Plataforma cognitiva para la predicción de demanda energética

Daniel Torres González

Informe de práctica profesional, como requisito para optar por el título de:

Ingeniero de Sistemas.

Asesor

Astrid Duque Ramos, Doctora en Ingeniería informática

Martín Elías Quintero Osorio, Ingeniero de Sistemas

Universidad de Antioquia

FACULTAD DE INGENIERÍA

INGENIERÍA DE SISTEMAS

MEDELLÍN

2022

Cita	Torres González [1]
Referencia Estilo IEEE (2020)	[1] D. Torres González, "Plataforma cognitiva para la predicción de generación y demanda energética", Semestre de industria, Ingeniería de Sistemas, Universidad de Antioquia, Medellín, Antioquia, Colombia, 2022.

Universidad de Antioquia - www.udea.edu.co

Rector: John Jairo Arboleda Céspedes.

Decano/Director: Jesús Francisco Vargas Bonilla.

Jefe departamento: Diego José Luis Botía Valderrama.

Agradecimientos

A cada profesor de la Facultad de Ingeniería de la Universidad de Antioquia, quienes acompañaron mi proceso durante todos estos semestres.

A la empresa guane Enterprises por la confianza depositada en mí y por todos estos meses de aprendizaje y desarrollo profesional.

TABLA DE CONTENIDO

LISTA DE FIGURAS

SIGLAS, ACRÓNIMOS Y ABREVIATURAS

CREG	Comisión de Regulación de Energía y Gas
SDL	Sistema de Distribución Local
STR	Sistema de Transmisión Regional
SIN	Sistema Interconectado Nacional
OR	Operador de Red
MC	Mercados Comercializadores
CND	Centro Nacional de Despacho
API	*Application Programming Interface*
DevOps	*Development and Operations*
IEEE	*Institute of Electrical and Electronics Engineers*
PO	*Product Owner*
HU	Historias de usuario
REST	*Representational State Transfer*
MSA	*Microservices Architecture*
JS	JavaScript

RESUMEN

Cuando una empresa suple la necesidad energética de un sector pueden ocurrir dos casos críticos que entran en manos de la Comisión de Regulación de Energía y Gas (CREG) de Colombia: el primero es que haya un déficit, provocando el sometimiento a cortes de energía voluntarios, y el segundo, puede haber una sobreproducción, provocando una multa a la empresa por parte de la CREG. Ambas situaciones son igual de malas, ya sea para los clientes por las consecuencias de un corte de energía o para la empresa por la pérdida monetaria. Para prevenir cualquiera de las dos problemáticas anteriormente mencionadas, Guane Enterprises, una empresa que proporciona soluciones basadas en machine learning, propone desarrollar un aplicativo que sirve para predecir la demanda energética.

En este proyecto se plasma la planeación, organización y desarrollo de la aplicación compuesto por una serie de *sprints* donde se deja constancia de las tareas realizadas durante cada uno como la construcción de una API con servicios que cumplan los requerimientos necesarios y que pueden ser consumidos por cada interfaz de usuario para que finalmente lleguen al cliente.

Palabras clave — **Sector energético, demanda eléctrica, generación eléctrica, comercializador, pronóstico, API.**

ABSTRACT

When a company meets the energy needs of a sector, two critical cases may occur that enter into the hands of the Colombian Energy and Gas Regulatory Commission (CREG): the first is that there is a deficit, causing the company to be subject to voluntary power cuts, and the second is that there may be an overproduction, causing the company to be fined by the CREG. Both situations are equally bad, either for the customers because of the consequences of a power outage or for the company because of the monetary loss. To prevent either of the two aforementioned problems, Guane Enterprises, a company that provides solutions based on machine learning, proposes to develop an application that serves to predict energy demand.

This project includes the planning, organization and development of the application composed of a series of sprints where the tasks performed during each one is recorded, such as the construction of an API with services that meet the necessary requirements and can be consumed by each user interface to finally reach the customer.

Keywords — **Energy sector, electrical demand, electricity generation, marketer, forecast, API**

I. INTRODUCCIÓN

Guane Enterprises SAS, es una compañía que ha tenido un crecimiento significativo en los últimos años y que ofrece soluciones a problemas de la industria y automatización por medio de desarrollo de sistemas informáticos aplicando herramientas de ciencia de datos. Esta empresa se desenvuelve en una serie de sectores económicos principales, entre estos está el sector energético, el cual abarca no solo la producción sino también el transporte, tratamiento y venta de productos energéticos del país.

Thori, es una plataforma cognitiva que automatiza el pronóstico de demanda eléctrica desarrollada precisamente para los desarrolladores y distribuidores de energía. Esta plataforma, además de generar pronósticos, proporciona la capacidad de agregar correcciones a un pronóstico ya generado por medio de factores externos a la demanda de un mercado comercializador como las condiciones meteorológicas, los eventos atípicos de índole social (e.g. protestas y paros nacionales, partidos de fútbol, entre otros) y eventos de conexión/desconexión de actores energéticos (cogeneradores, clientes industriales y autogeneradores, entre otros).

Esta práctica tiene como objetivo el desarrollo de una interfaz de programación de aplicaciones o API, con diversos servicios que le permiten al aplicativo Thori gestionar, programar, corregir y reportar pronósticos de generación y demanda eléctrica. Las funcionalidades que debe llevar la aplicación Thori se realizan acorde a las exigencias y necesidades de los usuarios funcionales.

II. PLANTEAMIENTO DEL PROBLEMA

La CREG es una entidad adscrita al Ministerio de Minas y Energía de Colombia y se encarga principalmente de regular los servicios de electricidad y gas del país. La red eléctrica colombiana está conformada por diferentes Sistemas de Distribución Local (SDL) y Sistemas de Transmisión Regional (STR), en un aparato jerárquico denominado Sistema Interconectado Nacional (SIN). Los SRTs son administrados por los Operadores de Red (OR) y están encargados de su planeación, inversión, operación y mantenimiento, esto según la resolución 097 de la comisión de regulación de energía y gas, de 26 de septiembre de 2008. Los ORs tienen definidos dentro de sus operaciones agrupaciones de usuarios regulados y no regulados conocidos como mercados comercializadores y son precisamente estos mercados los designados para un constante control, auditoría y regulación por parte de la CREG, a través del Centro Nacional de Despacho.

Hoy en día, los ORs deben enviar semanalmente pronósticos de la demanda eléctrica de los Mercados Comercializadores (MC), para la siguiente semana y el siguiente día respectivamente. La CREG en su resolución 100 de 2017 definió un proceso de auditoría mucho más riguroso sobre los pronósticos de demanda eléctrica con sanciones monetarias si los ORs continuamente se desvían en los pronósticos de sus MCs.

La plataforma de pronósticos permite a los comercializadores de energía automatizar, gestionar, programar, corregir y reportar el proceso de generación de pronósticos. Su implementación requiere de un largo proceso de planeación que tiene como objetivo brindarle al usuario una plataforma de fácil interacción y que cumpla con sus necesidades y, para llevar a cabo esta tarea, se escogen algunas herramientas de desarrollo que permitan la construcción de una API con una serie de servicios a los cuáles se puede conectar cualquier plataforma para consumirlos.

III. OBJETIVOS

A. Objetivo general

Apoyar el desarrollo del backend de una plataforma cognitiva para la predicción de la generación y la demanda de energía en Colombia.

B. Objetivos específicos

- Realizar un análisis exploratorio de las diferentes fuentes de datos proporcionadas por el cliente para determinar el diseño de una base de datos relacional basada en el dominio conceptual del cliente.
- Diseñar una arquitectura basada en microservicios que permita separar la aplicación en diferentes módulos basados en reglas de negocio.
- Desarrollar diferentes microservicios en el backend los cuales conectados entre sí cumplan con el funcionamiento esperado de la aplicación.
- Ejecutar pruebas unitarias y de integración al proyecto para aumentar la certeza y calidad del buen funcionamiento de la aplicación.

IV. MARCO TEÓRICO

El punto de partida de este trabajo de prácticas es la contextualización de la información necesaria para comprender el proyecto, es decir, la lógica del negocio, por esto, es fundamental definir algunos conceptos clave de este ámbito.

En este tipo de proyectos se debe realizar una recopilación de los requisitos definidos por usuario final o el Product Owner (PO), este tiene un rol muy importante en el proyecto ya que es quien tiene la responsabilidad de brindar la información requerida, definir prioridad en las funcionalidades o incluso, ser el intermediario entre todas las personas interesadas, que incluye tanto el equipo de trabajo como el usuario final [1]. A medida que el PO habla con el equipo de trabajo se definen las Historias de Usuario (HU) ya que son un método popular para plasmar las necesidades que debe cumplir la aplicación utilizando una plantilla general y sencilla de implementar, siempre con el objetivo de mantener la calidad que permite el mejor rendimiento del equipo [2]. Una manera segura y cómoda de almacenar las HU, además de ofrecer un control riguroso y actualizado, es implementar una herramienta específica para esta tarea como lo es Azure DevOps, una plataforma que permite agrupar el desarrollo de software junto con las operaciones de TI acelerando los tiempos de entregas [3].

La arquitectura de microservicios permite crear una aplicación como un conjunto de pequeños servicios granulares que utilizan un mecanismo de comunicación, normalmente por principios REST (Representational State Transfer) [4], mientras que la arquitectura orientada a servicios emplea un estilo de orquestación para coordinar los servicios. La arquitectura de microservicios la componen servicios pequeños, fáciles de comprender y que mantienen la lógica del negocio entre ellos, además, cada servicio puede escalar sus funcionalidades de manera independiente a los demás, entonces, si se maneja un acoplamiento lo suficientemente bajo entre ellos, MSA (Microservices Architecture) puede ser considerado como una fase siguiente de la arquitectura orientada a servicios [5].

Antes de desarrollar la aplicación se debe hacer una configuración previa de las herramientas de trabajo, estandarizándolas y apropiándolas con los valores requeridos acorde a las necesidades.

En la arquitectura, cuando se crea cada microservicio, se compacta en un contenedor que lo mantenga separado de los demás, para esto se utiliza Docker, una tecnología basada en contenedores que encapsula la aplicación volviéndola independiente de las demás aplicaciones [6]. Un microservicio es la unidad mínima en la arquitectura, entonces, a medida que la aplicación crece, el número de contenedores aumenta volviendo mucho más compleja la tarea de administrarlos y enviar datos entre ellos. Para resolver este problema se necesita implementar una tecnología que mantenga los servicios de manera automática, por ejemplo, Kubernetes, una herramienta que permite orquestar todos estos contenedores sin interferir con ninguna implementación a nivel de aplicación, es decir, facilita la gestión y administración del despliegue de los contenedores a nivel de infraestructura y hardware, además, “Kubernetes proporciona una recuperación al reiniciar automáticamente los contenedores fallidos y reprogramarlos cuando se deshabilitan sus hosts. Esta capacidad mejora la disponibilidad de la aplicación.” [7].

Existe una serie de herramientas necesarias que se emplean para llevar a cabo el desarrollo. Para editar el código se hizo uso de Visual Studio Code, ya que, es ligero pero potente y compatible con sistemas operativos como Windows, macOS y Linux, además, es afín con lenguajes de programación como Python [8] y JavaScript [9], los cuales se usan en el proyecto. En el proyecto es necesario usar persistencia de datos debido a que permite realizar el estudio relativo de las predicciones. Para gestionar los datos se optó por emplear SQL Server, un sistema de administración de bases de datos relacionales que tiene algunas características como cifrado de datos, enmascaramiento dinámico de datos, compatibilidad con JSON, entre otros [10] y se crea un servicio de Node Js que se conecte al gestor y haga las transacciones respectivas al momento de realizar peticiones.

V. METODOLOGÍA

La metodología empleada por el equipo de desarrollo para la realización de la aplicación consta de seis pasos que pueden ser iterativos ya que, gracias a la flexibilidad de la arquitectura usada, tiene una alta escalabilidad. Una vez que el microservicio ya está construido, este debe pasar una serie de pruebas especificadas para poder ser desplegado, de lo contrario, se realizan las correcciones correspondientes hasta que ninguna prueba falle. El anterior diagrama muestra el orden de los pasos que serán detallados más adelante en el orden respectivo.

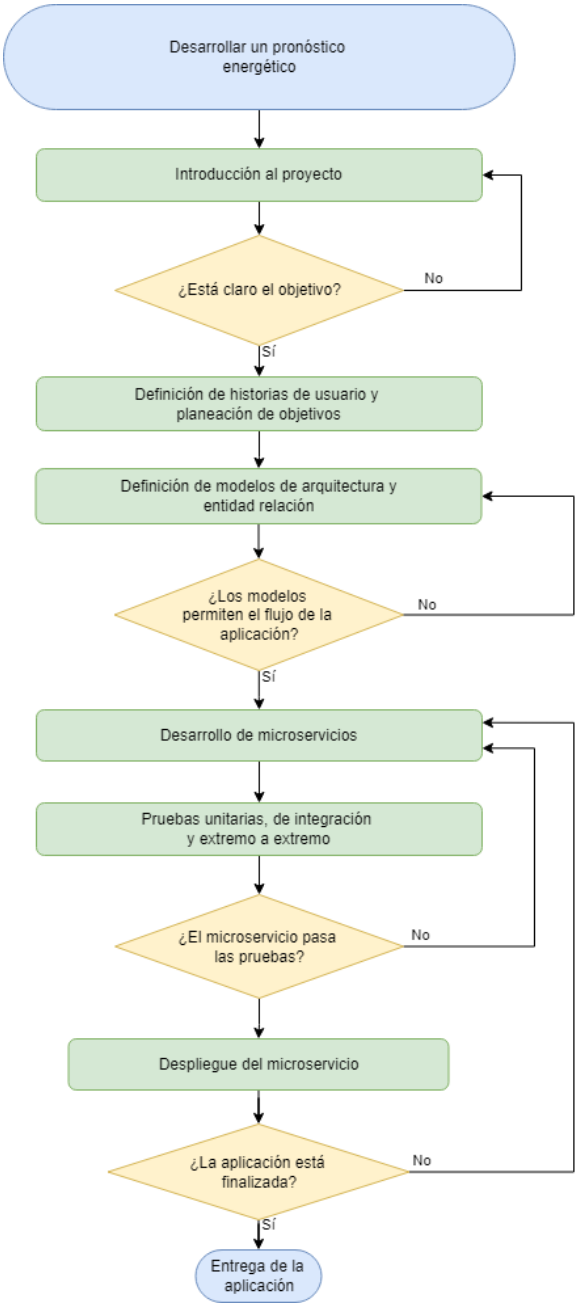

Fig. 1 Diagrama de flujo de la metodología

A. *Introducción al proyecto y definición de punto de partida*

Para empezar, la empresa realizó una introducción al proyecto para tener un primer acercamiento a la lógica del negocio. La prioridad fue mantener la comunicación con el PO (principal interesado en el proyecto, puede ser una única persona o un grupo de ellas). Para la explicación general de las funcionalidades requeridas se usaron HUs , las cuales fueron definidas desde la perspectiva del cliente y a partir de estas, se levantaron los requisitos con los que se trazó un punto de partida de la aplicación. Luego, se realizó la planeación de los objetivos a cumplir y se delegó las responsabilidades a cada miembro del equipo. Para guardar el registro tanto de los requerimientos, como de las tareas que se asignaron, se hizo uso de la plataforma Azure DevOps, ya que en esta se pueden escribir HU con un formato estándar y provee una vista más cómoda para leerlas, entenderlas y administrarlas. Por otra parte, Azure DevOps cuenta con la funcionalidad de administrar la implementación del marco de trabajo Scrum, ya que se pueden gestionar las tareas por fases, *sprints* y *daily*, entonces, se optó por adaptarlo en este desarrollo.

B. *Configuración del entorno de desarrollo*

Debido a que es necesario configurar el entorno de desarrollo, se utilizó una base estándar definida por la empresa, la cual consta de apropiar una serie de herramientas que facilitan la creación de la aplicación. Las herramientas que utilizaron son: Visual Studio code para editar el código, Docker que se encarga mantener los servicios en contenedores, Fast API y Node Js que son los frameworks con los que se desarrolla el código, estos dos últimos con sus respectivas bibliotecas; y SQL server que permite la persistencia de los datos que son usados cuando se realizan los estudios relativos a la hora de ejecutar el pronóstico. A medida que se desarrollaba el proyecto alguna configuración podría variar, aunque se manejaba un estándar, se tenía flexibilidad con el objetivo de facilitar cada tarea.

C. *Definición de modelos de arquitectura y entidad-relación*

Se utilizó una arquitectura de microservicios, esta hace referencia a que la aplicación se descompone en pequeñas partes y que cada una tenga su responsabilidad diferente a las demás, además, una de las principales ventajas que tiene esta arquitectura es que los problemas que surjan se pueden tratar de manera independiente sin afectar el resto de la aplicación.

Las entidades de la base de datos que se creó se deben plasmar conceptualmente, para este caso, se diseñó un modelo entidad-relación. En este modelo se plasman las entidades que

representan un objeto o concepto de la vida real compuestas por atributos que son las características que las definen, y de igual manera, entre ellas se pueden presentar relaciones, por ejemplo, si una entidad es una característica de otra entidad.

D. *Desarrollo de la aplicación*

En la fase de desarrollo comenzó la construcción de la aplicación. De la mano de este proceso se actualizaba el control de versionamiento por medio de repositorios de código. En ese sentido, existen una serie de aplicaciones para almacenar el código en la nube pero en este caso se optó por usar el repositorio proporcionado por Azure. Simultáneamente con el desarrollo, se implementaron pruebas unitarias a cada funcionalidad con el propósito de asegurar un nivel de confiabilidad en el código escrito.

E. *Desarrollo pruebas de integración*

En esta fase se realizaron pruebas donde se combinaron más de una funcionalidad, es decir, se probó que cada una de las funcionalidades independientes se comportara de la manera adecuada cuando se ejecutaban en conjunto.

F. *Despliegue de la aplicación*

La aplicación se hospedó en un host para poder hacer uso de ella desde cualquier lugar con conexión a internet. Esto fue posible gracias a las respectivas herramientas entre las cuales destacan Docker para crear contenedores independientes y kubernetes para administrar todos los contenedores trabajando en conjunto.

A medida que la aplicación necesitaba nuevas funciones a implementar se realizaba el proceso de designación de responsabilidades, desarrollo del código, las pruebas respectivas y el despliegue iterativamente para su completa construcción.

VI. RESULTADOS

En el desarrollo de la aplicación se definieron una serie de HU junto con el PO. Se optó por dividir el proyecto en fases y cada fase en cinco (5) *sprints*, además, en cada uno se hacían reuniones diarias para comunicar avances, dificultades o cualquier información importante para mantener a todo el equipo al tanto del estado del proyecto.

También se empleó una arquitectura de microservicios, la cual consta de un conjunto de aplicaciones más pequeñas y cada una está encargada de cumplir con responsabilidades independientes, no obstante, existe un microservicio que se encarga de transferir los datos entre todos o casi todos los microservicios restantes llamado Backend. En la figura 2 se presenta la arquitectura de la aplicación, donde cada cubo representa un microservicio que se puede comunicar con los demás para enviar la información necesaria y los cilindros representan un motor de persistencia de datos usado, de los cuales también se consulta en caso de ser necesario para alguna funcionalidad.

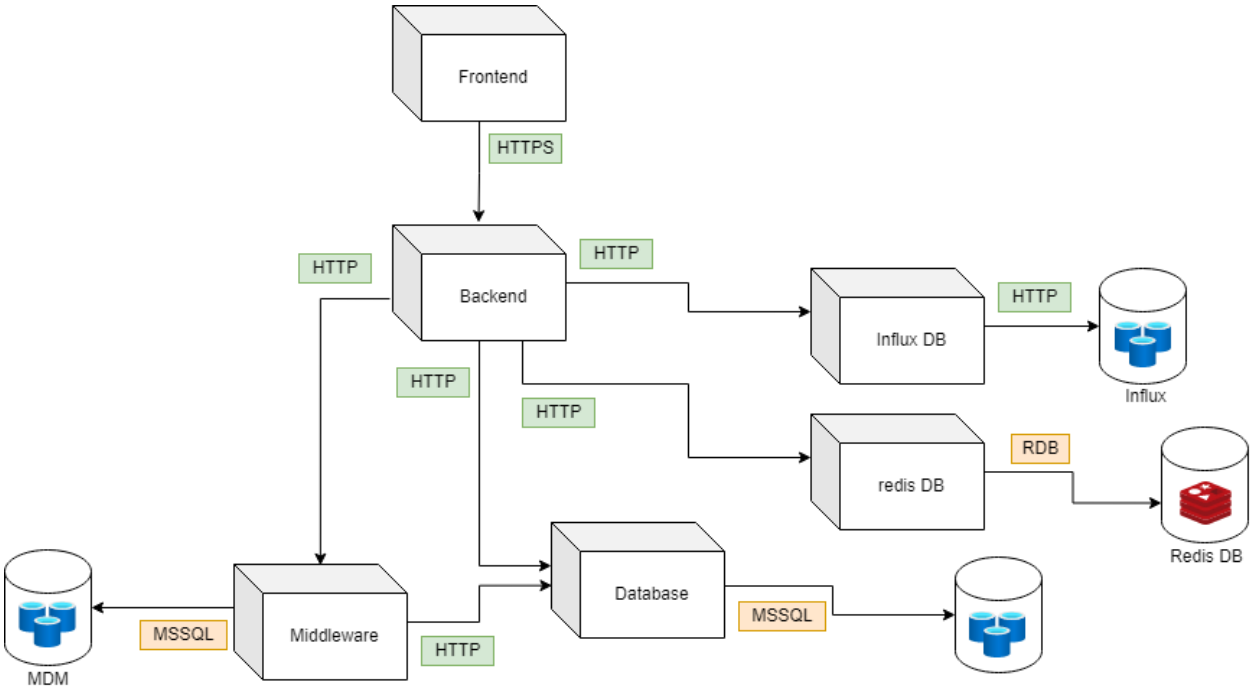

Fig. 2 Arquitectura de la aplicación.

Los microservicios principales creados se explican a continuación.

- **Database** es el microservicio encargado de administrar las transacciones de la base de la base de datos, en este caso, MsSql.

- **Redis DB**, usado para mantener la eficiencia en la aplicación. Se estima que la cantidad de datos que se van a transferir podrían aumentar tiempos de respuesta, sin embargo, Redis es una base de datos en memoria y por lo tanto, es mucho más rápida que una base de datos sql. Se implementa para optimizar los tiempos de respuesta cuando haya un alto índice de transacciones, y una vez terminada la función, se almacena todo en la base de datos principal.
- **Influxdb** es una base de datos para series de tiempo. Está optimizada para almacenar, leer, borrar y actualizar series de tiempo. Se tomó la decisión de implementarla ya que se especializa en este campo y permite eficiencia en los tiempos de respuesta.
- **Backend** es el microservicio que se encarga de administrar toda la API, desde la solicitud de información necesaria hasta el envío de las respuestas que se le da a cada uno de los microservicios. Este es el encargado de intermediar la comunicación entre toda la red arquitectónica.

A fin de expresar el modo en que las entidades se relacionan entre sí de una manera clara y concisa se diseñó un modelo entidad relación. Este modelo muestra las entidades que se almacenan en la base de datos junto con sus propiedades, es decir, pinta gráficamente el comportamiento que va a tener la base de datos. En este modelado, la entidad más importante se llama Boundary ya que contiene muchas características propias y relación con otras entidades, además, las entidades como consumo, ecuaciones de demanda, mercados, el pronóstico a predecir, entre otras funciones, tienen un comportamiento dependiente de la conducta de las fronteras.

En el segmento de diagrama de la Figura 3, se percibe una serie de relaciones entre varias entidades, la que más destaca es la de uno (1) a muchos, por ejemplo, un departamento tiene muchas fronteras. Al plasmarlas, se lee respectivamente cada relación y debe tener sentido a la hora de relacionarlo con el problema original.

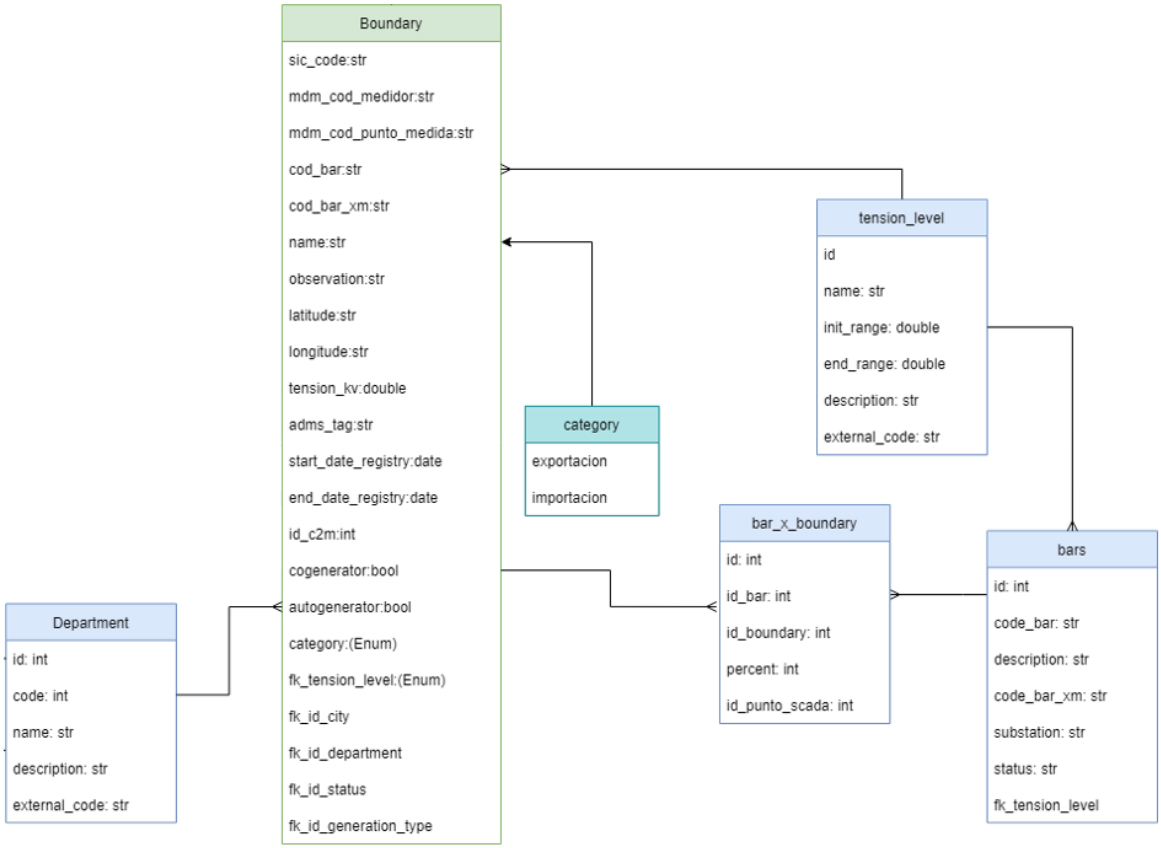

Fig. 3 Fragmento modelo entidad relación

Para el control de versionamiento, se hace uso de una aplicación que funciona como repositorio de código. A pesar de que Azure DevOps fue usada para registrar las HU y organizar los *sprints*, esta potente herramienta también cuenta con la capacidad de funcionar como repositorio de código; para poder almacenarlos de una manera ordenada, cada microservicio tiene su propio repositorio independiente.

La figura 4 muestra un segmento de código donde se establece la configuración inicial de un microservicio en desarrollo, por ejemplo, valores iniciales de la aplicación, si se hace una depuración o cualquier otra condición necesaria para el funcionamiento esperado.

main.py

Edit

Contents History Compare Blame

```python
from fastapi import FastAPI
from starlette.middleware.cors import CORSMiddleware
from starlette.requests import Request
import uvicorn

from app.api.api_v1.api import api_router
from app.core.config import config
from app.debugger import initialize_fastapi_server_debugger_if_needed


def initialize_app() -> FastAPI:
    initialize_fastapi_server_debugger_if_needed()
    application = FastAPI(
        title=config.PROJECT_NAME,
        openapi_url="/api/v1/openapi.json",
        version=config.API_VERSION,
    )
    application.include_router(api_router, prefix=config.API_V1_STR)
    return application


app = initialize_app()

# CORS
origins = []

# Set all CORS enabled origins
if config.BACKEND_CORS_ORIGINS:
    origins_raw = config.BACKEND_CORS_ORIGINS.split(",")
    for origin in origins_raw:
        use_origin = origin.strip()
        origins.append(use_origin)
    app.add_middleware(
        CORSMiddleware,
        allow_origins=origins,
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
        expose_headers=["filename"]
    ),


if config.MODE == "dev":
    if __name__ == "__main__":
        uvicorn.run(app, host=config.HOST, port=config.PORT)

```

Fig. 4 Segmento de archivo main de microservicio backend

El resultado final obtenido por parte la aplicación es una API, es decir, una interfaz de programación de aplicaciones, la cual expondrá una serie de servicios para que el microservicio frontend pueda consumirlos, de esta manera, funcionarán como dos aplicaciones diferentes. No obstante, se pueden conectar a partir de peticiones HTTP. Ahora bien, debido a que una API es la encargada de mostrar la respuesta que entrega el servidor, se vuelve complejo a la hora de visualizarla en texto plano, entonces, para facilitar esta tarea, se crea una documentación de las peticiones que recibe el servidor junto con la respuesta que puede dar cada petición.

La Figura 5 muestra de manera gráfica cuatro (4) de los servicios que existen en el microservicio backend y cada uno responde respectivamente a lo solicitado por las entradas, por ejemplo, el primero retorna una lista de objetos que representan niveles de tensión, el resto hacen la función de crear, obtener, eliminar y editar un único elemento respectivamente.

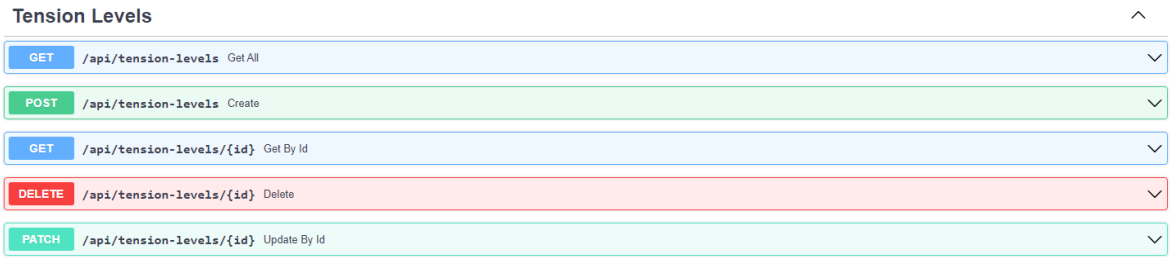

Fig. 5 Segmento de documentación de la API desarrollada.

VII. CONCLUSIONES

- La comunicación en un equipo es primordial para un desarrollo adecuado. En ese sentido, marcos de trabajo como scrum permiten una comunicación rápida y eficaz.

- Utilizar una arquitectura MSA permite que la aplicación sea fácilmente escalable debido a que a medida que se van teniendo necesidades se puede agregar un microservicio nuevo o editar uno existente.

- Desarrollar microservicios que sólo se conecten entre sí por peticiones HTTP permite mantener un bajo acoplamiento entre funciones aumentando su tolerancia al mantenimiento y la reutilización. De esta manera se evita que una actualización amerite cambios en varios servicios.

- Realizar pruebas unitarias y de integración evalúa que el funcionamiento de la aplicación sea el adecuado ya que, con las pruebas se cubren posibles casos de fallo que se pueden tener si la aplicación está en producción.

REFERENCIAS

[1] C. Unger-Windeler and K. Schneider, “Expectations on the Product Owner Role in Systems Engineering-A Scrum Team’s Point of View,” Proc. - 45th Euromicro Conf. Softw. Eng. Adv. Appl. SEAA 2019, pp. 276–283, Aug. 2019, doi: 10.1109/SEAA.2019.00050.

[2] G. Lucassen, F. Dalpiaz, J. M. E. M. van der Werf, and S. Brinkkemper, “The Use and Effectiveness of User Stories in Practice,” Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 9619, pp. 205–222, 2016, doi: 10.1007/978-3-319-30282-9_14.

[3] J. Rossberg, “Agile Project Management in Azure DevOps and TFS,” Agil. Proj. Manag. with Azur. DevOps, pp. 251–306, 2019, doi: 10.1007/978-1-4842-4483-8_8.

[4] O. Al-Debagy and P. Martinek, “A Comparative Review of Microservices and Monolithic Architectures,” 18th IEEE Int. Symp. Comput. Intell. Informatics, CINTI 2018 - Proc., pp. 149–154, Nov. 2018, doi: 10.1109/CINTI.2018.8928192.

[5] M. Waseem, P. Liang, and M. Shahin, “A Systematic Mapping Study on Microservices Architecture in DevOps,” J. Syst. Softw., vol. 170, p. 110798, Dec. 2020, doi: 10.1016/J.JSS.2020.110798.

[6] T. Combe, A. Martin, and R. Di Pietro, “To Docker or Not to Docker: A Security Perspective,” IEEE Cloud Comput., vol. 3, no. 5, pp. 54–62, 2016, doi: 10.1109/MCC.2016.100.

[7] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, “Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned,” IEEE Int. Conf. Cloud Comput. CLOUD, vol. 2018-July, pp. 970–973, Sep. 2018, doi: 10.1109/CLOUD.2018.00148.

[8] K. J. Millman and M. Aivazis, “Python for scientists and engineers,” Comput. Sci. Eng., vol. 13, no. 2, pp. 9–12, Mar. 2011, doi: 10.1109/MCSE.2011.36.

[9] E. Elliot, Programming JavaScript applications. 2014.

[10] K. Islam, K. Ahsan, S. A. K. Bari, M. Saeed, and S. Asim Ali, “Huge and Real-Time Database Systems: A Comparative Study and Review for SQL Server 2016, Oracle 12c & MySQL 5.7 for Personal Computer,” J. Basic Appl. Sci., vol. 13, pp. 481–490, Sep. 2017, doi: 10.6000/1927-5129.2017.13.79.