube o local.

CONCLUSIONES

En la universidad existen pocas clases que requieren desarrollar un proyecto bien estructurado y que siga estándares de diseño que dan una gran calidad al código; el desarrollo de este proyecto me permitió poner en práctica todos los conocimientos adquiridos en la carrera universitaria. Sin embargo, debido a la evolución constante de las herramientas y tecnologías de desarrollo no existe el camino perfecto, pero gracias a la investigación en el campo de diseño de sistemas existen herramientas como DDD y TDD que permiten tener una mejor visión sobre el sistema y que implementaciones son las óptimas para implementar la lógica de negocio.

Por otra parte, desarrollar el servicio y la aplicación cliente me permitió observar la importancia de la documentación de la API o cualquier capa de aplicación usada: CLI, RPC, etc. La documentación es parte fundamental de un sistema, ya que provee la información necesaria sobre el funcionamiento del sistema.

Finalmente, desarrollar un proyecto con altas prestaciones y con una visión de que será usado como un servicio independiente por otras personas que no formaron parte del equipo de desarrollo, me permitió aprender de la complejidad de un sistema real y como construirlo desde sus cimientos, pero sobre todo de cuán importante es el descubrimiento de requisitos y todas las consideraciones en el manejo de errores y exposición de los casos de uso siguiendo buenas prácticas de documentación.

REFERENCIAS BIBLIOGRÁFICAS

Andrawos, M., & Helmich, M. (2017). Cloud Native Programming with Golang: Develop microservice-based high performance web apps for the cloud with Go. Packt Publishing Ltd.

Boyd, C., Hale, B., Mjølsnes, S. F., & Stebila, D. (2016, February). From stateless to stateful: Generic authentication and authenticated encryption constructions with application to TLS. In *Cryptographers' Track at the RSA Conference* (pp. 55-71). Springer, Cham.

Dacosta, I., Chakradeo, S., Ahamad, M., & Traynor, P. (2012). One-time cookies: Preventing session hijacking attacks with stateless authentication tokens. *ACM Transactions on Internet Technology (TOIT)*, *12*(1), 1-24.

De, B. (2017). API documentation. In *API Management* (pp. 59-80). Apress, Berkeley, CA.

Janzen, D., & Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. *Computer*, *38*(9), 43-50.

Jones, M., Campbell, B., & Mortimore, C. (2015). JSON Web Token (JWT) profile for OAuth 2.0 client authentication and authorization Grants. *May-2015. [Online]. Available: https://tools. ietf. org/html/rfc7523*.

Krause, Jörg. "Introduction to Bootstrap." In Introducing Bootstrap 4, pp. 1-17. Apress, Berkeley, CA, 2020.

Owens, M., & Allen, G. (2010). SQLite. Apress LP.

Richardson, Leonard, and Sam Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008.

Rodriguez, A. (2008). Restful web services: The basics. *IBM developerWorks*, *33*, 18.

Sriramya, P., & Karthika, R. A. (2015). Providing password security by salted password hashing using Bcrypt algorithm. *ARPN journal of engineering and applied sciences*, *10*(13),

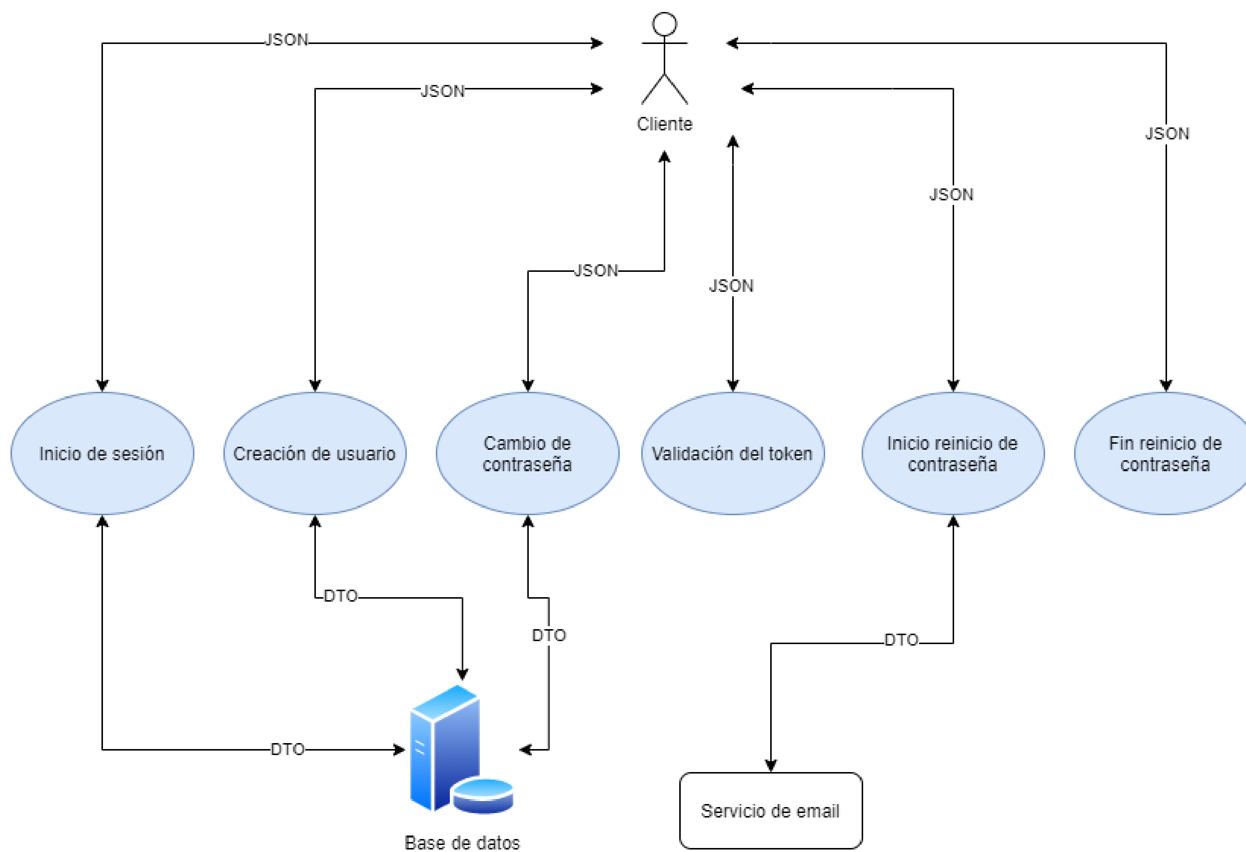
5551-5556.Martin, R. C. (2018). Clean architecture: a craftsman's guide to software structure and design. Prentice Hall. Martin, R. C. (2018). Clean architecture: a craftsman's guide to software structure and design. Prentice Hall.

Thönes, J. (2015). Microservices. *IEEE software*, *32*(1), 116-116.

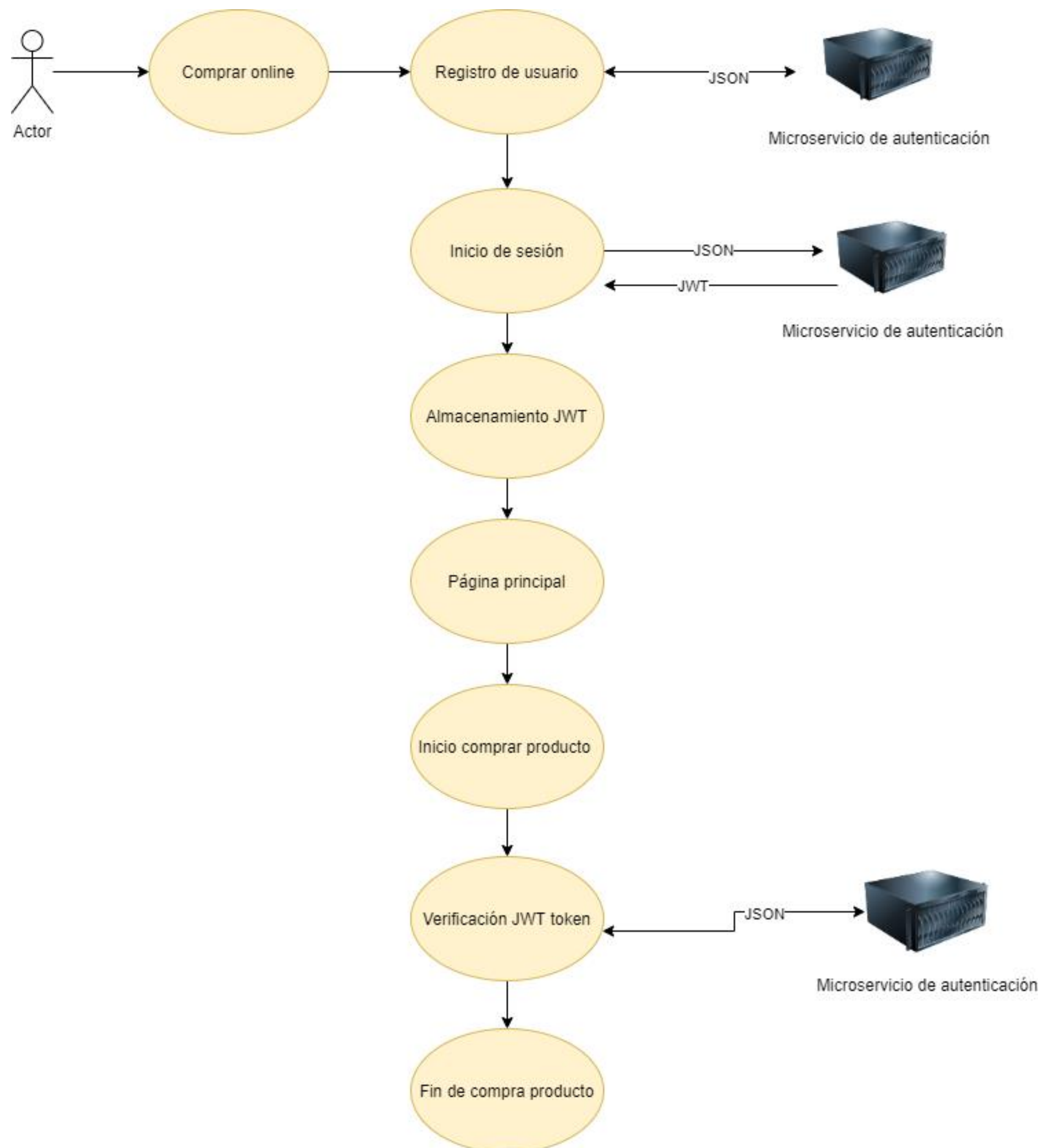
Vernon, V. (2013). *Implementing domain-driven design*. Addison-Wesley.

Wiggins, A. (2017). The Twelve-Factor App. *[Online]. Available: https://12factor.net/*.

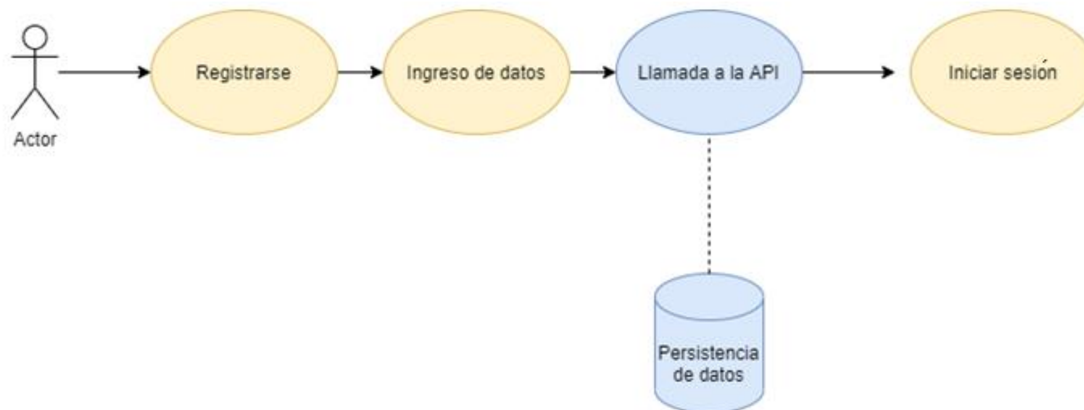
ANEXO A: DIAGRAMA DE LOS MÓDULOS PRINCIPALES



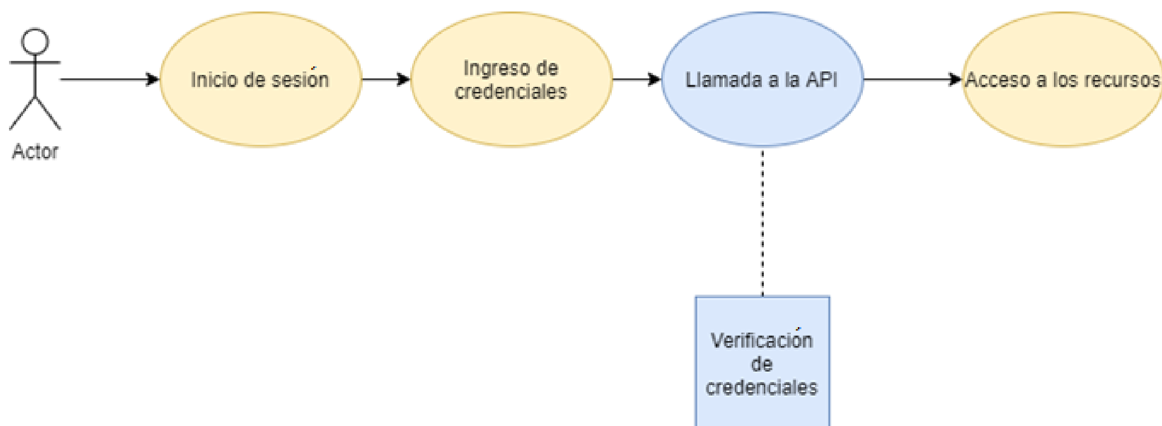
ANEXO B: DIAGRAMA DE FLUJO DEL JWT TOKEN



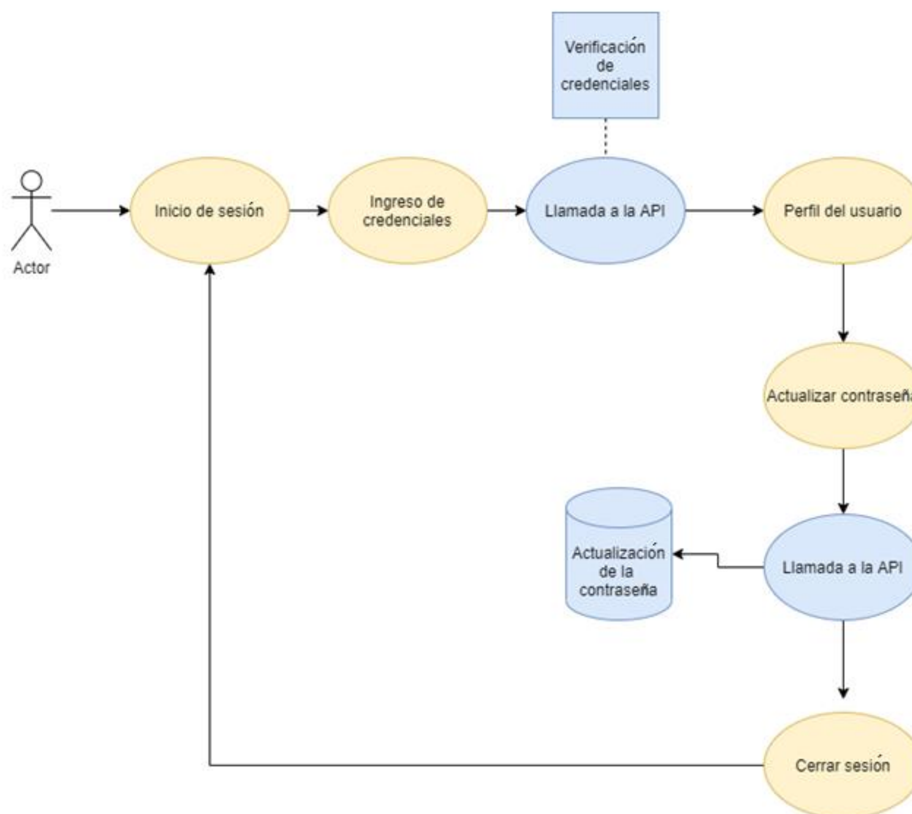
ANEXO C: DIAGRAMA DE FLUJO DEL CASO DE USO: REGISTRO DE USUARIO



ANEXO D: DIAGRAMA DE FLUJO DEL CASO DE USO: INICIO DE SESIÓN



ANEXO E: DIAGRAMA DE FLUJO DEL CASO DE USO: CAMBIO DE CONTRASEÑA



ANEXO F: DIAGRAMA DE FLUJO DEL DE USO: REINICIO DE CONTRASEÑA