

UNIVERSIDAD AUTÓNOMA DE CIUDAD JUÁREZ

Instituto de Ingeniería y Tecnología

Departamento de Ingeniería Eléctrica y Computación



SISTEMA DE MONITOREO Y CONTROL PARA ESTACIONES METEOROLÓGICAS

Reporte Técnico de Investigación presentado por:

Teo González Calzada 142912

Requisito para la obtención del título de:

INGENIERO EN SISTEMAS COMPUTACIONALES

ASESOR:

Mtro. José Fernando Estrada Saldaña

Co-ASESOR:

Dr. Felipe Adrián Vazquez Galvez

Ciudad Juárez, Chihuahua

20 de mayo de 2022

Ciudad Juárez, Chihuahua, a 20 de Mayo de 2022

Asunto: Liberación de Asesoría

Mtro. Ismael Canales Valdiviezo

Jefe del Departamento de Ingeniería

Eléctrica y Computación

Presente.-

Por medio de la presente me permito comunicarle que, después de haber realizado las asesorías correspondientes al reporte técnico SISTEMA DE MONITOREO Y CONTROL PARA ESTACIONES METEOROLÓGICAS, del alumno Teo González Calzada de la Licenciatura en Ingeniería en Sistemas Computacionales, considero que lo ha concluido satisfactoriamente, por lo que puede continuar con los trámites de titulación intracurricular.

Sin más por el momento, reciba un cordial saludo.

Atentamente:

Mtro. José Fernando Estrada Saldaña

Profesor Investigador

Ccp: Mtro. David Absalón Uruchurtu Moreno
Coordinador del Programa de Sistemas Computacionales
Teo González Calzada
Archivo

Ciudad Juárez, Chihuahua, a 20 de Mayo de 2022

Asunto: Autorización de publicación

C. Teo González Calzada

Presente.-

En virtud de que cumple satisfactoriamente los requisitos solicitados, informo a usted que se autoriza la publicación del documento de SISTEMA DE MONITOREO Y CONTROL PARA ESTACIONES METEOROLÓGICAS, para presentar los resultados del proyecto de titulación con el propósito de obtener el título de Licenciado en Ingeniería en Sistemas Computacionales.

Sin otro particular, reciba un cordial saludo.

Dr. Everardo Santiago Ramírez

Profesor Titular de Seminario de Titulación II

Declaración de Originalidad

Yo, Teo González Calzada declaro que el material contenido en esta publicación fue elaborado con la revisión de los documentos que se mencionan en el capítulo de Bibliografía, y que la solución obtenida es original y no ha sido copiada de ninguna otra fuente, ni ha sido usada para obtener otro título o reconocimiento en otra institución de educación superior.

Teo González Calzada

Agradecimientos

A mis Padres por haberme dado las oportunidades necesarias para superarme.

A la Ing. Areli Rubio Rodríguez por su apoyo en la redacción y diversas tareas para la finalización de este documento, así como a su familia por proveerme con un ambiente adecuado para llevarlo a cabo.

Agradezco a mi asesor, el Mtro. José Fernando Estrada Saldaña por su inmerecida paciencia y ayuda con el conocimiento necesario para el desarrollo del proyecto, ofreciéndome su apoyo en los tiempos más difíciles y complicados de mi etapa estudiantil.

Dedicatoria

A mi padre, que ya no está conmigo, y a mi madre y hermana que lo están, por su apoyo incondicional durante todos estos años.

Resumen

En el presente documento se reporta el desarrollo de un sistema de monitoreo de estaciones meteorológicas que también permite el control de las mismas, desarrollado con tecnologías Web y con un enfoque en desarrollo modular. Este sistema fue probado durante 20 días monitoreando continuamente el estado de las estaciones meteorológicas y ofrece la información recabada en una interfaz comprensible.

Palabras clave: Monitoreo de sistemas, Sistema integral, Desarrollo web.

Índice general

1. Planteamiento del Problema	4
1.1. Antecedentes	4
1.2. Definición del problema	8
1.3. Objetivo general	9
1.3.1. Objetivos específicos	9
1.4. Justificación	9
1.5. Alcances y limitaciones	10
2. Marco referencial	12
2.1. Marco teórico	12
2.2. Marco tecnológico	14
2.2.1. Docker	14
2.2.2. Lenguajes de programación y <i>frameworks</i>	15
3. Desarrollo del Proyecto	18
3.1. Producto propuesto	18

ÍNDICE GENERAL

IX

3.2. Metodología de desarrollo	19
3.3. Análisis	21
3.3.1. Conexión a estaciones remotas	21
3.3.2. Selección de herramientas	24
3.4. Diseño	26
3.4.1. Prototipado de la interfaz gráfica	26
3.4.2. Diseño de logo e identidad gráfica	30
3.4.3. Diseño de base de datos	31
3.4.4. Configuración de ambiente de desarrollo	34
3.5. Desarrollo	39
3.5.1. De la base de datos	39
3.5.2. Del módulo de monitoreo de las estaciones	42
3.5.3. Del API para el acceso a la información	54
3.5.4. De la interfaz gráfica del proyecto	55
3.6. Pruebas	62
3.7. Despliegue	69
3.8. Evaluación	71
3.9. Lanzamiento	73
4. Resultados y Discusiones	74
4.1. Sobre los datos obtenidos	74
4.2. Sobre la extensibilidad del proyecto	77

<i>ÍNDICE GENERAL</i>	x
4.3. Sobre las capacidades de carga del proyecto	78
5. Conclusiones	82
5.1. Con respecto al objetivo de la investigación	82
5.2. Recomendaciones para futuras investigaciones	83
Bibliografía	84

Índice de figuras

1.1.	Tablero principal de Nagios XI.	6
1.2.	Diagrama de red de CECATEV UACJ.	8
2.1.	Diagrama del protocolo REST.	13
2.2.	Diagrama del contenedor de procesos Docker.	15
3.1.	Diagrama de metodología ágil.	20
3.2.	Diagrama de la redundancia de las conexiones.	22
3.3.	Prototipo del tablero de control.	27
3.4.	Prototipo de la interfaz para agregar una nueva estación.	28
3.5.	Prototipo de la interfaz de solución de errores.	29
3.6.	Evolución del logo de Meteoreo, ordenado cronológicamente.	30
3.7.	Modelo entidad-relación del proyecto meteoreo.	33
3.8.	Diagrama de clase de controladores	44
3.9.	Diagrama de clase de ejecutores.	46
3.10.	Diagrama de secuencia para el monitoreo de las estaciones.	47

ÍNDICE DE FIGURAS

XII

3.11. Captura de documentación de librería.	48
3.12. Visualización de documentación en OpenAPI.	55
3.13. Vista general de las estaciones.	57
3.14. Vista de una estación con fallas.	57
3.15. Vista general en un celular.	58
3.16. Vista de registro de una estación.	59
3.17. Vista de bitácora con filtros aplicados.	59
3.18. Diálogo de notificaciones para OneSignal.	60
3.19. Solicitud al servidor de pruebas con Insomnia.	62
3.20. Registro de la estación de prueba.	64
3.21. Estado incial de la estación de prueba.	65
3.22. Estado de la estación de prueba con servicios desactivados.	65
3.23. Estado final de la estación de prueba.	66
3.24. Modo sólo lectura desactivado en la estación IIT.	67
3.25. Modo sólo lectura reactivado en la estación IIT.	68
3.26. Despliegue de la interfaz web.	70
3.27. Actualización del API y servicio de monitoreo.	71
3.28. Información de monitoreo.	72
4.1. Tipos de incidencia.	75
4.2. Incidencias y sus tipos, agrupadas por estación.	76
4.3. Tiempo de inaccesibilidad acumulado por estación.	76

ÍNDICE DE FIGURAS

XIII

4.4. Rancho escuela.	77
4.5. Datos de falla de Rancho Escuela.	77
4.6. Gráfica de usuarios y tiempos de respuesta creada con Locust.	80
4.7. Estadísticas de pruebas por ruta.	81

Listados y configuraciones

3.1.	Archivo docker-compose.	35
3.2.	Dockerfile	36
3.3.	Archivos de migración en el proyecto.	40
3.4.	Definición de accesos y mutador para modelo de estaciones.	41
3.5.	Ejemplo de estructura de un servicio.	43
3.6.	Ejemplo del servicio para revisión de tiempo.	45
3.7.	Ejemplo del servicio para revisión de RoPI.	49
3.8.	Ejemplo del servicio WeeWX.	51
3.9.	Controlador para estación Campbell.	52
3.10.	Ejemplo del servicio para monitoreo de estación Campbell.	53
3.11.	Serialización de datos con FastAPI y MasoniteORM.	56
3.12.	Configuración de OneSignal en VueJS.	61
3.13.	Funcionamiento de estación de prueba de docker	63
4.1.	Diferencia de código al agregar el monitoreo de disco.	79
4.2.	lcoustfile.py configuración de pruebas.	80

Introducción

Las redes de monitoreo meteorológico y de calidad del aire son una necesidad creciente debido a su importancia para la recolección continua de datos y preservación para análisis, por ello, es importante crear y mantener redes densas de recolección de datos que sean estables y puedan ser mantenibles. La baja fiabilidad de la conexión a las estaciones provocada por su acceso a sistemas de conexiones inestables, además de la instalación en lugares de difícil acceso [1], provoca que sea un reto monitorearlas con sistemas convencionales. Esto, junto con la falta de un sistema homogéneo de monitoreo especializado en estaciones meteorológicas provoca que sea una tarea compleja y de alto costo el mantener una red en óptimas condiciones.

El proyecto reportado en el presente documento pretende cubrir esta área de oportunidad, desarrollando un sistema integral encargado de monitorear las estaciones meteorológicas y ejecutar acciones en las mismas, basado en la arquitectura de soluciones existentes y con un enfoque en una amplia extensibilidad. El sistema resultante es una serie de proyectos capaces de manejar, registrar y visualizar el estado de las estaciones funcionando de manera integral para su continuo monitoreo.

En el primer capítulo del presente documento se aborda evaluación general del estado de las tecnologías de monitoreo y sus aplicaciones en el ámbito correspondiente, posteriormente se ahonda en las definiciones necesarias para la comprensión de el mismo. En el capítulo tres, se detalla el análisis para el desarrollo del proyecto propuesto y su definición. Para terminar en una evaluación general de los resultados obtenidos y las conclusiones a las que se llegó con

el desarrollo del mismo.

Capítulo 1

Planteamiento del Problema

Las redes de monitoreo de meteorológico y de calidad del aire de alta densidad son una necesidad creciente de las áreas urbanizadas, las cuales necesitan de una infraestructura cada vez más compleja para su monitoreo y mantenimiento. Estas necesidades crecientes de infraestructura han generado una demanda creciente de recursos humanos, y la falta de homogeneidad entre las estaciones de monitoreo presentan retos a conquistar para proveer servicios cada vez más sofisticados. En este capítulo, se discutirá la importancia de la infraestructura de monitoreo y se explicarán los diferentes métodos de realizarlo que son utilizados actualmente para monitorear las estaciones meteorológicas, así como una propuesta de solución para los problemas que los sistemas tradicionales existentes presentan.

1.1. Antecedentes

El desplegar y mantener una red meteorológica urbana compone bastantes retos: Entre la creciente dificultad de crear sistemas de medición estandarizados que se adapten al siempre cambiante paisaje urbano; como la instalación de los equipos de medición y de guardado de datos en áreas que permitan acceso para mantenimiento y que sean seguros; y la dificultad de encontrar un punto de acceso a internet adecuado para transferir la información generada, el generar una red de monitoreo es una tarea extensa y compleja.

Debido a estos retos, la comunidad de monitoreo climatológico y meteorológico se ha enfocado en la creación de sistemas que sean más eficientes y económicos. Entre estos esfuerzos, se encuentra el amplio uso de dispositivos RaspberryPi como centro de recolección de datos de estaciones de monitoreo [2], tanto caseras como profesionales, con la ayuda de sistemas abiertos para la recolección de datos como lo es WeeWX. Esto ha hecho factible el desplegar redes de 50 nodos de monitoreo con sensores económicamente viables para actores con un presupuesto limitado [3].

Estas redes densas requieren de un monitoreo continuo para mantener una alta calidad de los datos recabados, y así evitar las pérdidas por falta de mantenimiento. Entre los sistemas de monitoreo que pueden ser adaptados para el monitoreo de estaciones meteorológicas y la red que las soporta, se encuentra la plataforma Nagios, la cual es un sistema de monitoreo continuo orientado a redes y servidores. Entre la información que recaba Nagios continuamente para el estado de los servidores, se encuentra el uso de CPU y RAM, así como estado de los discos, puertos, e información variada de servicios de red en los hosts. Debido a que Nagios es un sistema de monitoreo de redes orientado a profesionales de la informática, la interfaz gráfica es poco amigable con los usuarios menos familiarizados con los conceptos técnicos de los sistemas computacionales, como se muestra en la Figura 1.1.

Nagios ofrece la posibilidad de monitorear parámetros adicionales y posee una comunidad establecida que desarrolla continuamente en diversos lenguajes, como Python, un sistema conformado por plugins relacionados con el proyecto. Sin embargo, la cantidad de éstos relacionados con estaciones meteorológicas existentes es mínima ya que los esfuerzos de la comunidad se centran principalmente en el monitoreo de centros de datos, redes y routers.

Además de los plugins existentes en la comunidad de Nagios, existe la alternativa abierta conocida como *monitoring plugins* [4], que es una plataforma compatible con diversos sistemas de monitoreo de redes y servicios, en la cual es posible encontrar una mayor cantidad de scripts de monitoreo relacionados con los sistemas de recolección de datos meteorológicos.

1.1. ANTECEDENTES

6

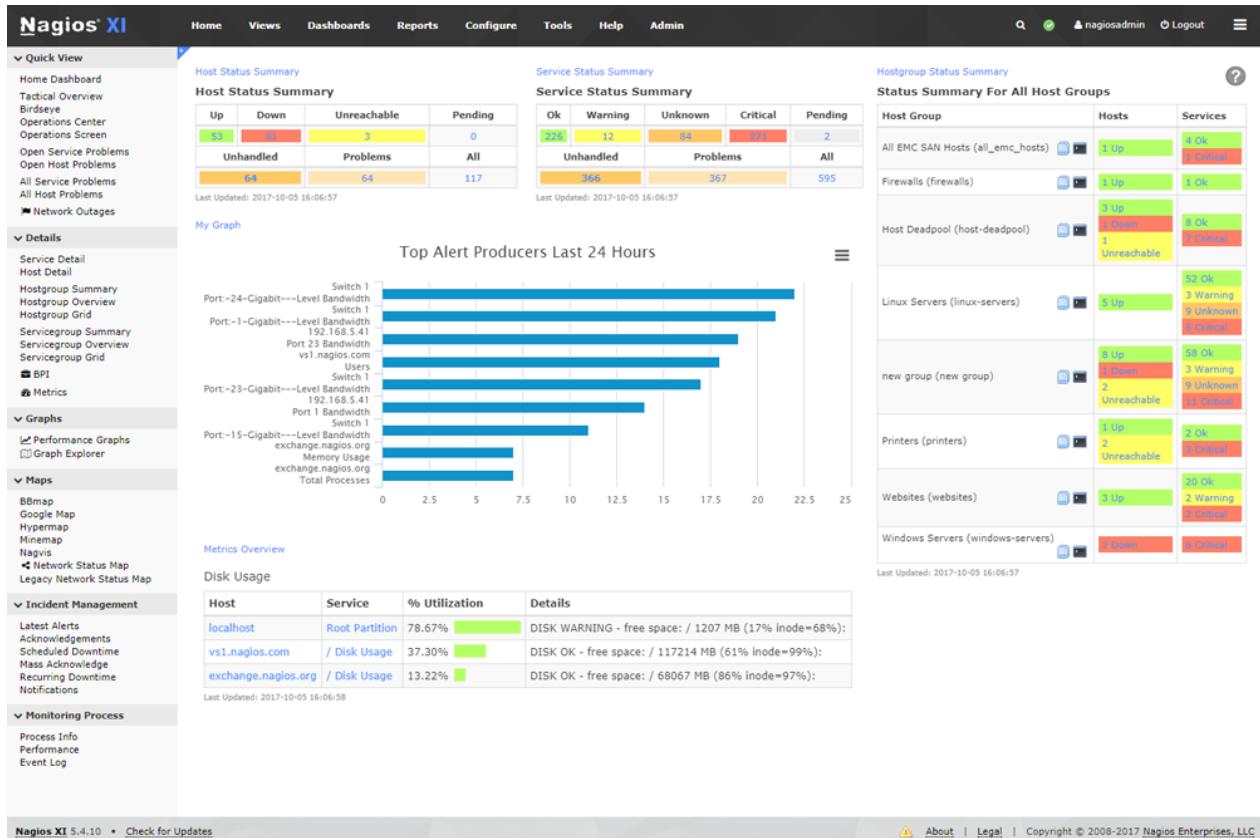


Figura 1.1: Tablero principal de Nagios XI.

La utilización de estos *plugins abiertos* junto con un sistema central como Nagios ya ha sido propuesto anteriormente, sin embargo, se descartó un monitoreo más profundo debido a que “las métricas obtenidas son muy simples y los valores se devuelven de comandos directos de sistema de GNU/Linux” [3].

Las limitaciones de Nagios vienen a raíz de su implementación orientada a sistemas de alta disponibilidad y el generalismo en la implementación de monitoreo, la configuración de los parámetros de tolerancia para la resiliencia a fallas son generalmente limitados en la variedad de los mismos y los valores fijos que se pueden establecer. Además, debido a las necesidades de seguridad y accesibilidad de las estaciones meteorológicas en el contexto urbano, estas suelen instalarse en sistemas en los que se posee poco control de la red que les

provee comunicación, como son las escuelas, hospitales y estaciones de policía, así como otros espacios públicos [1], dificultando aún más la capacidad de monitoreo y disponibilidad de la red y los sistemas que soporta. Esta limitante se hace más presente debido a que los sistemas se vuelven completamente dependientes de una VPN para funcionar y para su control, debido al extenso uso de redes IPv4 bajo NAT que predominan en los sitios de instalación.

Otra de las restricciones del sistema Nagios es que al ser un sistema de monitoreo limita la capacidad de acción de los usuarios ante un caso que lo requiera. Si bien es posible el realizar acciones por medio de scripts creados al momento de la configuración de Nagios, estos requieren una configuración extensa y compleja [5]. Además, el sistema basado en eventos impide la interacción directa de un usuario para la respuesta de forma directa desde la consola de administración, requiriendo que el usuario tenga un conocimiento del método de conexión, así como la información necesaria para realizar una tarea trivial como es el reiniciar un servicio.

Actualmente, el sistema de monitoreo de calidad del aire y climatológico del Centro de Ciencias Atmosféricas y Tecnologías Verdes (CECATEV) en la Universidad Autónoma de Ciudad Juárez, engloba varios de los retos antes descritos, ya que a pesar de la baja densidad de estaciones meteorológicas, comprende una variedad importante de las mismas. Esta compuesta por prototipos conectados por medio de redes virtuales privadas, monitoreados remotamente [6], estaciones de diferentes proveedores, siendo monitoreadas local y externamente, así como estaciones remotas con routers dentro de la red local de la universidad (véase la Figura 1.2). Lo que provoca que sea un reto monitorearlas adecuadamente debido a la variedad de sus componentes.

Desafortunadamente, la infraestructura de la red actual de CECATEV no cuenta con un sistema de monitoreo para el registro automático de problemas de las estaciones meteorológicas de la cual pueda ser derivada información de la calidad de los datos meteorológicos así como un estado de salud de las estaciones. Esto, dificulta el uso de forma adecuada de la

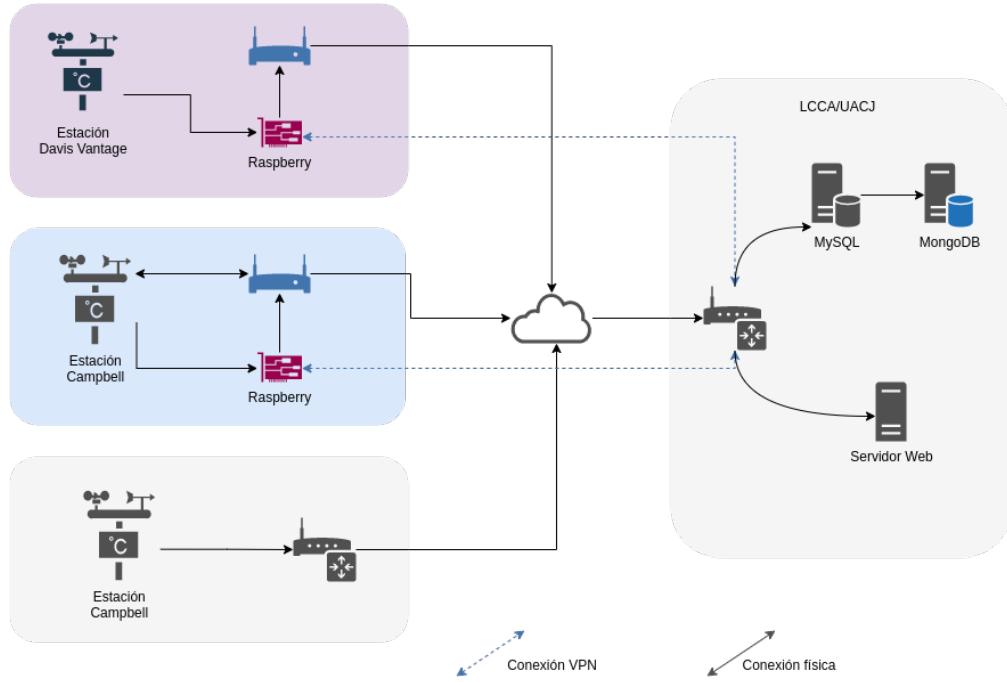


Figura 1.2: Diagrama de red de CECATEV UACJ.

información existente e impide la generación de reportes de calidad de la información.

1.2. Definición del problema

La falta de una plataforma estandarizada para el monitoreo de estaciones meteorológicas que permita un monitoreo continuo y control de diversos tipos de estaciones, así como la disponibilidad del acceso a los datos, y un registro de incidentes crea una dificultad creciente para los administradores de redes de monitoreo meteorológico. Con redes cada vez más densas que son más accesibles económicamente y menos complejas de crear, se hace notar la necesidad de un sistema estandarizado y compatible con soluciones existentes para el monitoreo de las estaciones.

1.3. Objetivo general

Desarrollar un sistema de monitoreo y control de estaciones meteorológicas para personal no especializado, capaz de proveer datos y herramientas que coadyuven en el mantenimiento preventivo y correctivo de las mismas.

1.3.1. Objetivos específicos

- Desarrollar un sistema modular central para coordinar y recolectar los datos del estado de las estaciones.
- Diseñar un API REST para consultar el estado de las estaciones meteorológicas.
- Diseñar e implementar una interfaz gráfica para monitorear el estado de las estaciones.
- Integrar las diferentes estaciones meteorológicas existentes al sistema creando los controladores correspondientes.
- Integrar un sistema existente de notificaciones/alertas de terceros para fallos críticos de las estaciones.

1.4. Justificación

Creando un sistema de monitoreo eficaz para las estaciones meteorológicas se pretende alcanzar una mayor calidad de los datos obtenidos de las mismas, así como una mejor documentación por consecuencia de los problemas de los sistemas meteorológicos. Lo cual permitirá dar el tiempo al personal especializado en enfocarse en expandir las redes existentes meteorológicas, mejorando a largo plazo la calidad y la definición de los datos recabados con la misma cantidad de esfuerzo.

Además, haciendo el sistema de monitoreo un proyecto público y compatible con soluciones existentes, se busca ayudar a mejorar la calidad y confiabilidad de las redes de monitoreo meteorológico, de calidad del aire y climatológico alrededor del mundo.

Este proyecto tiene un impacto tecnológico, debido a que se plantea desarrollar un sistema en el que se puedan integrar diversos datos que ayuden a generar un estado comprehensible de la calidad de la información de las redes meteorológicas que monitorea. Con respecto al impacto económico, el proyecto permitirá identificar y solucionar los problemas que pudieran ocurrir en las diferentes estaciones sin la necesidad de desplazarse para corregir la solución, propiciando un uso más eficiente de los recursos humanos y económicos de CECATEV cuyo financiamiento principal para la operación son recursos públicos.

El impacto ecológico del proyecto es indirecto, ya que el personal de CECATEV podrá dedicar mayor tiempo al tratamiento de la información generada por las estaciones y no en desplazarse a las diferentes estaciones para identificar los problemas y solucionarlos.

1.5. Alcances y limitaciones

Entre los alcances que se pretendían lograr con el proyecto, se contemplaron los siguientes:

- Se integraron estaciones meteorológicas existentes, que se encontraban conectadas a la red de comunicaciones del CECATEV ya sea directamente o por medio de una VPN.
- Se entregó un sistema instalado, funcional y listo para monitorear las estaciones seleccionadas, sin necesidad de configuraciones extraordinarias.

Si bien se pretende que el proyecto tenga la flexibilidad suficiente para adaptarse a nuevos casos de uso sin necesidad de reescribir grandes partes del mismo, también se consideran las siguientes limitaciones:

- La integración se limitó a cubrir casos conocidos y recurrentes de estaciones meteorológicas existentes.
- Solo se considera integrar una estación meteorológica de cada tipo en cada topología, dejando como proyecto futuro integrar la totalidad de la red de la universidad.
- El sistema se limita a cubrir los casos de fallas comunes y conocidos de las estaciones.
- Sólo se monitorean los servicios básicos de las estaciones meteorológicas que son vitales para el funcionamiento.

Capítulo 2

Marco referencial

En este capítulo se abordarán los conceptos clave para entender las necesidades y funcionalidad del proyecto, así como las tecnologías utilizadas para el desarrollo del mismo. Esto abarca el contexto necesario para el desarrollo del sistema

2.1. Marco teórico

De acuerdo a [1], las estaciones meteorológicas son una parte fundamental de las redes meteorológicas que ayudan a monitorear diversas variables. Éstas pueden ser temperatura, dirección y velocidad del viento, humedad relativa, precipitación, entre otros. Las redes meteorológicas se categorizan por tamaño, los cuales van desde redes a micro-escala que tienen un alcance de $10m^2$ hasta redes regionales, a macro escala, y globales. Cada red tiene diferentes configuraciones, métodos de recolección, y objetivos.

En el caso de la red meteorológica del CECATEV, la red es de una escala a nivel regional. Esta red se compone de diversas estaciones, las cuales están conectadas por redes privadas virtuales (VPN, por sus siglas en inglés) para facilitar la comunicación entre las mismas, esto, debido a que las redes de comunicación existentes no suelen contar con una forma de acceder a servicios dentro de la red a la que están conectadas desde un punto externo. La red meteorológica se compone de una variedad heterogénea de estaciones meteorológicas de

diversas marcas, capacidades, sensores y métodos de conexión y recolección de datos [6].

Los sistemas informáticos que se componen de más de un componente, utilizan diversos métodos de comunicación entre ellos. Desde el accesar directamente a localizaciones de memoria física o virtual en un dispositivo para compartir información hasta crear librerías compartidas entre sistemas para accesar a la información en un depósito externo (conocidas como APIs), cada forma de acceso a los datos tiene su propio nivel de abstracción, oportunidades, y desventajas, las cuales deben ser evaluadas antes de elegir una tecnología adecuada para responder a las necesidades de cada proyecto.

Un API REST es un estándar de acceso a la información de sistemas externos por medio de protocolos *Web*, tales como HTTP/HTTPS, los cuáles permiten la consulta de datos en cualquier lenguaje que permita realizar conexiones y consultas a sitios web [7], como se muestra en la Figura 2.1. Entre las principales características de los API REST se encuentra que no es necesario proveer un estado previo para acceder a la información, esto implica que no es necesario mas que conocer la ruta en la que se encuentran los datos requeridos para acceder a ellos. Las ventajas que ofrece es la amplia disponibilidad de acceso a los datos y la fácil integración con sistemas existentes de manejo y procesamiento de información [8].



Figura 2.1: Diagrama del protocolo REST.

Debido a la naturaleza libre de el desarrollo web, y la poca estandarización de la comunicación entre los clientes web con los servidores, se creó la iniciativa OpenAPI a partir de un estándar existente proveído por la compañía Smartbear, Swagger. Este estándar para la comunicación con sitios por medio de APIs REST rápidamente fué ganando popularidad gracias a la fundación Linux hasta convertirse en un estándar utilizado ampliamente por

diversas organizaciones y empresas [9].

El estándar OpenAPI es un esquema de definición de estructura de datos en JSON. En este esquema se definen las rutas a las cuales se puede acceder, los parámetros que aceptan y sus respectivos tipos de datos, así como la información que responde y los tipos de datos de los mismos. Y debido a la naturaleza abierta del esquema, este puede ser generado e integrado nativamente con el uso de diversas tecnologías de desarrollo. Permitiendo, por ejemplo, el generar las clases e interfaces correspondientes para el uso por medio de clases de los datos para lenguajes de programación fuertemente tipados [10].

2.2. Marco tecnológico

A continuación se presenta una descripción de las herramientas de tecnología que se utilizarán para el desarrollo del proyecto.

2.2.1. Docker

Docker es un sistema para la creación y distribución de imágenes de software, principalmente orientado a servidores, que permite el crear un ambiente agnóstico al sistema operativo del host que además es replicable. Es un estándar en la industria de desarrollo de software para crear sistemas complejos manteniendo una relativa simpleza al desplegar nuevas instancias [11].

Docker utiliza un sólo Kérmel de linux para la creación de los contenedores y cada uno de los contenedores puede contener hasta n procesos, lo que lo ayuda a reducir el tamaño de sistemas complejos como se muestra en la Figura 2.2. Además de ofrecer una mayor flexibilidad y escalabilidad para tanto para realizar pruebas en máquinas de desarrollo como para distribuir y empaquetar nuevas instancias en ambientes de producción, se ha demostrado que

el costo en eficiencia al sistema que lo ejecuta es mínimo comparado con otros métodos para la administración de sistemas complejos tales como las máquinas virtuales y el empaquetado en KVM [11, 12].

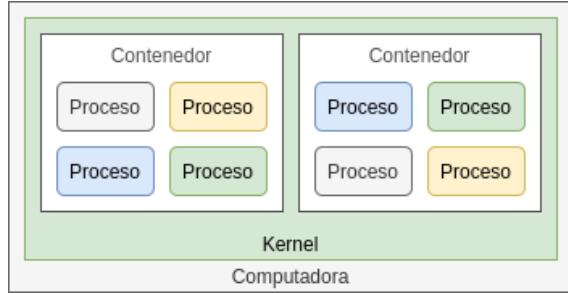


Figura 2.2: Diagrama del contenedor de procesos Docker.

2.2.2. Lenguajes de programación y *frameworks*

Python es un lenguaje de programación de alto nivel, multipropósito, interpretado, que ha sido adoptado por la comunidad de desarrollo como una alternativa a diversos tipos de lenguajes debido a su facilidad de lectura y de introducción en el lenguaje. Es útil para el desarrollo web y de escritorio, además de que ha encontrado una fuerte comunidad en el desarrollo de sistemas de inteligencia artificial y comúnmente en el desarrollo de scripts con diferentes propósitos [13].

JavaScript (JS) es un lenguaje de programación de alto nivel desarrollado originalmente como una forma comprehensiva de desarrollar elementos para sitios web, que ha evolucionado hasta convertirse en una herramienta utilizada para el desarrollo de diversos tipos de proyectos, tales como motores de bases de datos, suites de diseño, entre otros [14]. Este lenguaje es indispensable para el desarrollo de sistemas que requieran de interacciones web, además de ser útil como herramienta complementaria en otros tipos de proyectos.

Debido a la creciente complejidad de los sistemas computacionales, se suelen utilizar *frameworks* o entornos de trabajo que nos permiten interactuar con sistemas de forma estan-

darizada, sin gastar grandes cantidades de tiempo realizando implementaciones propias de operaciones comunes. Los frameworks relevantes para este documento son los siguientes:

- **Masonite ORM** es una solución creada para Python que permite la manipulación de sistemas relacionales de bases de datos creando una interfaz de código. Abstacta la complejidad de la manipulación de base de datos para convertirla en un modelo de clases con una interfaz simple para la edición de los datos. Tiene soporte nativo para transacciones, es compatible con motores de bases de datos tales como MariaDB, PostgreSQL y SQLite, además está diseñado para ser incluido en proyectos complejos sin necesidad de incluir el framework al que pertenece [15].
- **FastAPI** es un framework para desarrollo de APIs REST para Python centrado en el desarrollo rápido con ayuda de las anotaciones estándar de Python. Además de ser uno de los frameworks de desarrollo más rápidos en su ejecución, permite el crear directamente documentación compatible con el estándar OpenAPI sin necesidad de librerías externas [16]. Todo esto lo convierte en un framework ideal para extender proyectos existentes en Python y con su permisiva licencia permite.
- **VueJS** es una tecnología de desarrollo de interfaces para sitios web con un enfoque en el uso de componentes reactivos, lo cual hace menos complejo el manejo y manipulación de datos, así como la actualización de los mismos, debido a que no se tiene que manipular cada elemento manualmente sino que sólo se modifica la información de la que fue originado un elemento para que este sea redibujado [17].

Debido a la complejidad que supone el crear estándares para almacenar la información de forma eficiente para un caso de uso específico de cada proyecto, generalmente se suele optar por el utilizar sistemas de bases de datos previamente existentes. En este caso, se optó por MariaDB, el cual es un motor de base de datos relacional creado por el equipo original que desarrolló MySQL, es un motor que tiene como objetivo mantenerse completamente abierto

y tiene una licencia de uso permisiva para su uso en ambientes comerciales y no comerciales [18]. Tiene un rendimiento similar a MySQL en operaciones transaccionales, por lo cual es una excelente alternativa cuando se requiere un modelo de licencias permisivo [19].

Capítulo 3

Desarrollo del Proyecto

En este capítulo se detallará el proceso de diseño e implementación que se realizó para desarrollar el proyecto de monitoreo de estaciones meteorológicas, así como las limitaciones técnicas que se encontraron durante el proceso.

3.1. Producto propuesto

Se creó un sistema que permite monitorear el estado de los servicios de las estaciones meteorológicas del CECATEV, y por consecuencia de la infraestructura en la que dependen, así como ofrecer un control limitado para la solución de problemas de forma remota.

El proyecto consiste de tres partes independientes:

- Un módulo para el monitoreo del estado de las estaciones, que permite el cargar la información de las estaciones de una base de datos, para luego cargar los controladores específicos de cada estación basado en la información existente de las mismas para después guardar el estado en el que se encuentran en una base de datos.
- Un API REST que tiene el objetivo de proveer un acceso sencillo a la información de los sistemas de monitoreo, así como el de proveer una interfaz de control para las estaciones

que permita la ejecución remota de comandos preestablecidos, desde cualquier punto con la autorización adecuada que haga una petición a la ruta correspondiente.

- Una interfaz gráfica, que permita el acceso a la información correspondiente de los sistemas de monitoreo, así como acceso a los reportes que se generen y permita capturar informes de solución de problemas de las estaciones para su posterior análisis.

3.2. Metodología de desarrollo

Para el desarrollo de este programa, se utilizó la estrategia de desarrollo ágil centrada en el usuario. En ella se combina la metodología de desarrollo ágil, la cuál tiene como características principales la entrega continua de resultados y la preferencia de sistemas funcionales sobre documentaciones de código robustas [20], con el diseño centrado en el usuario, el cuál tiene a los usuarios como objetivo principal para satisfacer las necesidades de requerimientos.

Debido a que la experiencia de usuario es uno de los factores que pueden separar al sistema en desarrollo de los sistemas actuales de monitoreo para equipos de cómputo, el esquema de entregas, desarrollo y planeación estarán centrado en el mismo [21]. El ciclo de entregas será con un sprint de máximo dos semanas, para una revisión de las metas, planeación y objetivos a alto nivel con el usuario y redifinir los requisitos y como sea necesario. La documentación para el usuario final, así como la documentación del API y la información técnica del sistema será un producto que será entregado al finalizar el mismo, apoyándose de la información generada en los sprints.



Figura 3.1: Diagrama de metodología ágil.

3.3. Análisis

Debido a la naturaleza autónoma de las estaciones meteorológicas, y a que el hecho que las mismas se encuentran sometidas a diversos factores que pueden resultar en funcionamiento impredecible, se busca crear un sistema centralizado de recolección de información que tenga gran tolerancia a las diversas condiciones adversas que se enfrentan las estaciones meteorológicas, a la vez que es lo suficientemente confiable para hacer un impacto positivo en la recolección de la información de las mismas.

3.3.1. Conexión a estaciones remotas

La conexión a las estaciones remotas se creó como un sistema modular de conexiones. Teniendo el objetivo de la extensibilidad como objetivo prioritario para el sistema de interacción con las interfaces.

Cada sistema de conexión supone sus propios retos, si bien hay diversos métodos de conexión que podrían ser útiles para la conexión a las estaciones meteorológicas, se decidió enfocarse en la conexión vía SSH a las estaciones meteorológicas que poseen una Raspberry-PI como *datalogger* y como medio de interfaz que se encuentran conectadas por medio de puerto serial a las mismas. Y de las estaciones meteorológicas Campbell, que poseen diversos protocolos de comunicación pero se decidió por utilizar el protocolo HTTP.

Para la conexión a las estaciones RaspberryPi se considera lo siguiente:

- Actualmente cuentan con una VPN configurada para facilitar el acceso a SSH por medio de una dirección IP en el mismo segmento de red que el segmento al que se pretende el servidor final tenga.
- Ocasionalmente, las estaciones meteorológicas perderán acceso a la VPN, ya sea por fallas técnicas del servidor, del ISP, pérdidas de energía eléctrica o demás.

- Que una estación se encuentre fuera de línea de la VPN temporalmente no implica que esta no pueda operar, o incluso que no pueda contactar al servidor, tal como se observa en la Figura 3.2.

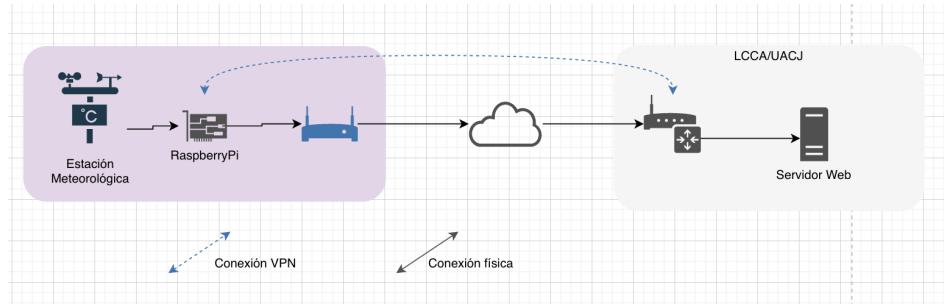


Figura 3.2: Diagrama de la redundancia de las conexiones.

Las estaciones poseen una serie de servicios e información que requiere ser recabada y monitoreada. Si bien, algunas de las estaciones meteorológicas podrían funcionar bajo un plan de datos celulares, los costos actuales de transferencia de datos permiten el no preocuparse tanto por una compresión fuerte de la información transferida.

Los servicios vitales que las estaciones requieren para proporcionar datos de calidad son los siguientes:

- El tiempo de la estación.
- El servicio de monitoreo climático WeeWX.
- Servicio de copias de respaldo, como la base de datos (si aplica).

Todos estos procesos y servicios es posible monitorearlos por medio de comandos Bash en la terminal de la estación, por lo tanto, se considera factible monitorearlos de forma adecuada todos para desarrollar este sistema.

Consideraciones de seguridad

Debido a que generalmente no se crea una red virtual privada separada para el manejo exclusivo de estaciones meteorológicas (ya que estas suelen instalarse sobre infraestructura existente) es importante tener consideraciones de seguridad respecto a el acceso a las estaciones, debido a que pueden ser un punto de acceso a una, de otra forma, red segura e isolada.

Para realizar la conexión a las estaciones meteorológicas se requiere de acceso a la RaspberryPI que funciona como puente entre ambas. Para realizar cambios, crear un servicio, y establecer la información del sistema con una mínima interacción se requiere de un usuario de alta prioridad a la máquina. En el caso del sistema operativo basado en Linux que utilizan las estaciones, es el usuario con la mayor cantidad de procesos *root*.

Al considerarse comprometido el ambiente de la aplicación, se consideraría comprometido el sistema completo. Ya que en este ambiente se encontrarán las contraseñas de acceso a la base de datos y la llave privada que se utiliza para hacer autenticación, si bien existen servicios como AWS-KMS (Key Management Service), el implementar un sistema tan robusto para la administración de secretos sale del alcance de este proyecto.

Por lo tanto, se decidió crear un servicio que tome un usuario y password con acceso *root* de forma temporal (o al menos uno que tenga permisos de *sudoer*) y utilizarlo para almacenar la llave pública local del servidor para realizar operaciones sin tener un par de usuario y contraseña almacenados en la base de datos que pudiera ser comprometido. De esta forma, se mitiga el impacto de una posible intrusión a la base de datos, y en caso de comprometerse, sólo es necesario actualizar la llave privada y públicas.

3.3.2. Selección de herramientas

Para el desarrollo de el proyecto, se decidió por realizar la aplicación de servidor con el lenguaje de programación Python. Esto debido a que otros proyectos de los que depende el funcionamiento de los sistemas de el CECATEV, tales como el monitoreo climatológico y meteorológico con Weewx, y el proyecto para obtener la información de las estaciones meteorológicas en diferentes estándares [22] fueron realizados con este lenguaje. Además, cuenta con una librería estándar extensa así como una librería de terceras partes madura que permite el desarrollo de forma eficaz utilizando estas librerías existentes, contando con la certeza de que están listas para un proyecto con niveles de producción.

El *ORM* utilizado para el desarrollo de la aplicación, fué *MasoniteORM*, un ORM para Python que tiene como objetivos principales la simpleza y extendibilidad de proyectos. Si bien, *MasoniteORM* es parte de *Masonite*, un framework para el desarrollo de aplicaciones web, este es bastante extenso y complejo, y si bien es fácilmente extensible no es simple para usarse. Por lo tanto, se decidió utilizar el framework de desarrollo de aplicaciones web *FastApi*, creado por Sebastián Ramírez (tiangolo), ya que ofrece una forma fácil de crear un *API* web, el cuál será utilizado posteriormente para el desarrollo de una interfaz fácilmente accesible para los usuarios.

La interfaz gráfica para proveer acceso a la información a los usuarios se decidió hacer en una aplicación web. Esto debido a que las interfaces web ofrecen una amplia y madura plataforma desarrollo que se puede acceder desde diferentes tipos de dispositivos, así como una gran variedad de *frameworks*, metodologías, y paradigmas, lo que ofrece una gran flexibilidad al momento de realizar un desarrollo a la medida. También está el hecho de que debido a la naturaleza del proyecto, un sistema que centraliza toda la información que es accesible por medio de un API, no parecía posible que un proyecto de una aplicación de escritorio o una aplicación web ofreciera una ventaja que no ofreciera la interfaz web.

Para el desarrollo de la interfaz web se optó por el desarrollo con los *frameworks* para desarrollo de aplicaciones web *VueJS* y *TailwindCSS*. Se seleccionó *VueJS* por la cualidad reactiva que posee, la cual permite desarrollar sistemas de monitoreo de información que requieren una constante actualización de los datos sin afectar la experiencia de usuario.

Por el motivo de hacer el proyecto lo más estándar, fácilmente accesible y mantenible, se decidió realizar la programación y documentación del proyecto en inglés, pero manteniendo las interfaces en las que el usuario interactúa con el mismo en español. Además, se eligió el configurar *pylint* con el estándar *pep8* para el formato automático de el código del proyecto en el estándar. De la misma forma, se configuró *ESLint* en el proyecto de *VueJS* para el fronted, extendiendo los estándares *vue:essential* y *eslint:recommended*, para realizar el formato automático en los archivos.

3.4. Diseño

Conforme a la metodología de desarrollo centrada en el usuario utilizada en este proyecto, se decidió empezar por el desarrollo de un prototipo de alta fidelidad para el diseño del sistema, para posteriormente continuar con el diseño de la infraestructura y los sistemas necesarios para entregar el proyecto, con el objetivo de crear un sistema integral que pudiera satisfacer las necesidades del CECATEV al mismo tiempo que cumpliera con los requisitos técnicos necesarios para cumplir los objetivos propuestos.

3.4.1. Prototipado de la interfaz gráfica

Para el desarrollo de un prototipo rápido de la interfaz gráfica, se utilizó el software *Figma*, esto debido a que es un software gratuito, sencillo de utilizar orientado al prototipado de interfaces de alta fidelidad. *Figma* ofrece la capacidad de integrar diferentes *frameworks* de diseño de interfaces, para crear un sistema de diseño coherente, consistente a través de todo un proyecto. Utilizando este software se decidió utilizar una plantilla de un tablero de control para una plataforma genérica.

Utilizando esta plantilla como base para el lenguaje de diseño de la aplicación, se realizó un prototipado de la interfaz propuesta para evaluar la posible utilidad de la misma, creando un prototipo inicial de alta fidelidad. La primer interfaz en prototiparse fue el tablero de control, después de algunas iteraciones con cambios menores, el prototipo quedó tal como en la Figura 3.3, cabe notar que esta interfaz gráfica tiene elementos en la barra lateral que no corresponden con el proyecto (tal como la sección de chat y otros similares), esto debido a que se dejaron ciertos elementos predefinidos de la plantilla original, para no romper con la estética del diseño.

En este prototipo de interfaz se tomaron en cuenta en varios factores, uno de ellos, es que una estación que es monitoreada puede tener un error de conexión que no permita el

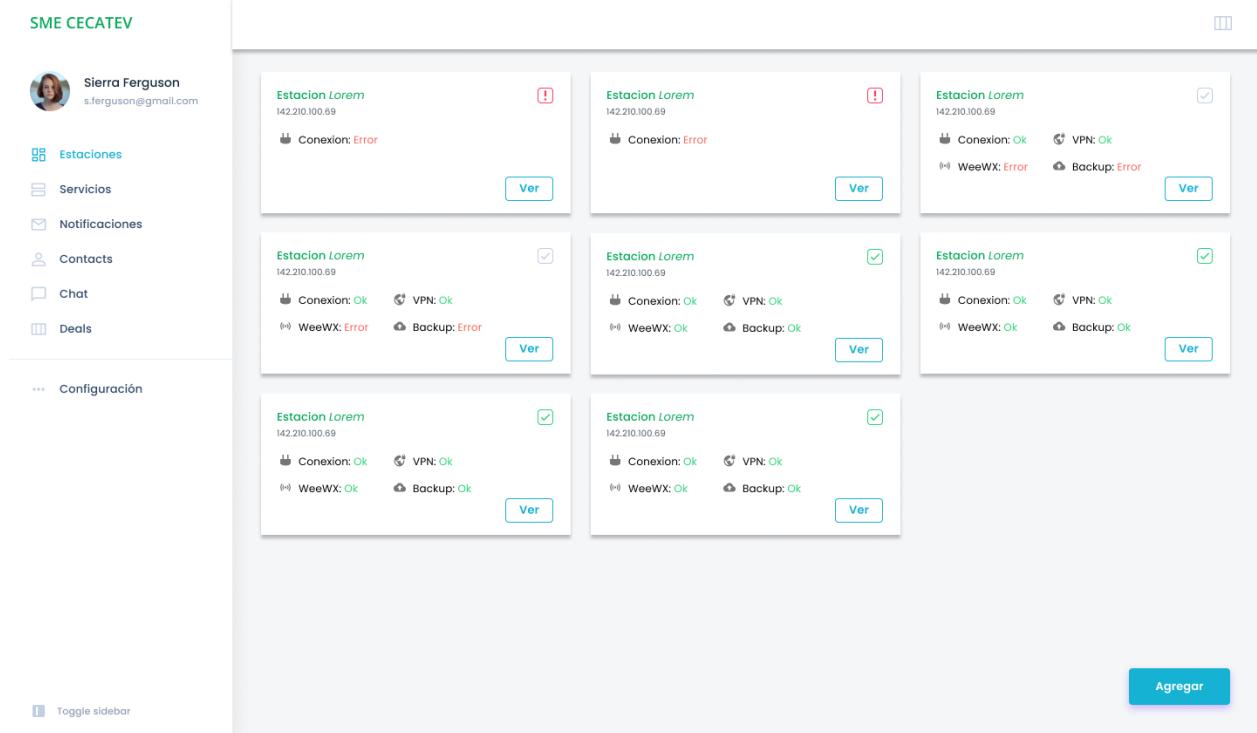


Figura 3.3: Prototipo del tablero de control.

acceso a la misma, y que al encontrarse con un error de conexión, no es posible obtener el estado de los servicios. De ser así, la información de estos servicios no se muestra. También se tomó en cuenta el identificar las estaciones por un nombre particular, pero también tener presente la dirección IP de la misma (en caso de poseer una) para volver más sencillo el acceso técnico a la información en caso de ser necesario para un usuario técnico. Además, se hizo la consideración de tener una variedad de diferentes servicios que se podrían monitorear, y que el estado de los servicios y de las estaciones fuera independiente.

La interfaz para agregar una nueva estación al sistema contiene la información elemental que se requiere para registrar una nueva estación. En este caso, se compone de una dirección IP, un usuario, el método de conexión a la misma, el tipo de estación y el dispositivo por el que se accede a la misma. El prototipo realizado se puede observar en la Figura 3.4.

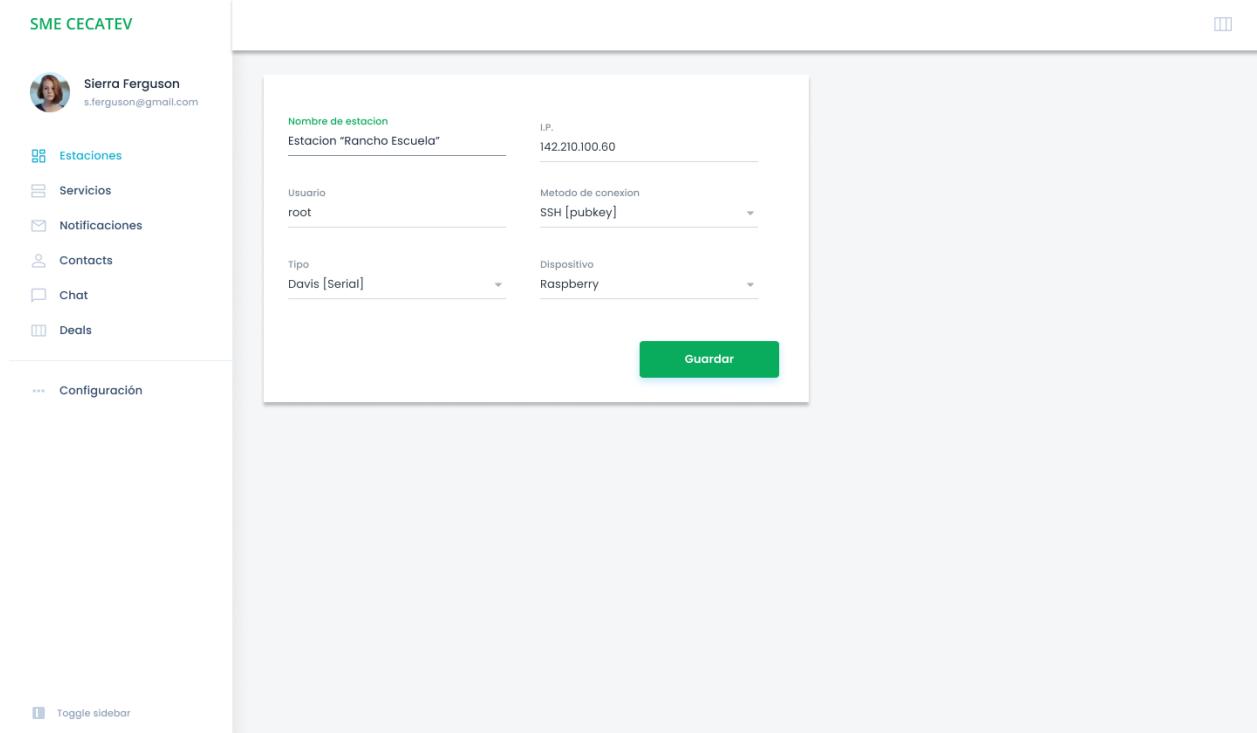


Figura 3.4: Prototipo de la interfaz para agregar una nueva estación.

Posteriormente se creó el componente de la tercera parte importante del monitoreo de las estaciones meteoreológicas, el prototipo de la interfaz de solución de problemas de las estaciones. Al ser un objetivo secundario importante el poder tener y acceder a la información que las estaciones meteorológicas generan, se considera igualmente importante el poder capturar una razón de la solución de los inconvenientes para futuros análisis. Esta información será capturada con la ayuda de una interfaz como la de la Figura 3.5.

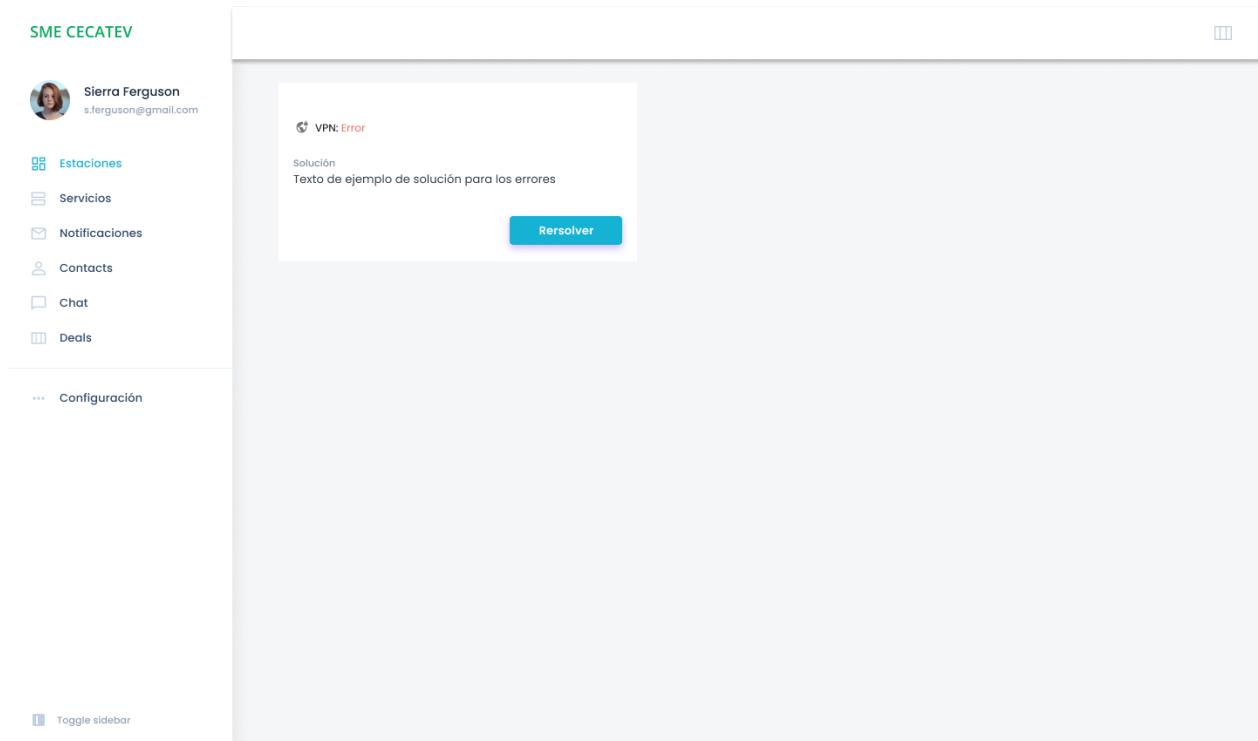


Figura 3.5: Prototipo de la interfaz de solución de errores.

3.4.2. Diseño de logo e identidad gráfica

Los colores utilizados para el desarrollo de los prototipos, y posteriormente para el desarrollo de la totalidad del proyecto fueron; el color verde *Go Green*, con el código hexadecimal #09AB5D como color principal de la interfaz, debido a su relación con el diseño actual del sitio de consulta de la información de las estaciones meteorológicas, y el color azul *Battery Charged Blue* con el código #16B2D4 como secundario por su contraste con el color verde y por su actual uso en el sitio existente de CECATEV.

El nombre del proyecto, surge de la combinación del griego *meteoro* (que se encuentra en la atmósfera o en el cielo) y la palabra *monitoreo*. Esto da como resultado la palabra *meteoreo*. Este nombre fue utilizado para referirse a diversos elementos del proyecto de forma interna, tales como carpetas y archivos, así como para logos y demás documentación que se tuviera que generar del sistema.

Con la ayuda de Alan J. Ovalle L., un diseñador con experiencia en proyectos de software, se procedió a limitar y diseñar un logo para el sistema. El logo tiene dos elementos principales. Una nube, significando la naturaleza centralizada y de recolección de información del sistema; y la veleta de un sensor de viento, esta es utilizada como una flecha para dar un sentido de orientación, además de ser una referencia directa a las estaciones meteorológicas.

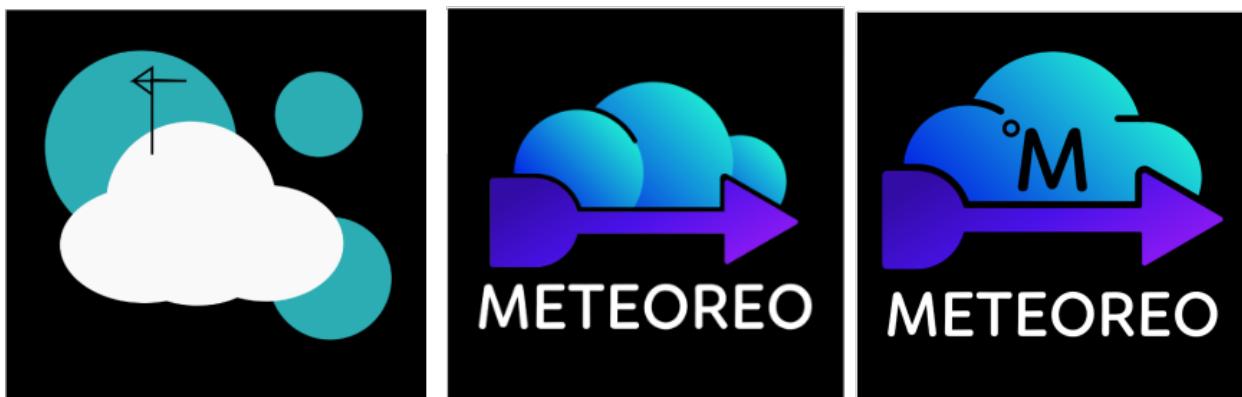


Figura 3.6: Evolución del logo de Meteoreo, ordenado cronológicamente.

3.4.3. Diseño de base de datos

Para el diseño de base de datos del sistema se consideraron dos elementos como relevantes, los datos de las estaciones meteorológicas, que incluyen datos como la conexión a las estaciones y la forma en la que se accederá a ellas, y los eventos que las mismas generen.

Si bien es posible almacenar la información de el estado de las estaciones en una *time series database*, en la que la información del estado de cada uno de los servicios y elementos que se monitorean es almacenado sin tomar en consideración el estado de los mismos, no se considera necesaria la información de los sistemas en su estado funcional, esto debido a que el volumen de datos generado, $O(n*t*s)$, donde n es el número de estaciones, t la cantidad de veces que la información se consulta por hora y s el número de servicios que se consultan, es demasiado grande en comparación a la utilidad que se podría obtener de los datos generados.

La información más relevante que se podría obtener de acuerdo a los objetivos de este proyecto y las necesidades del laboratorio, es el estado de las estaciones en caso de falla, e información detallada sobre los estados del evento con el objetivo de incrementar la calidad recabada de los datos.

Debido a esto, la base de datos se diseñó con el objetivo de almacenar la información en forma de eventos, y tomando en cuenta la utilidad futura de auditoría se decidieron agregar tiempos de creación de eventos y de la solución de los mismos. Además, para ayudar a preservar una mayor consistencia de los datos, se hizo uso de los *borrados suaves*, una metodología de manejo de la información en la que los registros físicos no son removidos, sino que son desactivados lógicamente.

En cuanto al manejo de la información adicional de las estaciones meteorológicas, es decir, la información de las mismas que no es indispensable para el funcionamiento del sistema pero es necesaria para controles internos, se agregó una tabla de atributos extra. Esta tabla contiene información diversas de las estaciones, y consiste en la forma *llave ->valor* para los

campos, esto, para asegurar la mayor flexibilidad posible de los datos y su almacenamiento. Sacrificando velocidades de indexamiento y procesamiento por una mayor libertad para extender y modificar el sistema.

Si bien habría sido posible el crear un campo extra en formato *JSON* para almacenar la información extra de las bases de datos, estos datos habrían sido enviados por defecto en cada consulta a la base de datos incrementando significativamente el tráfico a la base de datos, sin mencionar la complejidad que agrega en ORM convencionales el hacer uso de este tipo de dato.

Otra consideración que se tomó para el diseño de la base de datos, es el crear una estructura en la que fuera posible migrar datos de una instancia a otra del proyecto, para que dado el caso que se requiriera el cambio de instancia de unos datos (por ejemplo, en el caso de un cambio de posesión del equipo) pudiera ser mantenida la información recolectada de las estaciones. Para esto, se utilizó un *uuid* como llave principal de la tabla de estaciones, lo que permite migrar la información de un sistema sin tener que realizar cambios al identificador único de las estaciones, que suele ser un dato de tipo entero sin signo autoincremental.

Con todas estas consideraciones en mente, se creó una base de datos que corresponde con los contenidos que se muestran en el modelo entidad-relación que se puede observar en la Figura 3.7. Para el diseño de esta base de datos, se optó por utilizar una normalización orientada al costo beneficio, sin perder la calidad de los datos obtenidos. Por ello se realizó un análisis informal con base en decisiones lógicas para la organización de la información en formas normales [23].

Las relaciones de la información después de un proceso de normalización quedaron de la siguiente forma:

- De uno a muchos, de la tabla de datos de estaciones a los campos adicionales que pudieran ser requeridos.

- De uno a muchos, de la tabla de estaciones a una tabla de eventos que pudieran presentar estas estaciones.
- De uno a muchos, de la tabla de soluciones o comentarios a la tabla de eventos.
- De uno a uno, de la tabla de eventos a la tabla de soluciones.

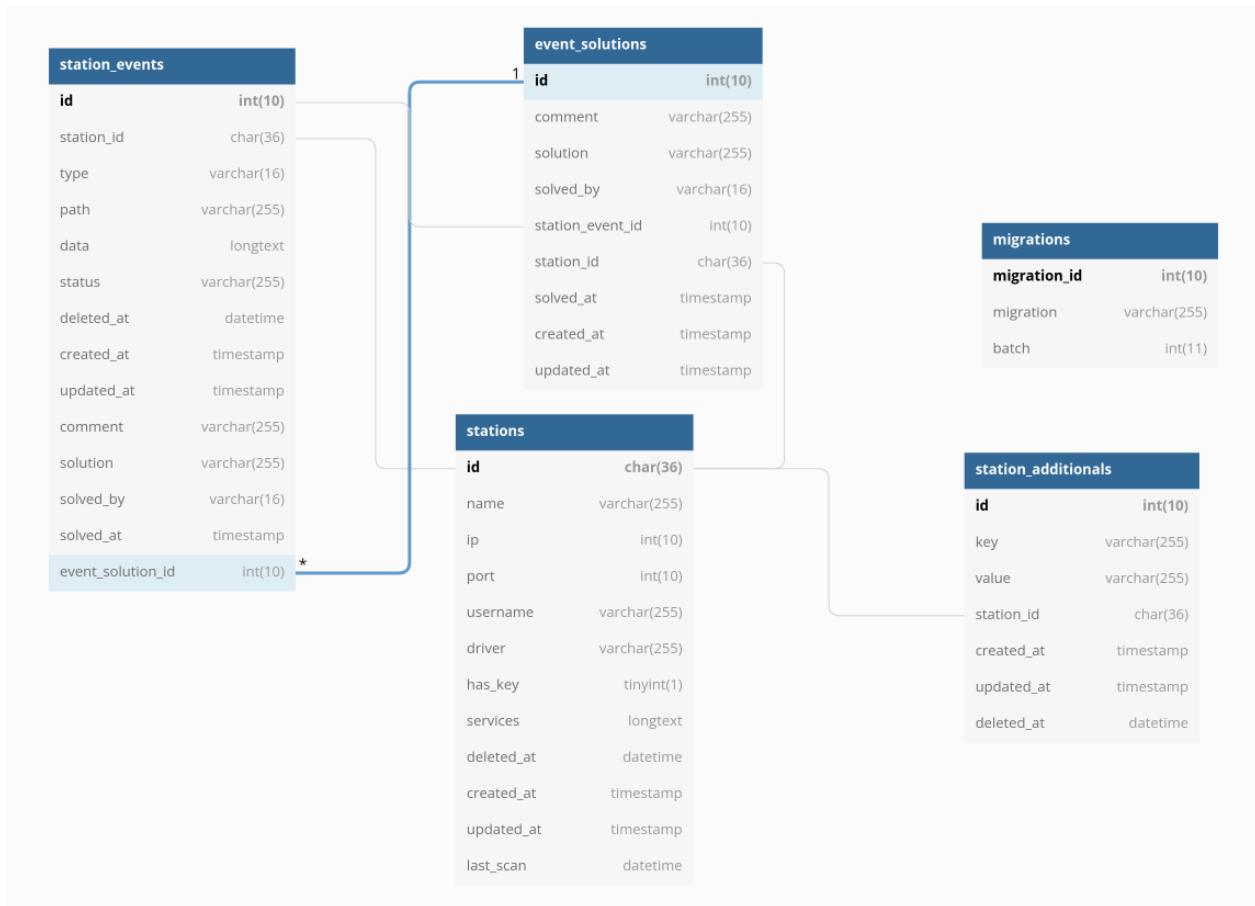


Figura 3.7: Modelo entidad-relación del proyecto meteoreo.

3.4.4. Configuración de ambiente de desarrollo

El ambiente de desarrollo que se seleccionó para la sección de Python del proyecto fué seleccionado con el objetivo de proveer la mayor flexibilidad y portabilidad a corto y largo plazo, haciendo fácil la modificación posterior del proyecto por los que no estuvieron involucrados inicialmente, y sencillo de replicar para futuras investigaciones. Para lograr estos objetivos, se optó por realizar el proyecto con ayuda de las tecnologías de Docker, debido a que permite realizar imágenes de proyectos de forma sencilla, y permite hacer contenedores multiplataforma que con una mínima configuración se vuelven útiles para el desarrollo.

Debido a la sencillez que el sistema de cnfiguración de contenedores *docker compose* ofrece, se eligió para almacenar los parámetros de configuración de los contenedores en vez de crear comandos compatibles con Docker para ello. Esto permite una fácil edición de los servicios y la aplicación de los mismos de una forma estandarizada que permite una comprensión más eficaz de los parámetros y de las dependencias.

El archivo de configuración fué almacenado en la raíz del proyecto, con el nombre de `docker-compose.yml` tal como el estándar de la utilería `docker-compose` sugiere, y este archivo tiene el contenido siguiente que se muestra en el Listado 3.1. En este archivo, se especifica que se requiere de un servicio de MariaDB, el cuál fué utilizado para corroborar que el desarrollo era de utilidad con el motor seleccionado para producción, así como una referencia al archivo *Docker* en el que se tiene el contenedor que se utilizará para el desarrollo. Además, se hace referencia a algunas variables, como `MYSQL_DATABASE`, que son obtenidas de un archivo `env` estandarizado en la raíz del proyecto.

El archivo de *docker-compose* tiene una dependencia con un archivo de Docker, que se pretende que facilite la adición de librerías adicionales al proyecto en la posteridad. Actualmente, extiende la imagen existente de Sebastián Ramírez "tiangolo", el proyecto *uvicorn-unicorn-fastapi*. Esta imagen fue utilizada como base debido a su increíble flexibilidad para

```
version: '3.3'
services:
  api:
    container_name: meteoreo-api
    build:
      context: ./  
      dockerfile: Dockerfile
    # ...
    # Configuración del ambiente
    # ...
  networks:
    - meteoreo-backend
  mariadb:
    image: mariadb
    container_name: meteoreo-mariadb
    environment:
      MYSQL_DATABASE: '${MYSQL_DATABASE}'
      MYSQL_ROOT_PASSWORD: '${MYSQL_ROOT_PASSWORD}'
      MYSQL_PASSWORD: '${MYSQL_PASSWORD}'
      MYSQL_USER: '${MYSQL_USER}'
      SERVICE_TAGS: dev
      SERVICE_NAME: mysql
    ports:
      - '3306:3306'
  networks:
    - meteoreo-backend
  restart: always
networks:
  meteoreo-backend:
    driver: bridge
```

Listado 3.1: Archivo docker-compose.

el desarrollo de proyectos en FastApi, sus optimizaciones automáticas para el balanceo de cargas entre diversos procesos creados automáticamente (ya que python es monoproceso) y además, por ser una imagen altamente mantenida por la comunidad, debido a su popularidad. En este archivo también se especifica el instalar la librería `inetutils-ping` debido a que el proyecto dependerá de realizar pruebas por `ping` para revisar la conectividad con las estaciones antes de intentar realizar una conexión y la imagen base no tenía esta librería, el archivo final quedó como se muestra en el Listado 3.2.

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7
```

```
# Installs lib to do pings from the server
RUN apt-get update && apt-get install -y \
    inetutils-ping \
    && rm -rf /var/lib/apt/lists/*
CMD [ "/start-reload.sh" ]
```

Listado 3.2: Archivo Dockerfile.

El editor de código seleccionado para el desarrollo del proyecto es Visual Studio Code, el cual posee una gran extensibilidad y predeterminados sensibles que permiten configurar el ambiente de trabajo de la forma que más sea conveniente para el desarrollo del proyecto, además provee la funcionalidad de *devcontainers*, los cuales son parte de una extensión que permiten el crear ambientes de desarrollo dentro de ambientes virtuales en docker, utilizando las herramientas instaladas en la imagen de docker y que no requieren de configuración adicional por parte del desarrollador para comenzar a trabajar en un proyecto. Al detectar un archivo `devcontainer.json`, esta extensión automáticamente informa al desarrollador de su existencia y le invita a iniciar su ambiente de desarrollo utilizando los parámetros definidos en el archivo.

En este archivo se especifica un nombre para identificar el ambiente de desarrollo que sea

reconocible por el desarrollador, la localización del archivo que describe el contenedor, y una lista de extensiones para el editor de código. Entre las más importantes se encuentra *pylance* que permite realizar el formato automático de código con *pep8* y *magicpython* una adición al editor de código que provee un motor de autocompletación para python, las demás siendo preferencias personales útiles para agilizar el desarrollo del proyecto.

```
{  
  "name": "Meteoreo API",  
  "service": "api",  
  "remoteUser": "root",  
  "shutdownAction": "stopCompose",  
  "workspaceFolder": "/app",  
  "dockerComposeFile": "../docker-compose.yml",  
  "extensions": [  
    "editorconfig.editorconfig",  
    "mikestead.dotenv",  
    "njpwerner.autodocstring",  
    "aaron-bond.better-comments",  
    "mhutchie.git-graph",  
    "hookyqr.beautify",  
    "magicstack.magicpython",  
    "gruntfuggly.todo-tree",  
    "ms-python.vscode-pylance",  
    "sleistner.vscode-fileutils"  
  ]  
}
```

Para el desarrollo de la sección de la interfaz de web del proyecto, se instaló en la máquina de desarrollo NPM versión 14.8.1, debido a que era la última versión con soporte a largo plazo (LTS por sus siglas en inglés) disponible, y el sistema de manejo de dependencias *yarn* debido a las ventajas que ofrece sobre *npm*, tales como mayor velocidad de instalación de paquetes y caché multiproyecto. No se vió como un elemento necesario el intergrar Docker o

algún otro tipo de tecnología de contenedores para el desarrollo, debido a la ubicuidad de las herramientas y la simpleza de instalación y de mantenimiento de las mismas en un ambiente común compartido.

3.5. Desarrollo

Después de haber realizado el análisis inicial de el alcance del proyecto y las necesidades de los usuarios, se comenzó con el desarrollo del proyecto. Este desarrollo se hizo en tres partes. Primero, el desarrollo de un módulo de monitoreo de las estaciones meteorológicas por medio de drivers, después un API como intermediaria entre la información almacenada en la base de datos y una interfaz gráfica para el monitoreo eficaz.

3.5.1. De la base de datos

Debido a que se seleccionó un sistema basado en un ORM para el manejo de la base de datos, esta tiene que modelarse en el sistema en forma de código para ser reconocida, de la misma forma, la creación de la estructura de la base de datos en el motor se hará por medio de migraciones creadas con el sistema de creación de base de datos, para facilitar su mantenimiento e interoperabilidad en diferentes sistemas y facilitar las integraciones con otros sistemas existentes de información.

Los archivos resultantes de estas migraciones fueron tal como se muestran en el Listado 3.3, las cuales se pueden encontrar en la ruta `/app/database/migrations` del proyecto, tal como es especificado en las guías de desarrollo de MasoniteORM.

Si bien los modelos creados para la ejecución de este proyecto siguen los estándares de modelado de base de datos especificados en MasoniteORM, tal que al modelar la base de datos es posible obtener el mismo código que el implementado, una modificación importante al modelado de los datos es el uso de un mutador para traducir las direcciones IP a enteros y viceversa. Este mutador es accesado con un nombre alternativo al nombre del campo en el modelo, debido a que por limitaciones del ORM no es posible utilizar mutadores y accesores con el mismo nombre del campo objetivo. En el Listado 3.4 es posible observar el mutador mencionado y su definición, cabe resaltar que la presencia de los caracteres [...] indican

```
2021_08_03_052340_stations.py  
2021_10_14_162126_station_additional.py  
2021_11_13_020934_event_solutions.py  
2022_04_03_003436_add_timestamps.py  
2021_08_16_000028_station_events.py  
2021_10_29_035514_stations.py  
2022_03_25_013638_normalize_events_comments.py  
2022_04_20_222630_events_fix.py
```

Listado 3.3: Archivos de migración en el proyecto.

que existe más código con otras funcionalidades que se omitió por fines de brevedad.

Selección del motor de base de datos

Para el caso de uso del centro de monitoreo de estaciones meteorológicas de la UACJ, en el que la red actual cuenta con 13 estaciones, no es necesario considerar como cuello de botella el motor de base de datos que se utilizará para el sistema. Esto debido a que, con un tiempo mínimo para la consulta del estado de las estaciones de hasta 5 minutos entre consultas, el sistema podría funcionar incluso con un tiempo promedio de 23 segundos desde la consulta hasta el almacenamiento de la información. Esto, sin tomar en cuenta que es posible parallelizar el proceso de consulta y generación de eventos de las estaciones meteorológicas, por lo que no se considera como algo relevante la selección de un motor de base de datos que cuente con alto rendimiento de lectura y/o escritura de la información.

Debido a que la infraestructura del sistema de las estaciones meteorológicas ya utiliza un motor relacional de base de datos adecuado para el proyecto, MySQL, se pretende utilizarlo para este proyecto, reduciendo la carga de mantenimiento para el equipo de la universidad, además de un sistema familiar que permitirá a los involucrados realizar consultas a la información sin necesidad de aprender nuevas tecnologías.

```
import ipaddress
[...]
class Station(Model, UUIDPrimaryKeyMixin, SoftDeletesMixin):
[...]
def get_ip_address_attribute(self):
    return str(ipaddress.ip_address(self.ip))

def set_ip_attribute(self, attribute):
    try:
        ip = ipaddress.ip_address(attribute)
    except ValueError:
        raise ValueError("Invalid IP Address %s" % attribute)

    return int(ipaddress.ip_address(attribute))
```

Listado 3.4: Definición de accesos y mutadores para modelo de estaciones.

Para esto, se utilizó la flexibilidad que ofrecen los sistemas modelado de objetos y roles (ORM, por sus siglas en inglés) [24], en la que se permite el crear sistemas agnósticos de un motor de base de datos en específico, y la creación de modelos, esquemas y relaciones de base de datos se dejan al *framework* de modelado de datos. Esto además ofrece soporte para migraciones para realizar actualizaciones de base de datos controladas en caso de requerir extender un sistema existente.

El motor de base de datos seleccionado para el desarrollo local del proyecto fué *SQLite*, debido a la flexibilidad que ofrece al ser una base de datos que sólo depende de un archivo para funcionar y que no requiere de la instalación de paquetes de software extra en la estación que se utiliza para desarrollar y probar el proyecto.

3.5.2. Del módulo de monitoreo de las estaciones

Para realizar el monitoreo a las estaciones meteorológicas, se decidió dividir el módulo en dos componentes principales, un submódulo de generación de reportes y un sistema de controladores que contuvieran el código de conexión y restauración de servicios de las estaciones.

Tomando como referencia el proyecto de *Monitoring Plugins* [4], el cual es compatible con diversos proyectos especializados en monitoreo de sistemas de alta resiliencia, tales como *Nagios* y *Icinga*, se decidió crear un proyecto basado en drivers que pudieran ser extendibles. Cada uno de estos drivers, puede cargar una cantidad n de módulos o servicios, que contienen la información necesaria para obtener el estado de la estación meteorológica y generar un reporte que contenga información comprehensiva del estado de las estaciones.

Con el objetivo de crear un sistema que fuera posible integrar en diferentes ambientes y no requiriera de previa instalación de componentes extra en el dispositivo objetivo, se decidió que la técnica con cual se revisaría el estado de las estaciones es por medio de un comando que se ejecutaría en una estación remota, o de forma local dado el caso, y se comparara la respuesta obtenida con el resultado de la operación. En caso de que la respuesta de esta operación sea diferente a la respuesta esperada, se buscaría en un arreglo de casos conocidos, que pueden ser solucionados con la ejecución de un comando. Debido a esto se utilizó una estructura de

Debido a que es necesaria más información que sólo el comando que se requiere ejecutar en una estación, se decidió crear una estructura basada en servicios para el monitoreo de las estaciones. Además, de esta forma, un controlador de una estación podría reutilizar fácilmente servicios que fueran utilizados en controladores para el monitoreo de otras estaciones, tal como es el estado de la base de datos y otros similares.

La estructura del servicio resultante es un objeto en el lenguaje Python con la forma que

```

service = {
    "command": "", # Comando a ejecutar
    "stdout": "", # Salida esperada
    "stderr": "", # Error esperado
    "actions": {
        "read_write_enabled": {
            "response_stdout": "", # Si la respuesta del comando previo es esta
            "response_stderr": "", # Si el error del comando previo es este

            "description": "", # Descripción para el usuario del error
            "solution": "", # Solución propuesta, si existe, para el usuario

            "command": "", # Comando a ejecutar
            "stdout": "", # Salida esperada
            "stderr": "", # Error esperado
            "actions": {
                # ...
            }
        }
    }
}

```

Listado 3.5: Ejemplo de estructura de un servicio.

se puede apreciar en el Listado 3.5, donde el arreglo de casos conocidos tiene el nombre de *actions*, y es anidable, lo que permite listar una serie de comandos que pueden ayudar a la solución de problemas con una profundidad *n*.

Para el desarrollo de la estructura del controlador de monitoreo de las estaciones meteorológicas, se optó por crear un sistema orientado a la extensión de un componente base que fungiera como sistema principal de ejecución, verificación de credenciales y ejecución de comandos en el sistema objetivo. La estructura de los controladores resultante es que

cualquier módulo puede contener una serie de controladores relacionados, y cada uno de estos controladores extiende un driver base para hacer uso de diversas funciones, como el obtener información o registrar la estación. La estructura implementada resultante puede ser observada en la Figura 3.8.

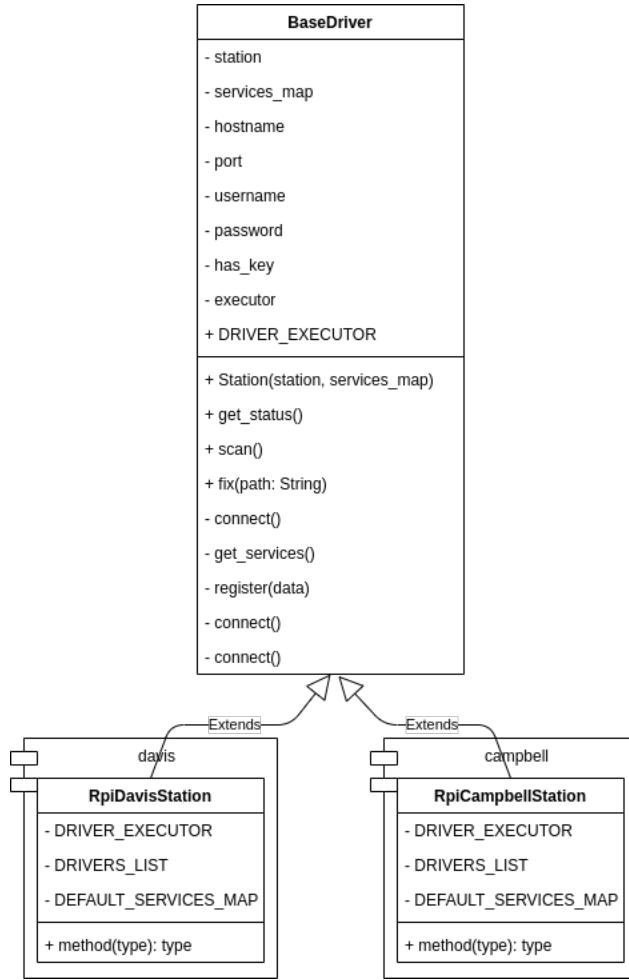


Figura 3.8: Diagrama de clase de controladores

Debido a que este diseño de controlador requiere de ser instanciado cada vez que se busca obtener la información más reciente, esto permite el crear servicios más complejos que la estructura mencionada anteriormente, un ejemplo es la implementación del servicio de control de la fecha y la hora de la estación, que impide que esta sea desvío por más de 15

```
import time

def service():
    # Gets and compares the datetime against the current time in bash
    command = 'CURRENT=$(date +%s); MINUTES=$(( ({0} - $CURRENT) / 60
    ↵ ));'.format(int(time.time()))

    # Checks if the obtained minutes is greater than 15 minutes, prints true
    ↵ if it is.
    command += 'if (( $MINUTES > 15 || $MINUTES < -15 )); then echo "true
    ↵ $MINUTES"; else echo "false $MINUTES"; fi'

    service = {
        "command": command,
        [...]
    }

    return service
```

Listado 3.6: Ejemplo del servicio para revisión de tiempo.

minutos de la hora del sistema de monitoreo, para asegurar la calidad de los datos recabados. En este caso, se hace uso de la librería `time` de Python para obtener la hora actual en formato Unix, para después compararla por medio de bash con la hora del sistema objetivo.

La implementación de el servicio mencionado anteriormente puede ser observada en el listado 3.6, sin embargo, cabe notar que es posible escribir otro tipo de estructuras de servicios, tales como clases, funciones como en este caso, e incluso ser generado con una base de datos.

Para poder estandarizar la ejecución de los comandos en diversos tipos de sistemas, se creó un módulo que por medio de interfaces estandariza la ejecución, si es proveído con la información necesaria para realizar la tarea. Para esto, se creó una clase de ejecutor base de

la cual surgen un ejecutor local y uno remoto, el último de estos requiriendo una serie de elementos, como credenciales y un *host* destino para su funcionamiento. El diagrama de clase que representa el código resultante puede ser visto en la Figura 3.9.

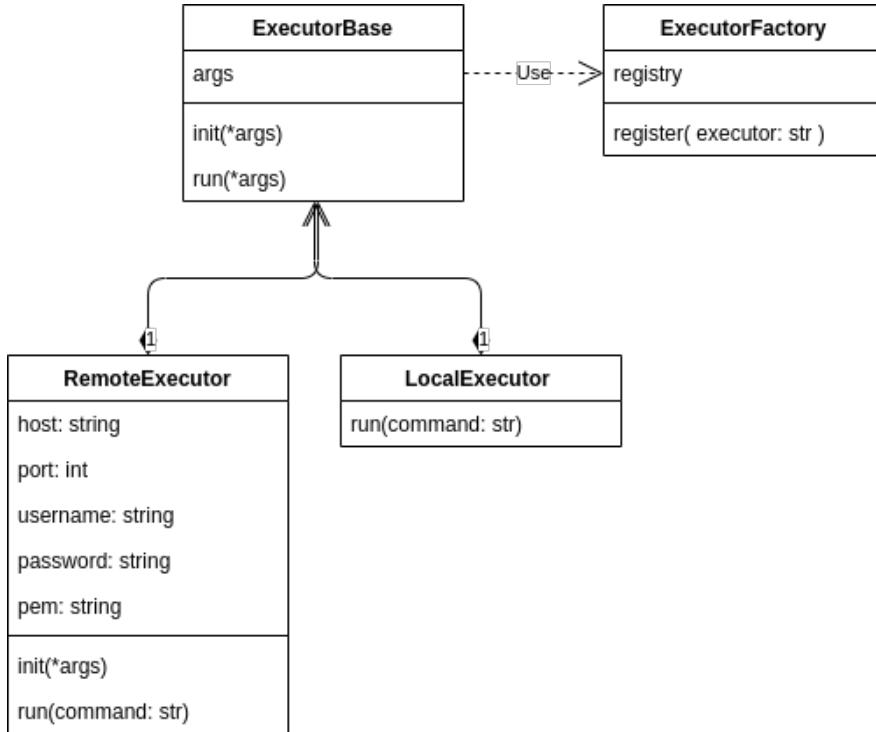


Figura 3.9: Diagrama de clase de ejecutores.

Esta implementación puede facilitar la creación de más ejecutores para casos de uso específico, como un ejecutor para un sistema que dependa de la comunicación por puertos seriales, gracias a la capacidad de registrar nuevos ejecutores en la fábrica sin necesidad de reescribir el código existente.

El módulo de monitoreo tiene como objetivo principal el observar la información obtenida por los diversos controladores y generar reportes conforme sea adecuado. Durante este proceso, se toman los datos de una estación y se utilizan para crear una conexión a la estación para obtener la información con un controlador. Este controlador utiliza un ejecutor para enviar los comandos en la definición de los múltiples servicios a la estación, y regresa un arreglo

estandarizado que contiene la información de los problemas que se encontraron al realizar el proceso, tal como se muestra en la Figura 3.10.

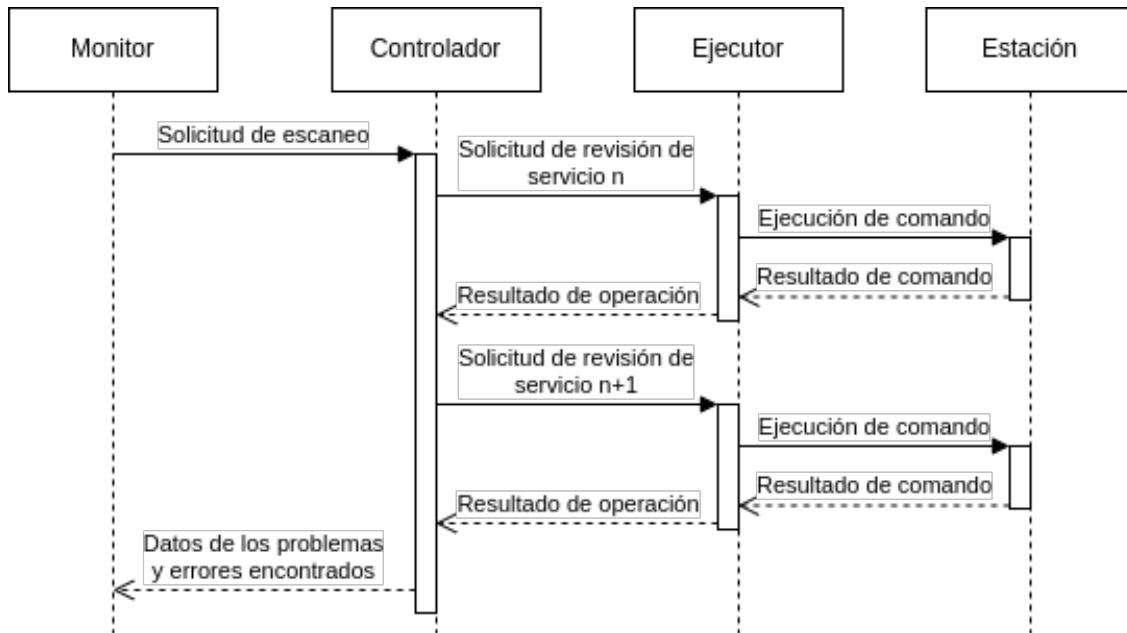


Figura 3.10: Diagrama de secuencia para el monitoreo de las estaciones.

El módulo de monitoreo después toma la información obtenida para ser guardada en base de datos y generar los eventos conforme a la configuración del proyecto. Cabe notar que sólo se recaba la información de los servicios que han sido detectados con algún problema, esto se hace para cumplir con el objetivo de maximizar el espacio de la base de datos discriminando la información que no se considera relevante.

Cuando uno de estos eventos es generado, se al sistema de notificaciones que se integró. Este sistema de notificaciones consiste de un archivo de configuración para obtener las credenciales necesarias para realizar operaciones, así como una clase interface que instancia el SDK del servicio seleccionado, en este caso OneSignal. La elección de OneSignal se debió a su previo uso en la Universidad Autónoma de Ciudad Juárez, así como por la existencia de una librería estándar en el lenguaje para interactuar con el servicio.

Estos procesos además de ser documentados en el presente documento, fueron documentados en el código con el estándar de *docstrings* de Python, debido a esto fué posible generar una documentación automática del proyecto para una fácil navegación. Esta documentación fué generada con la librería *pdoc* y la documentación que puede ser observada en la Figura 3.11, fué generada ejecutando el comando `pdoc generate -html` en la estación de desarrollo.

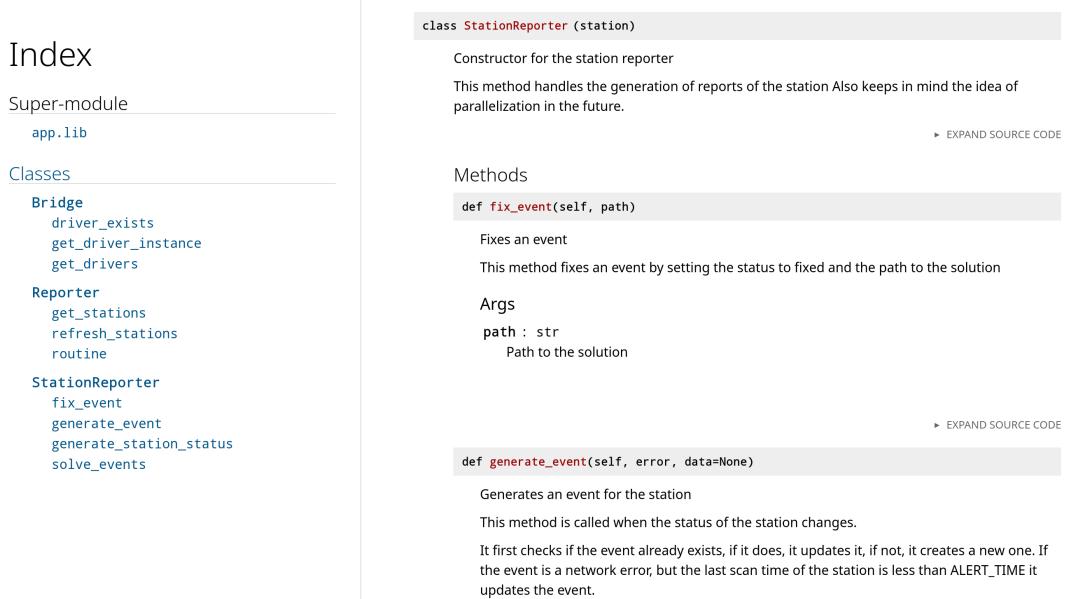


Figura 3.11: Captura de documentación de librería.

Desarrollo de los servicios de monitoreo

Un servicio fundamental para la longevidad de las estaciones meteorológicas, es el de RoPI, este programa convierte la tarjeta SD de la estación meteorológica en un disco de sólo lectura, lo que reduce el estrés que esta sufre al estar funcionando y reduzca el riesgo de corrupción. Para el monitoreo de este servicio, se analizó el código de el servicio y se copió la misma instrucción que utiliza la herramienta para verificar el estado del sistema. En caso de que este servicio no esté funcionando, sólo se requiere reactivar el modo sólo lecutura. El servicio de monitoreo resultante es el que se puede observar en el Listado 3.7

```
service = {  
    "command": "echo $(awk '/root/{print $4}' /proc/mounts | awk -F ,  
    ↳ '{print $1}')",  
    "stdout": "ro",  
    "stderr": None,  
    "actions": {  
        "read_write_enabled": {  
            "description": "La estación está en modo escritura",  
            "solution": "Reactivar el modo sólo lectura",  
            "response_stdout": "rw",  
            "response_stderr": None,  
  
            # Solution and the expected solution result  
            "command": "sudo remountro",  
            "stdout": None,  
            "stderr": None,  
        }  
    }  
}
```

Listado 3.7: Ejemplo del servicio para revisión de RoPI.

En el caso del servicio WeeWX, el encargado de recolectar la información de las estaciones con la ayuda de diversos métodos específicos, sólo es necesario verificar que el proceso que recolecta la información esté funcionando. Este proceso está configurado como un servicio de Linux, por lo cual es posible obtener su estado por medio de la herramienta de sistema `systemd`.

Entre las fallas comunes de este servicio, podemos encontrar una sobrecarga de la estación ocurrida cuando el servicio hace demasiados intentos de recolección al equipo destino sin enviar los terminadores adecuados. Para solucionar este problema, es necesario conectarse con la herramienta destinada a la conexión con los dispositivos y ejecutar una limpieza de memoria, con el comando `wee_device -clear-memory`. Por último, cuando no sea posible ejecutar este comando por una conexión previa no finalizada al sistema, se desconecta forzozamente la conexión serial con las herramientas que Linux proporciona. Este proceso mencionado, se definió en el sistema de monitoreo de la forma que se muestra en el Listado 3.8

En el caso de los dispositivos que se componen de una estación Campbell conectada por Ethernet a una RaspberryPi, es necesario revisar el estado de conexión al dispositivo, así como asegurar que dicho dispositivo ha generado los registros correspondientes y los ha almacenado en su memoria interna. Para esto, se obtiene la información desde una consulta a una página web que la contiene, generada dinámicamente de los registros internos. Ya que estos registros poseen la información de la fecha de última actualización, se puede tomar este valor y compararlo con el de la fecha y hora actuales del servidor de monitoreo para obtener la diferencia.

Los retos de este servicio de monitoreo radican en la necesidad de parámetros extra de información de la estación meteorológica que no se encuentran dentro de la misma por defecto, entre estos parámetros se encuentran la dirección IP local y el puerto en el que el dispositivo Campbell hace accesibles los datos, así como el nombre de la tabla en la que se encuentran los registros. Por esto, fué necesario extraer estos datos en la instancia del driver y utilizarlos

```
service = {

    "command": "systemctl status weewx",
    "stdout": "Active: active (running)",
    "stderr": None, # None implica que esperamos que se encuentre vacío

    "actions": {
        "unable_to_wake": {
            "description": "Problema de conexión a la consola Davis por
                           → puerto serial",
            "solution": "Reinicia la memoria de la consola Davis por medio
                         → de la opción --clear-memory",
            "response_stdout": "vantage: Unable to wake up console",
            "response_stderr": None,
            "command": "sudo wee_device --clear-memory && sudo wee_device
                         → --info",
            "actions": {
                # There is no connection to clear the memory of the station
                "bad_serial": {
                    "description": "No tenemos conexión para limpiar la
                                   → memoria de la estación",
                    "solution": "Reiniciar conexión serial",
                    "response_stdout": "OSError: [Errno 11] Resource
                                      → temporarily unavailable",
                    "response_stderr": None,
                    "command": "sudo systemctl stop
                               → serial-getty@ttyS0.service"
                }
            }
        }
    }
}
```

Listado 3.8: Ejemplo del servicio WeeWX.

```
[...]  
class RpiCampbellStation(BaseDriver):  
    def __init__(self, station, services_map=None):  
        services_map = DEFAULT_SERVICES_MAP  
        extra_data = station.extra_data.serialize()  
        # Get's the IP from the extra_data array of dictionaries  
        for d in extra_data:  
            if d['key'] == 'campbell_ip':  
                campbell_ip = d['value']  
            if d['key'] == 'campbell_port':  
                campbell_port = d['value']  
            if d['key'] == 'campbell_table':  
                campbell_table = d['value']  
  
[...]  
  
    # Create the services  
    services_map['campbell'] = campbell.Service(  
        campbell_ip, campbell_port, campbell_table  
    ).service()  
    return super().__init__(station, services_map)
```

Listado 3.9: Controlador para estación Campbell.

para instanciar el servicio. La instancia del servicio en el driver se muestra en el Listado 3.9 y su aplicación en el servicio en el Listado 3.10.

```
class Service:
    def __init__(self, ip, port, tableName):
        [...]

    def service(self):
        command = 'TARGET_STRING=$('
        # Get's the page from the campbell station
        command += 'curl -s "http://{0}:{1}?command>NewestRecord&table={2}"'
        → |'.format(self.ip, self.port, self.tableName)

        # From the output of curl, get's the record date
        command += 'grep "Record Date" | grep -oP "(\d{4})-(\d{2})-(\d{2})"
        → (\d{2}):(\d{2}):(\d{2})"'
        command += ')'

        # Gets and compares the datetime against the current time in bash
        command += 'TARGET=$(date -d "$TARGET_STRING" +%s); CURRENT=$(date
        → +%s); MINUTES=$(( ($TARGET - $CURRENT) / 60 ));' 

        command += 'if [ $MINUTES -gt 15 ]; then echo "true"; else echo "false";
        → fi'

    return {
        "command": command,
        "stdout": "false",
        "stderr": None,
        "actions": {}
    }
```

Listado 3.10: Ejemplo del servicio para monitoreo de estación Campbell.

3.5.3. Del API para el acceso a la información

El API para el acceso a la información fue desarrollado con la ayuda de FastAPI y MasoniteORM. Estos fueron utilizados para proveer acceso a los estados de las estaciones, datos históricos, registros de datos, así como para la llamada a los controladores de las estaciones para operaciones especiales.

Debido a la arquitectura de FastAPI, el sistema se dividió en `routers`, `requests` y `modelos`. Todos estos son instanciados y llamados desde un archivo `main.py` que se encuentra en la ruta `/app/app/main.py`, conforme a las especificaciones de desarrollo estándar del framework. Además, en este archivo se especifican descripciones estándar generales que serán incluídas al autogenerar la documentación, tal como la de la Figura 3.12, así como definiciones generales de acceso, permisos de CORS y definiciones globales de las rutas.

Para el manejo de la información, se crearon rutas para los módulos de autenticación, incidentes, controladores y estaciones. Estas rutas soportan operaciones CRUD básicas y existen rutas específicas para la interacción con algunos servicios, tal como es el caso del registro de estaciones que permite instanciar una instancia del controlador para registrar una nueva estación en el sistema.

Para el manejo de la información, se utilizó MasoniteORM de forma estándar como lo indica su documentación, aunque cabe notar que debido a limitaciones del framework y a que es utilizado con FastAPI, a pesar de que las definiciones de los modelos contienen las relaciones de la información necesarias estos tienen un problema al intentar ser serializados automáticamente por el framework al momento de ser llamados en una ruta, por lo que se tiene que hacer el llamado a la función `serialize()` de cada uno de los miembros en relación para cada una de las estaciones, esto resulta en código como el que puede ser observado en el Listado 3.11.

Meteoreo API (0.0.1)

Download OpenAPI specification: [Download](#)

Meteoreo API

Meteoreo is a system for monitoring and control oriented to meteorological stations. This API is one of the three main components of the system. This API permits you to modify and create the data required for a meteorological station, and to obtain the global status of them. *Así como* send (directly, or in a queue) a system command to the station, if available.

Meteoreo es un sistema de monitoreo y control orientado a estaciones meteorológicas, el API es uno de los tres componentes principales del sistema. Este permite agregar y modificar los datos de las estaciones que se monitorean, así com obtener el estatus general de las mismas y enviar comandos directamente, así como enviar un comando en método pull.

Authentication

HTTPBasic

Security Scheme Type	HTTP
HTTP Authorization Scheme	basic

Figura 3.12: Visualización de documentación en OpenAPI.

3.5.4. De la interfaz gráfica del proyecto

El desarrollo de la interfaz gráfica del proyecto pasó por una serie de iteraciones para poder llegar a un estado de calidad que se considerara adecuado para el proyecto. Estas iteraciones fueron realizadas de forma eficaz gracias a la arquitectura basada en componentes que ofrece Tailwind y VueJS, permitiendo reescribir sólamente partes del sistema cada vez que fuera requerido algo.

La estructura del proyecto se dividió en un router para la definición de las rutas que serían accesibles en el sistema, una carpeta de vistas que son incluídas desde el router para la

```
from uuid import UUID
from ..models.Station import Station
[...]
@router.get("/{uuid}")
def get_station(uuid: str):
    """
    Get a specific station from it's uuid=
    """
    result = Station.find(uuid)
    result.incidents = result.events.serialize()

    return result.serialize()
```

Listado 3.11: Serialización de datos con FastAPI y MasoniteORM.

generación de las estructuras principales de las páginas web a las que son necesarias acceder, y por último una carpeta de componentes que permitiera el crear archivos individuales para la funcionalidad de los componentes que integran la aplicación.

La vista principal del sistema es una vista general de todas las estaciones tal como estaba definida en el capítulo de diseño (véase Figura 3.13), la cual ofrece un estado comprensivo de cada estación y que al seleccionar una estación, abre una segunda vista con información más específica de la estación, como puede ser observado en la Figura 3.14. Esta vista posee una cualidad reactiva, lo cual permite que los datos sean actualizados en la página sin necesidad de ejecutar una operación de recarga en el navegador en el que se está viendo.

Gracias a la cualidad adaptativa de Tailwind, que utiliza un sistema de puntos de ruptura orientado a la máxima eficacia en el desarrollo para móviles, el sistema resultante fue posible hacerlo compatible con estos sistemas con una cantidad mínima de esfuerzo, la interfaz resultante, que puede verse en la Figura 3.15, tiene una visualización adecuada en sitios que no son de escritorio.

3.5. DESARROLLO

57

Estación	IP	Última actualización	Sensores
Estación Clínica Universitaria	148.210.8.33	hace 20 días	network: OK, mysql: OK, weewx: OK, RoPi: OK, proxy: Err, campbell: OK
Estación Rancho escuela	148.210.8.39	hace 2 días	network: Err, mysql: Err, weewx: ??, RoPi: ??, campbell: ??
Estación Anapra	148.210.8.31	hace 7 minutos	network: OK, mysql: OK, weewx: OK, RoPi: OK, time: OK
Estación IIT	148.210.123.117	hace 7 minutos	network: OK, mysql: OK, weewx: OK, RoPi: OK
Estación Babícora	148.210.8.32	hace 7 minutos	network: OK, mysql: OK, weewx: OK, RoPi: OK
Estación Rancho Reforma	148.210.8.37	hace 2 días	network: Err, mysql: ??, weewx: ??, RoPi: ??

Figura 3.13: Vista general de las estaciones.

Figura 3.14: Vista de una estación con fallas.

Posteriormente se desarrolló la vista de creado de una nueva estación, la cual se comunica con el API para tomar la información de la interfaz gráfica y enviarla, dicha interfaz se ve

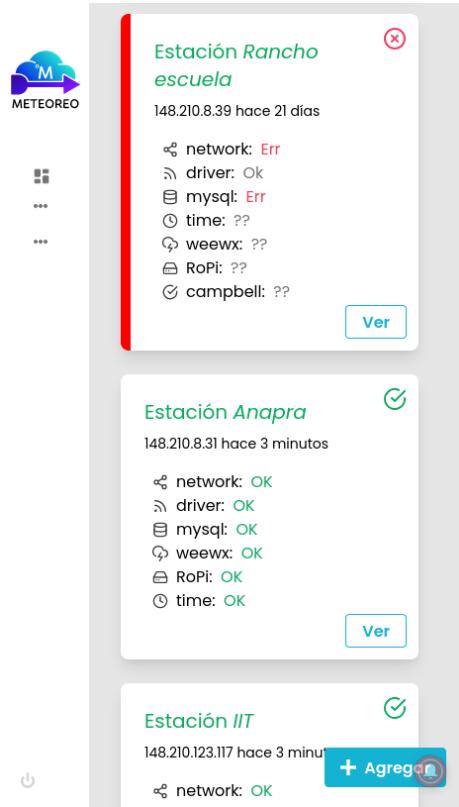


Figura 3.15: Vista general en un celular.

como puede ser observada en la Figura 3.16. Al ser enviada, el API se conecta con el resto del sistema para dar de alta la estación, y de ser necesario el utilizar la información proveída para registrar la llave SSH en el sistema destino. Si este proceso es exitoso, se obtiene una respuesta satisfactoria y la interfaz es redirigida a la vista general de las estaciones, donde se puede observar la estación que se estaba registrando ya en el sistema.

Una funcionalidad que no se encontraba contemplada al realizar el diseño inicial del sistema es la de tener una vista comprensiva de todos los eventos registrados en las estaciones que permitiera ver el historial de eventos completo, por lo tanto, se diseñó y se implementó una vista de bitácora. En esta vista, se agregó la funcionalidad de poder filtrar la información por diversos parámetros para poder realizar una evaluación rápida y precisa del sistema, así como eventualmente utilizar la información para generar reportes de calidad de los datos de

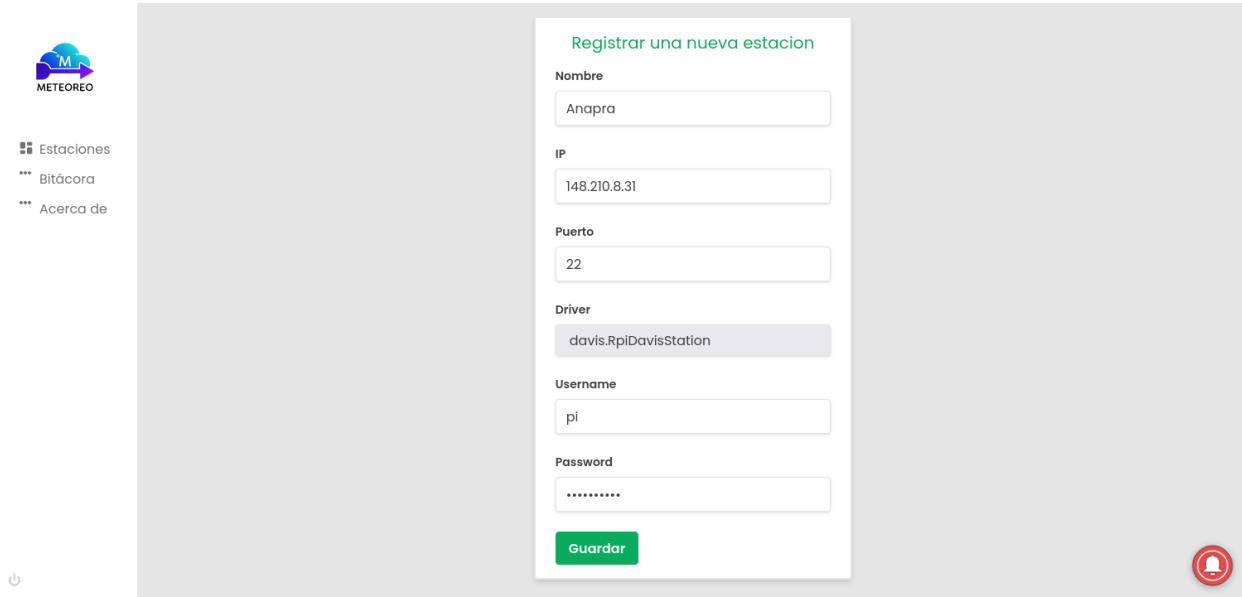


Figura 3.16: Vista de registro de una estación.

los sistemas monitoreados, la pantalla resultante es la que se puede observar en la Figura 3.17.

ESTADO	TIPO DE FALLA	ESTACIÓN	
Pendientes	Servicio	Todas	
Estación Clínica Universitaria		Estado Pendiente	Encontrado 28/04/2022
Problema: El proxy no está corriendo		Solución: Iniciar el servicio de proxy	Actualizado 28/04/2022
Comando a ejecutar: cd /mnt/usb/proxy; sudo python port-forward.py &		Dirección: proxy.actions.start_the_proxy	
Salida: Not running			
Estación Rancho escuela		Estado Pendiente	Encontrado 28/04/2022
Dirección: mysql.actions		Actualizado 28/04/2022	
Salida:			
<ul style="list-style-type: none"> • mysql.service - LSB: Start and stop the mysql database server daemon Loaded: loaded (/etc/init.d/mysql) Active: failed (Result: exit-code) since Wed 1969-12-31 17:00:55 MST; 52 years 3 months ago Process: 1848 ExecStart=/etc/init.d/mysql start (code=exited, status=1/FAILURE) 			
Estación Rancho escuela		Estado Pendiente	Encontrado 28/04/2022
Problema:		Solución:	Actualizado 28/04/2022
Dirección:			

Figura 3.17: Vista de bitácora con filtros aplicados.

El sistema de autenticación que utiliza el proyecto está basado en el estándar *HTTPBasic*, que sólo requiere de un usuario y contraseña para realizar operaciones en el sistema, el manejo de las operaciones de autenticación y de cerrado de sesión fueron posibles con un middleware creado gracias a la librería *Axios*, la cual permite definir un preproceso a la información para tomar acciones correspondientes en el ambiente, como puede ser renovar un token de seguridad cuando se detecte que está próximo a expirar, o como lo es en este caso para agregar las cabeceras adecuadas para realizar la autenticación.

Por último, se integró el SDK de OneSignal, para ser integrado, sólo se requirió agregar las dependencias al sistema por medio de el gestor de paquetes npm, y seguir las instrucciones que aparecen al momento del registro. Debido a que el sistema se configuró para utilizar variables de entorno que fueran accesibles mientras se utiliza el sistema, fué posible especificar el AppID sin dejarlo escrito directamente en el código. Además de esto, se utilizaron opciones especiales para que el diálogo de notificaciones que aparece fuera diferente al original. El diálogo resultante es el que se puede observar en la Figura 3.18 y el código relevante en el Listado 3.12.

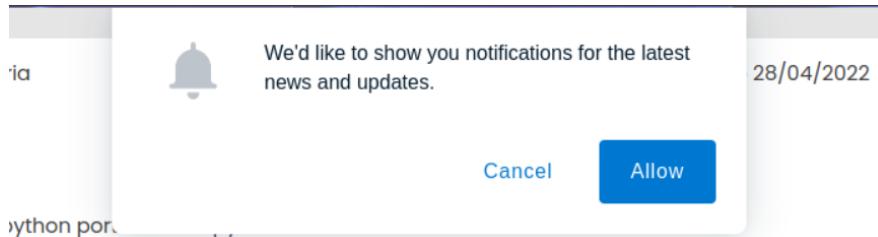


Figura 3.18: Diálogo de notificaciones para OneSignal.

```
[...]  
new Vue({  
  router,  
  render: h => h(App),  
  beforeMount() {  
    this.$OneSignal.init({  
      appId: process.env.VUE_APP_ONESIGNAL_APP_ID,  
      allowLocalhostAsSecureOrigin: true,  
      autoRegister: false,  
      notifyButton: {  
        enable: true,  
        size: 'medium',  
        theme: 'default',  
        position: 'bottom-right',  
      }  
    });  
  },  
}).$mount('#app')
```

Listado 3.12: Configuración de OneSignal en VueJS.

3.6. Pruebas

Una cualidad fundamental del desarrollo ágil son las pruebas continuas del software que es desarrollado, esto fue llevado a cabo, en parte, gracias a la generación automática del esquema OpenAPI con el que se realizaron pruebas continuamente del API con la herramienta Insomnia, tal como se muestra en la Figura 3.19. Con esta herramienta, se corroboró la validez de la información, su estructura y el correcto funcionamiento aún cuando no era posible realizar pruebas del sistema por medio de la interfaz gráfica.

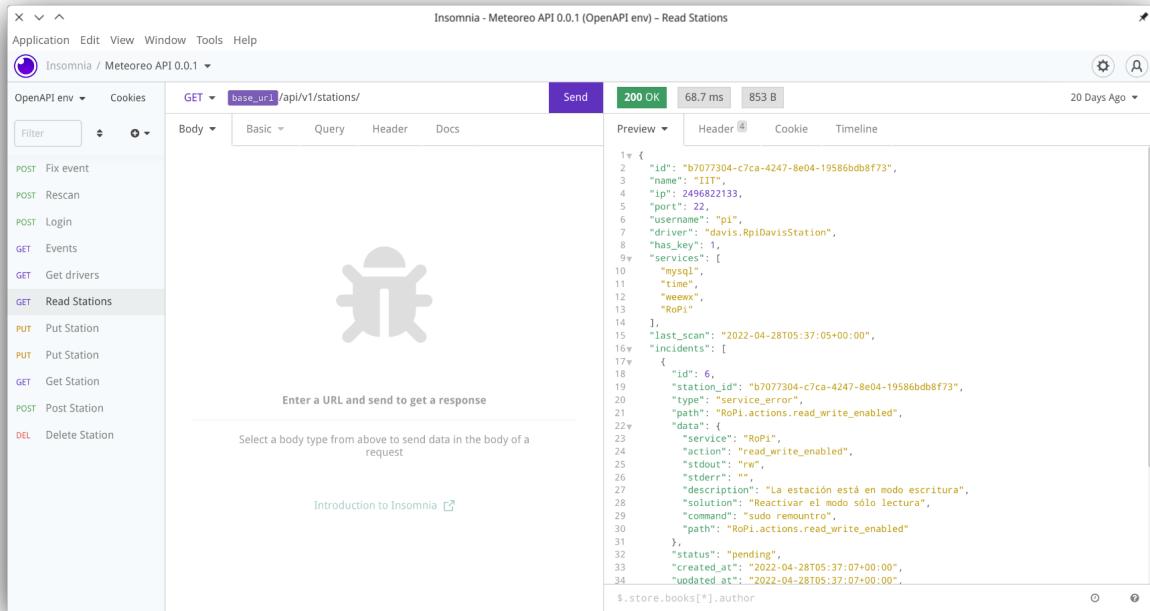


Figura 3.19: Solicitud al servidor de pruebas con Insomnia.

En preparación para un lanzamiento formal del proyecto, se decidió el realizar múltiples pruebas. Una de estas fue realizada con una estación de prueba para no afectar la recolección de datos de las estaciones instaladas. Para ello se utilizó una imagen de Docker existente con el sistema de WeeWX preconfigurado. Esta imagen posee la peculiaridad de tener instalado el servicio como un módulo de `systemd`, esto es poco común debido a que las guías de Docker

```
docker run -td \
--stop-signal=SIGRTMIN+3 \
-v /sys/fs/cgroup:/sys/fs/cgroup:rw --cgroupns=host \
-v /weatherdir:/var/lib/weewx:rw \
--hostname=weewx \
--tmpfs /run:size=100M --tmpfs /run/lock:size=100M \
-e DEBBASE_SSH=enabled \
--network=meteoreo-api_meteoreo-backend \
--name=weewx jgoerzen/weewx:4.6.0
```

Listado 3.13: Funcionamiento de estación de prueba de docker

fomentan la replicabilidad y la modularidad de los contenedores [25], lo cual es incompatible con la configuración seleccionada.

Esta imagen fue utilizada como punto de partida para simular una estación meteorológica, la cual se inició en el ambiente de desarrollo configurado previamente. Para logralo, se creó un comando que ejecutara la imagen de Docker como un proceso en segundo plano, y se adjuntara a la red virtual `meteoreo-api_meteoreo-backend`, la cual fue creada automáticamente al iniciar el sistema localmente. La instrucción resultante, que puede ser encontrada en el Listado 3.13, contiene además de estas especificaciones, configuraciones para el correcto funcionamiento de la imagen y la activación de los servicios SSH.

Después de haber ejecutado el comando anterior, la imagen ya era accesible desde el ambiente de desarrollo creado con VSCode, y después de resolver inconvenientes que surgieron con la estación al no tener configuraciones adecuadas de SSH fue posible acceder de forma convencional por este protocolo a la estación. Después de verificar la conectividad y su correcto funcionamiento, se registró la por medio de la interfaz gráfica, como se muestra en la Figura 3.20.

Al completar el registro de la estación meteorológica, se decidió verificar el correcto fun-

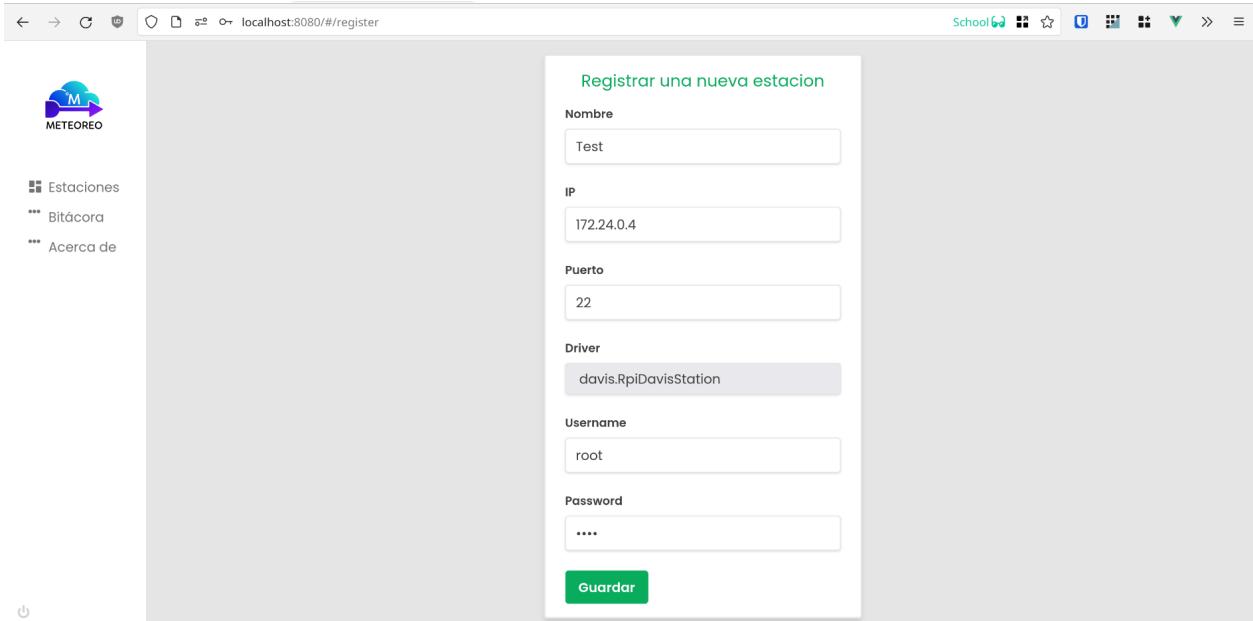


Figura 3.20: Registro de la estación de prueba.

cionamiento del servicio de monitoreo. Al terminar este proceso, era posible observar que la estación tenía 2 fallas, como se muestra en la Figura 3.21. Estas fallas eran el error del servicio RoPI, la cual era esperada debido a que las limitaciones de Docker impiden volver el sistema raíz como solo escritura; y una falla porque el servicio de MySQL no fue encontrado en la imagen.

Esta falla fue solucionada al instalar MySQL en la estación de prueba, lo cual se realizó descargando el paquete correspondiente a la estación del sitio oficial y configurándolo con la utilería de sistema `dpkg`. Al haber completado este proceso, se volvió a revisar el estado de la estación de acuerdo al servicio de monitoreo, y el resultado esperado fue encontrado.

Después de este proceso de configuración se procedió a realizar la prueba del correcto funcionamiento del servicio de monitoreo. Para ello, se detuvieron los servicios de MySQL y WeeWX en la estación de prueba, el resultado de esta acción puede observarse en la Figura 3.22.

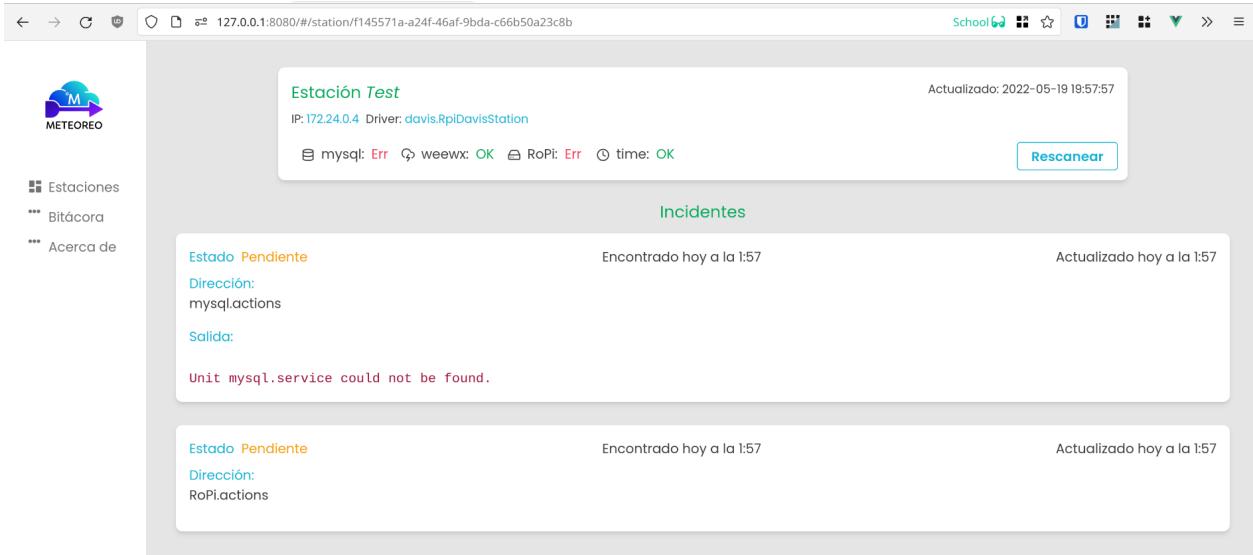


Figura 3.21: Estado inicial de la estación de prueba.

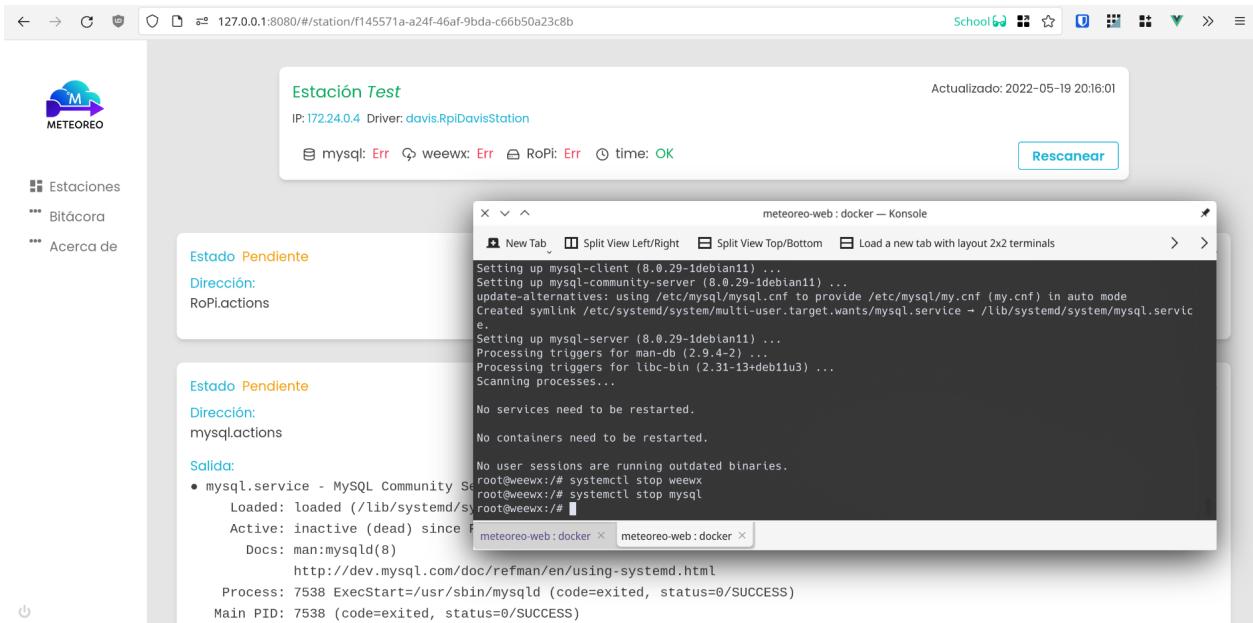


Figura 3.22: Estado de la estación de prueba con servicios desactivados.

Este resultado fué el esperado, ya que no existen configuraciones de fallas comunes para el estado detendo del servicio MySQL o WeeWX dado que estos pueden estar detenidos por una serie de razones impredecibles y no se ofrece el botón de acción de resolver estos errores,

pero se da el detalle de este estado en la interfaz para facilitar el diagnóstico y retención de la información. Cuando los servicios en cuestión fueron vueltos a ejecutar, el sistema resolvió automáticamente los eventos y se mostró correctamente la información, con lo cual se dió por concluída la prueba, como puede observarse en la Figura 3.23.

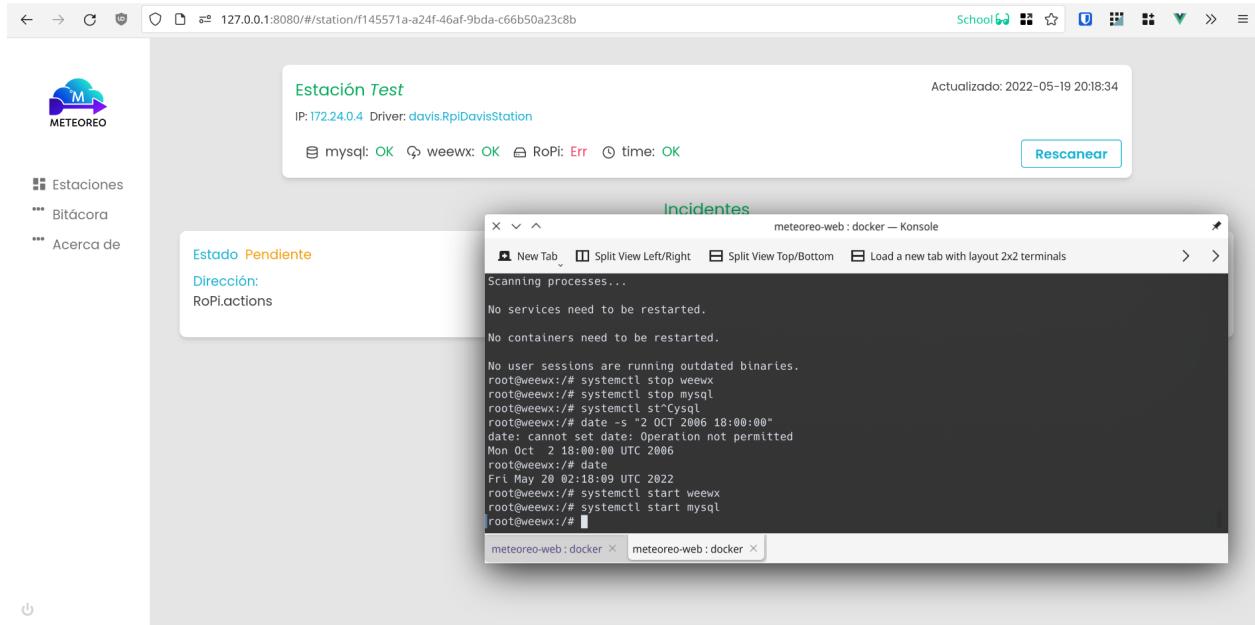


Figura 3.23: Estado final de la estación de prueba.

Además de esta prueba realizada se realizó un proceso similar, aunque menos invasivo, en la estación meteorológica ubicada en el Instituto de Ingeniería y Tecnología de la UACJ. Esta estación se eligió debido a su alta disponibilidad y facilidad de diagnóstico ante un posible problema. En este caso, se hicieron pruebas con el servicio de RoPI, debido a que el impacto de su funcionamiento no ideal es negligible en un periodo corto de tiempo. Para esto, se realizó una conexión por SSH a la estación y se ejecutó el comando `sudo remountrw`, lo cual desactiva el modo sólo lectura. Después de ejecutarlo, se observó la interfaz gráfica para corroborar el correcto registro de este evento fué el previsto, como puede ser observado en la Figura 3.24.

Al corroborarse que los resultados fueran los esperados, se presionó el botón de solución

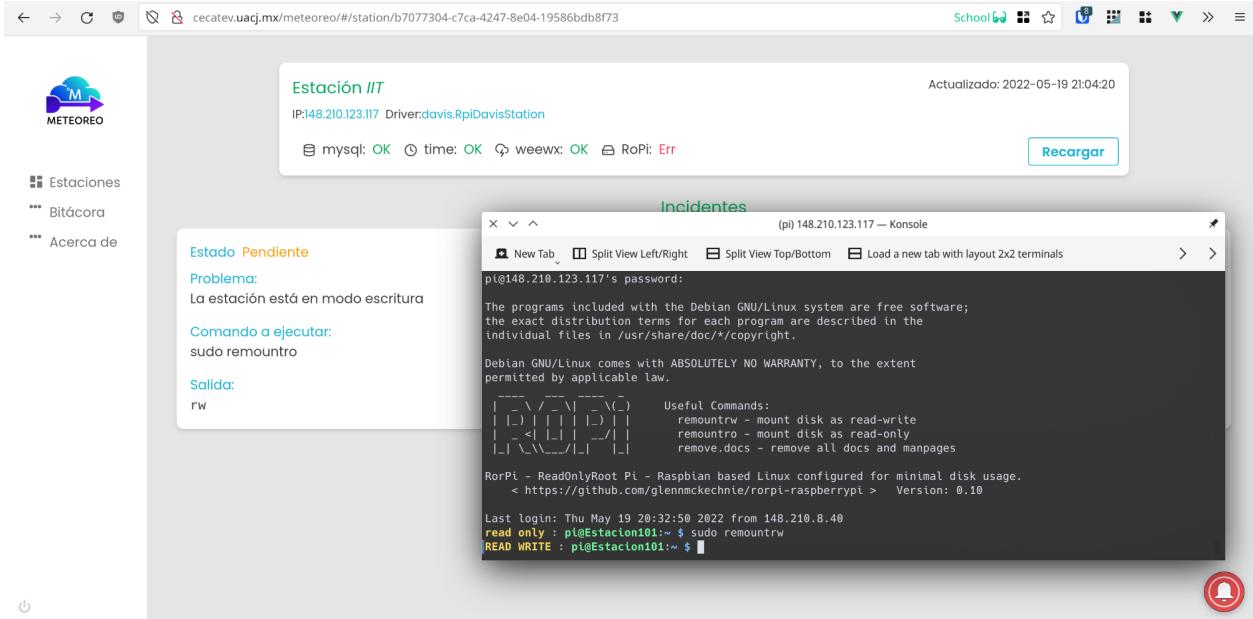


Figura 3.24: Modo sólo lectura desactivado en la estación IIT.

automática del error. Esto ejecutó el comando `sudo remountro`, como indica la tarjeta de error de servicio, en la estación meteorológica, y después de que el servicio de monitoreo detectara el cambio del estado, este fue reflejado correctamente en la interfaz, como se observa en la Figura 3.25. Con este resultado, y el resultado de la prueba anterior se determinó que el sistema era adecuado para su lanzamiento.

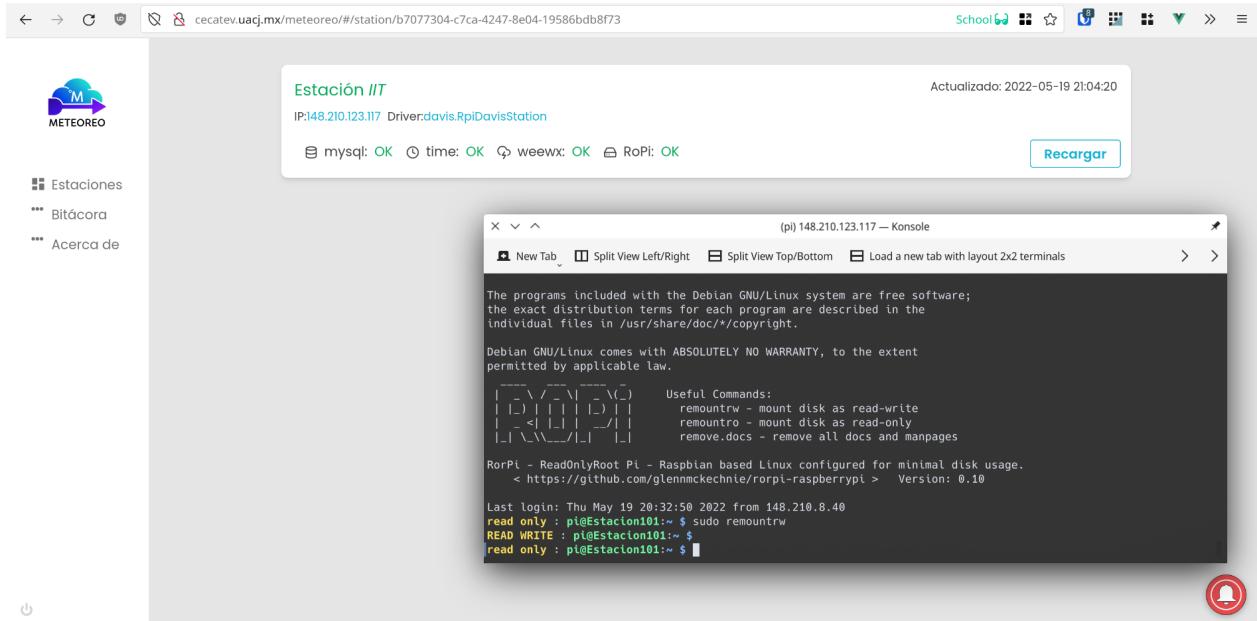


Figura 3.25: Modo sólo lectura reactivado en la estación IIT.

3.7. Despliegue

El despliegue del sistema fué realizado en dos procesos separados: El despliegue de la interfaz gráfica del sistema web, y el de el sistema de monitoreo y API. Esto se realizó debido a que las herramientas necesarias para el funcionamiento del API no eran accesibles desde la plataforma principal, por lo que se decidió dividir el proyecto y tenerlo en servidores independientes.

Para simplificar el proceso del sistema de despliegue de la interfaz gráfica, se creó un usuario en el servidor del sitio CECATEV para ahí depositar los archivos necesarios. Además, para utilizar la infraestructura existente de el Centro, se decidió crear una ruta dentro del sitio en la que fuera posible acceder al proyecto desarrollado. Para esto, se creó la carpeta `meteoreo` en la ruta `/var/www/html/`.

Después de crear la carpeta, se utilizó el gestor de paquetes para iniciar la compilación de un sitio para producción, con el comando `yarn build`, el cual fue configurado automáticamente gracias al framework. Este comando produce una versión del sitio web sin librerías extra y con mejoras de velocidad en comparación a su contraparte de desarrollo.

Por último, para enviar la versión de producción del sitio web al servidor, se utilizó la herramienta `rsync`, la cuál fue seleccionada debido a su simpleza de uso, replicabilidad y compatibilidad con el protocolo SSH que es utilizado para acceder al servidor. Esta operación fué repetida múltiples veces durante el desarrollo del proyecto, y fué posible encapsularlo en una sola línea de código, con la forma `yarn build && rsync -avz dist/ -e ssh al142912@cecatev.uacj.mx:/var/www/html/meteoreo`. Un fragmento de este proceso puede de ser observado en la Figura 3.26.

El proceso de despliegue del API y servicio de monitoreo se realizó en un servidor diferente, una máquina virtual manejada por el instituto a la que se le asignaron 8GB de RAM, 8 núcleos del procesador Intel Xeon E5-4640 con una velocidad de 2.399GHz y 492GB de disco duro. En

```

~/repos/meteoreo-web main !3 yarn build && rsync -avz dist/ -e ssh al142912@cecatev.uacj.mx:/var/www/html/meteoreo
yarn run v1.22.19
$ vue-cli-service build
Browserslist: caniuse-lite is outdated. Please run:
  npx browserslist@latest --update-db
  Why you should do it regularly: https://github.com/browserslist/browserslist#browsers-data-updating

  Building for production...
warn - Tailwind is not purging unused styles because no template paths have been provided.
warn - If you have manually configured PurgeCSS outside of Tailwind or are deliberately not removing unused styles, set 'purge: false'.
warn - https://tailwindcss.com/docs/controlling-file-size/#removing-unused-css
  Building for production.. Browserslist: caniuse-lite is outdated. Please run:
  npx browserslist@latest --update-db
  Why you should do it regularly: https://github.com/browserslist/browserslist#browsers-data-updating
Browserslist: caniuse-lite is outdated. Please run:
  npx browserslist@latest --update-db
  Why you should do it regularly: https://github.com/browserslist/browserslist#browsers-data-updating
Browserslist: caniuse-lite is outdated. Please run:
  npx browserslist@latest --update-db
  Why you should do it regularly: https://github.com/browserslist/browserslist#browsers-data-updating
  Building for production...

```

Figura 3.26: Despliegue de la interfaz web.

este servidor fué instalado el sistema operativo Ubuntu Server versión 18.04.6 LTS, y sobre este sistema operativo se configuró docker conforme a la guía oficial de Docker para la versión correspondiente del sistema. Cabe resaltar que por fines de asegurar una mejor conexión con las estaciones destinadas a ser monitoreadas, este servidor fue asignado al segmento de red perteneciente a estas, ocasionando que no fuera accesible desde la red pública, y para tener acceso se requiere del uso de una VPN.

Después de haberse instalado Docker, se instaló la utilería *docker-compose*, para facilitar la configuración del sistema. Después que se configuraron estas herramientas, se descargó el código del repositorio con la herramienta Git en la carpeta */opt/meteoreo-api*. Después de esto, se configuraron las variables de entorno correspondientes y se ejecutó el comando *docker-compose up* para arrancar el funcionamiento del servicio.

El servicio de monitoreo se decidió ejecutar en la misma imagen de Docker como un proceso manual, esto debido a que se podría facilitar el proceso de monitoreo y actualización. Por lo tanto, se dejó el proceso ejecutando en una ventana de *tmux*, el cual permite crear terminales virtuales para ejecutar procesos sin afectar la terminal principal. Una nota importante es que el sistema se configuró para utilizar una instancia local de la base de datos MariaDB, esto debido a que el sistema que se desplegó será utilizado temporalmente para revisar la estabilidad del mismo.

Como resultado de la configuración seleccionada es posible actualizar el sistema con una versión nueva con facilidad, como se puede observar en la Figura 3.27, debido a que sólo es necesario actualizar el repositorio de Git para que los cambios se vean reflejados. Esto, a pesar de que es un proceso manual, agiliza el despliegue de nuevas versiones que solucionen problemas sin afectar el tiempo de respuesta al usuario final.

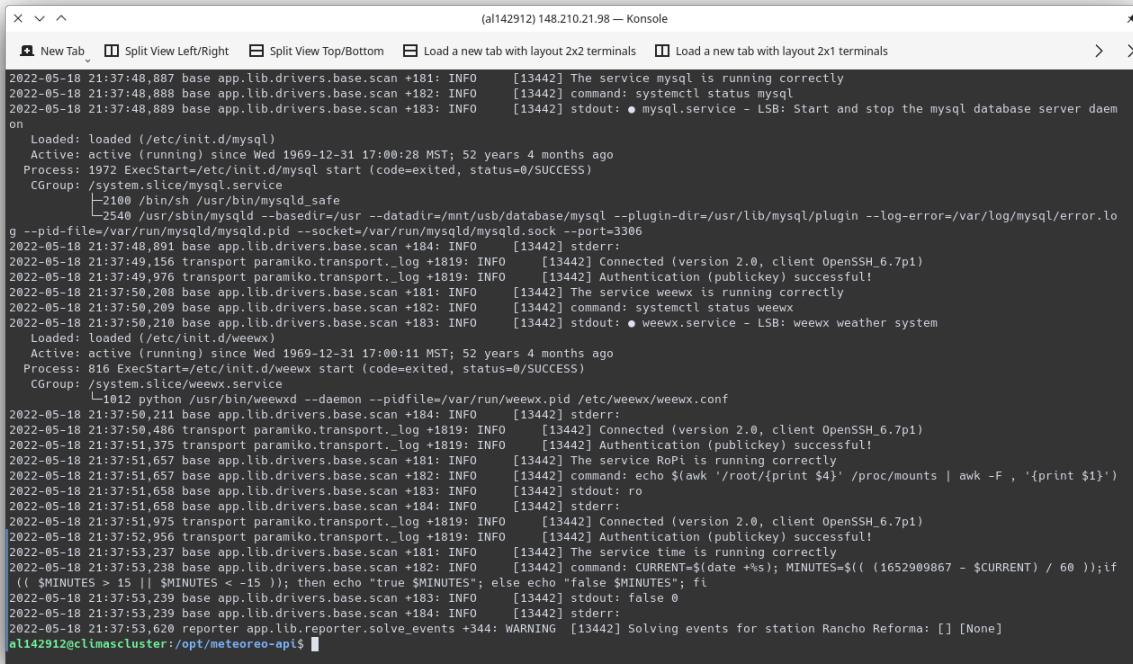
```
al142912@climascluster:/opt/meteoreo-api$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
b852e1d2d86e meteoreoapi_api "/start-reload.sh" 10 days ago Up 10 days 0.0.0
c19df8b81381 mariadb "docker-entrypoint.s..." 10 days ago Up 10 days 0.0.0
al142912@climascluster:/opt/meteoreo-api$ docker-compose stop
Stopping meteoreo-api ... done
Stopping meteoreo-mariadb ... done
al142912@climascluster:/opt/meteoreo-api$ git pull
remote: Enumerating objects: 31, done.
remote: Counting objects: 100% (31/31), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 21 (delta 16), reused 21 (delta 16), pack-reused 0
Unpacking objects: 100% (21/21), done.
From https://github.com/thblckjkr/meteoreo-api
  52cb76f..ac38293 main      -> origin/main
Updating 52cb76f..ac38293
Fast-forward
 .vscode/launch.json | 14 ++++++-----+
 app/lib/notifications.py |  5 +----
 app/lib/reporter.py | 67 ++++++++++++++++++++++++++++++-----
 app/lib/services/time.py |  1 +
 app/lib/services/weewx.py |  1 +
 5 files changed, 64 insertions(+), 24 deletions(-)
al142912@climascluster:/opt/meteoreo-api$ docker-compose start
Starting mariadb ... done
Starting api     ... done
al142912@climascluster:/opt/meteoreo-api$
```

Figura 3.27: Actualización del API y servicio de monitoreo.

3.8. Evaluación

El proyecto fue desplegado en los servidores destinados a su lanzamiento final durante 20 días, mientras se realizaba un monitoreo continuo de las estaciones meteorológicas así como se recolectaba la información. Durante este periodo, se realizó un proceso de mejora continua en la que se realizaron ajustes a la visualización de la información.

Además, para facilitar la evaluación de la calidad de la información del sistema, a este se le configuró el nivel de información en las variables de entorno con el nivel `INFO`, el cuál es el nivel mas informativo. Este nivel de información es útil para la verificación de un correcto funcionamiento del sistema, ya que contiene información detallada de los servicios monitoreados, respuestas de sistemas externos, así como información sobre la autenticación que se realiza, un ejemplo se de esta información se puede ver en la Figura 3.28.



```

x v ^
(al142912) 148.210.21.98 — Konsole
New Tab Split View Left/Right Split View Top/Bottom Load a new tab with layout 2x2 terminals Load a new tab with layout 2x1 terminals
2022-05-18 21:37:48,887 base app.lib.drivers.base.scan +181: INFO [13442] The service mysql is running correctly
2022-05-18 21:37:48,888 base app.lib.drivers.base.scan +182: INFO [13442] command: systemctl status mysql
2022-05-18 21:37:48,889 base app.lib.drivers.base.scan +183: INFO [13442] stdout: • mysql.service - LSB: Start and stop the mysql database server daem
on
    Loaded: loaded (/etc/init.d/mysql)
    Active: active (running) since Wed 1969-12-31 17:00:28 MST; 52 years 4 months ago
      Process: 1972 ExecStart=/etc/init.d/mysql start (code=exited, status=0/SUCCESS)
     CGroup: /system.slice/mysql.service
             └─2100 /bin/sh /usr/bin/mysqld safe
g --pid-file=/var/run/mysqld/mysqld.pid --socket=/var/run/mysqld/mysqld.sock --port=3306
2022-05-18 21:37:48,891 base app.lib.drivers.base.scan +184: INFO [13442] stderr:
2022-05-18 21:37:49,156 transport paramiko.transport._log +1819: INFO [13442] Connected (version 2.0, client OpenSSH_6.7p1)
2022-05-18 21:37:49,976 transport paramiko.transport._log +1819: INFO [13442] Authentication (publickey) successful!
2022-05-18 21:37:50,208 base app.lib.drivers.base.scan +181: INFO [13442] The service weewx is running correctly
2022-05-18 21:37:50,209 base app.lib.drivers.base.scan +182: INFO [13442] command: systemctl status weewx
2022-05-18 21:37:50,210 base app.lib.drivers.base.scan +183: INFO [13442] stdout: • weewx.service - LSB: weewx weather system
    Loaded: loaded (/etc/init.d/weewx)
    Active: active (running) since Wed 1969-12-31 17:00:11 MST; 52 years 4 months ago
      Process: 816 ExecStart=/etc/init.d/weewx start (code=exited, status=0/SUCCESS)
     CGroup: /system.slice/weewx.service
             └─1012 python /usr/bin/weewx --daemon --pidfile=/var/run/weewx.pid /etc/weewx/weewx.conf
2022-05-18 21:37:50,211 base app.lib.drivers.base.scan +184: INFO [13442] stderr:
2022-05-18 21:37:50,486 transport paramiko.transport._log +1819: INFO [13442] Connected (version 2.0, client OpenSSH_6.7p1)
2022-05-18 21:37:51,375 transport paramiko.transport._log +1819: INFO [13442] Authentication (publickey) successful!
2022-05-18 21:37:51,657 base app.lib.drivers.base.scan +181: INFO [13442] The service RoPi is running correctly
2022-05-18 21:37:51,657 base app.lib.drivers.base.scan +182: INFO [13442] command: echo ${awk '/root/{print $4}' /proc/mounts | awk -F , '{print $1}')
2022-05-18 21:37:51,658 base app.lib.drivers.base.scan +183: INFO [13442] stdout: ro
2022-05-18 21:37:51,658 base app.lib.drivers.base.scan +184: INFO [13442] stderr:
2022-05-18 21:37:51,975 transport paramiko.transport._log +1819: INFO [13442] Connected (version 2.0, client OpenSSH_6.7p1)
2022-05-18 21:37:52,956 transport paramiko.transport._log +1819: INFO [13442] Authentication (publickey) successful!
2022-05-18 21:37:53,237 base app.lib.drivers.base.scan +181: INFO [13442] The service time is running correctly
2022-05-18 21:37:53,238 base app.lib.drivers.base.scan +182: INFO [13442] command: CURRENT=$(date +%-s); MINUTES=$(( ( 1652909867 - $CURRENT ) / 60 ));if
(( $MINUTES > 15 || $MINUTES < -15 )); then echo "true $MINUTES"; else echo "false $MINUTES"; fi
2022-05-18 21:37:53,238 base app.lib.drivers.base.scan +183: INFO [13442] stdout: false 0
2022-05-18 21:37:53,239 base app.lib.drivers.base.scan +184: INFO [13442] stderr:
2022-05-18 21:37:53,620 reporter app.lib.reporter.solve_events +344: WARNING [13442] Solving events for station Rancho Reforma: [] [None]
al142912@ctmascluster:/opt/meteorologia$ 

```

Figura 3.28: Información de monitoreo.

3.9. Lanzamiento

Este proyecto es de código libre, y fue publicado en Github bajo la licencia GPL-3.0. El API junto con el servicio de monitoreo puede encontrarse en el repositorio thblckjkr/meteoreo-api y el código de la interfaz web puede encontrarse en thblckjkr/meteoreo-web. Los hashes de commit específicos que se utilizaron para la muestra final bajo la que fué evaluada en este proyecto, es el 540ffcf para el repositorio meteoreo-api y el ca37947 para meteoreo-web. Por último, la interfaz web es accesible desde el URL cecatev.uacj.mx/meteoreo.

Una copia de estos repositorios se encuentra en la instancia de Gitlab propia de la Universidad Autónoma de Ciudad Juárez, para permitir un mayor control del código alojado en los servidores así como una mayor facilidad para la futura modificación del repositorio por los interesados. La copia de estos repositorios se encuentra en la ruta meteoreo/meteoreo-api y en meteoreo/meteoreo-web.

Capítulo 4

Resultados y Discusiones

Durante la fase de evaluación del sistema, se estuvo recolectando información de las diversas estaciones meteorológicas instaladas en la región. Esta información permitió evaluar la viabilidad del proyecto a corto y mediano plazo, además, también facilitó realizar un análisis preliminar de los datos recolectados. En la primer sección de este capítulo, se analizarán los datos recolectados por el sistema, posteriormente, se hará un análisis de la capacidad de extensibilidad del sistema para cubrir nuevos casos de uso y se terminará en pruebas de estrés y carga realizadas al mismo.

4.1. Sobre los datos obtenidos

Gracias a la información recabada durante el periodo de evaluación del sistema, fue posible realizar un análisis preliminar de los datos recolectados de las estaciones. Dentro de estos datos se encuentra los tipos de incidencias, su duración y la fecha en la que éstas ocurrieron, así como, datos diversos específicos por cada tipo de incidencia.

Al revisar la información recolectada, se encontró que el mayor problema que se presenta en las estaciones es la inaccesibilidad por red a las mismas, como puede observarse en la figura 4.1. Esta falta de comunicación con las estaciones puede deberse a una serie de factores, como una conexión deficiente de red, problemas temporales del servicio de VPN con el que cuentan

las estaciones, o la falta de energía en el dispositivo.

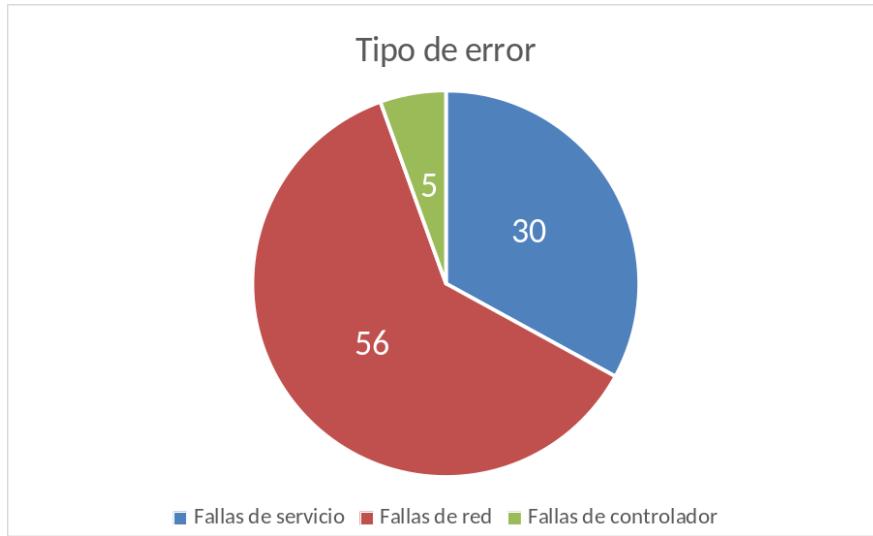


Figura 4.1: Tipos de incidencia.

Como se muestra en la Figura 4.2, el número de incidentes es dependiente de cada estación, esto implica que no existe una coorelación entre ellas. Además, al compararlo con los datos obtenidos de la Figura 4.3, es posible observar que el número de fallas de red es directamente proporcional al tiempo de indisponibilidad de cada estación.

También, es posible observar que la falla que permaneció activa por más tiempo es un error en la base de datos MySQL que impedía el correcto almacenamiento de la información recabada por la estación. La pérdida de estos datos se puede verificar en la página de monitoreo de los datos generados de la estación en la Figura 4.4 en la forma de ausencia de información en gráfica de la vista semanal de los sensores.

El error de recolección de la información fue resuelto y la fecha de solución, como se muestra en la Figura 4.5, coincide con la pérdida de datos de la estación. Esto permite realizar una relación entre una falla concreta de una estación con la pérdida de información. Dicha información, puede ayudar al personal de CECATEV a realizar un proceso de mejora continua para la evaluación y categorización de fallas de las estaciones, con el objetivo de

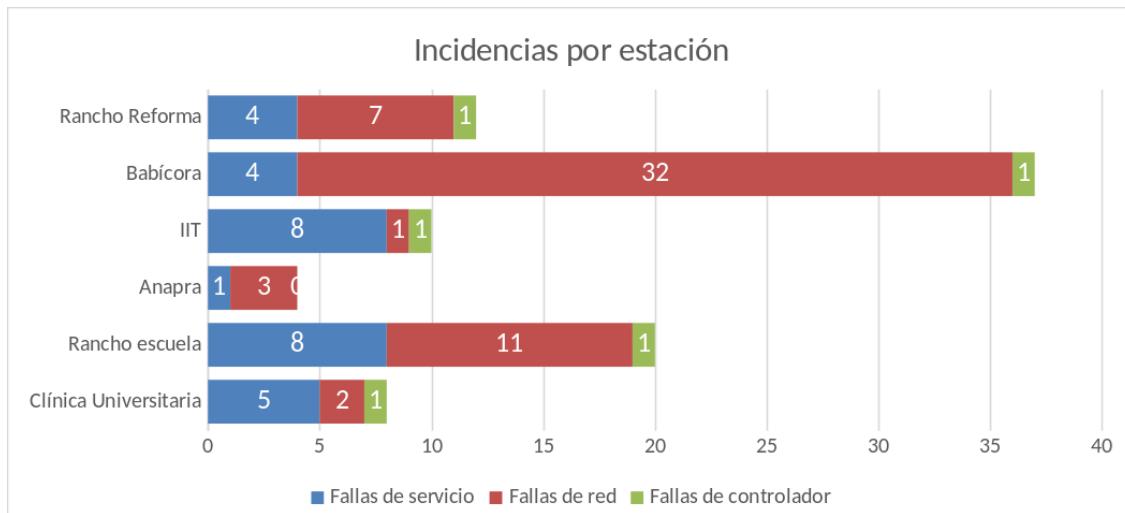


Figura 4.2: Incidencias y sus tipos, agrupadas por estación.

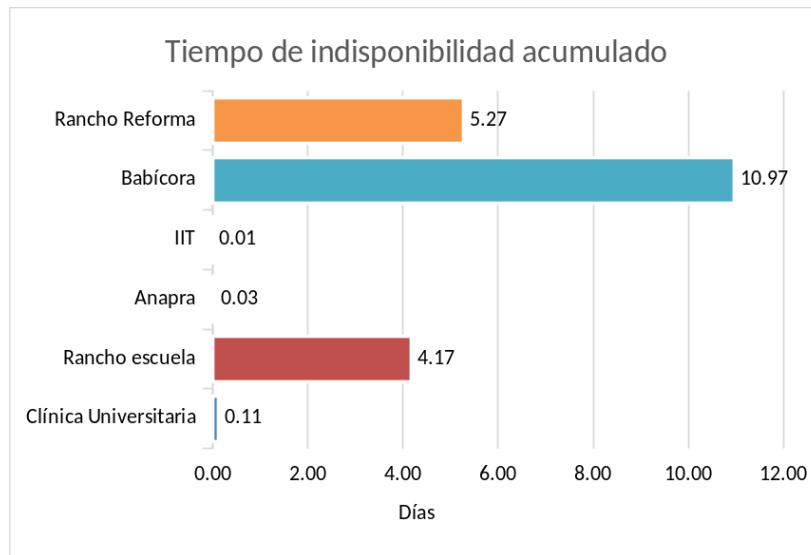


Figura 4.3: Tiempo de inaccesibilidad acumulado por estación.

priorizar los fallos que afectan la calidad de los datos como tarea primordial.



Figura 4.4: Rancho escuela.



Figura 4.5: Datos de falla de Rancho Escuela.

4.2. Sobre la extensibilidad del proyecto

La modularidad del sistema y la forma en la que fue construído permite extenderlo, para que crear nuevos servicios sea una tarea trivial. Ésto, es gracias a que al ser una interfaz

para ejecución de comandos, permite que las complejidades de monitoreo sean resueltas una sola vez, en la terminal de ejecución y siendo limitado sólo por la dificultad de la tarea que se busca llevar a cabo. Esto se puede exemplificar creando un nuevo servicio de monitoreo y analizando la complejidad y el tiempo que llevan implementarlo.

Para este ejercicio, se seleccionó la tarea de crear un nuevo servicio para el monitoreo de uso de disco en las estaciones meteorológicas. Esto es importante, ya que aunque poseen un sistema de sólo lectura para asegurar la longevidad de las tarjetas SD, también cuentan con un sistema de respaldos local en memorias USB. El llevar este servicio de su concepción a un monitoreo activo, tomó aproximadamente 11 minutos, la adición de 16 líneas de código y la modificación de 2 como es posible ver en el Listado 4.1. Si bien, fue una tarea sencilla haciendo uso de las amplias herramientas que poseen los sistemas en los que depende, también es importante notar que el sistema de monitoreo no agregó una fricción considerable para implementarlo.

4.3. Sobre las capacidades de carga del proyecto

Con la ayuda de la herramienta Locust, se realizó una prueba de estrés y carga al sistema. Esta prueba se configuró para realizar pruebas en el sistema final en el que se realizó despliegue, la prueba se compone de dos rutas a probar (véase Listado 4.2), a las cuales se les realizarán peticiones con una cantidad creciente de usuarios concurrentes. Después de haber instalado la librería, se realizaron las pruebas con el comando `locust -H http://148.210.21.98:81 -u 50 -r 0.1 -t 300s -autostart`.

Al analizar la gráfica resultante, la Figura 4.6, de las pruebas, es posible observar que el tiempo de respuesta incrementa de forma lineal conforme a la cantidad de usuarios, comenzando con menos de 1 segundo por respuesta e incrementando hasta alcanzar casi 50 segundos por respuesta al tener a 50 usuarios concurrentes.

```
diff --git a/app/lib/drivers/campbell.py b/app/lib/drivers/campbell.py
-index 3a2a4d5..a2a2a2c 100644
+from ..services import mysql, ropi, time, weewx, proxy, campbell
+from ..services import mysql, ropi, time, weewx, proxy, campbell, disk
[...]
+    "disk": disk.service,
[...]
diff --git a/app/lib/drivers/davis.py b/app/lib/drivers/davis.py
-index 3a2a4d5..a2a2a2c 100644
+from ..services import mysql, ropi, time, weewx
+from ..services import mysql, ropi, time, weewx, disk
[...]
+    "disk": disk.service,
[...]
diff --git a/app/lib/services/disk.py b/app/lib/services/disk.py
+service = {
+    "command": "PERCENT=$(df /mnt/usb --output=pcent | tail -n 1 | tr -d
+    '%')\n"
+    "if (( $PERCENT > 80 )); then echo \"true $PERCENT\"; else echo
+    \"false $PERCENT\"; fi",
+    "stdout": "false",
+    "stderr": None,
+    "actions": {
+        "disk_almost_full": {
+            "description": "El disco está casi lleno",
+            "solution": "Elimine algunos archivos de la estación",
+            "response_stdout": "true",
+            "response_stderr": None,
+        }
+    }
+}
```

Listado 4.1: Diferencia de código al agregar el monitoreo de disco.

```

from locust import HttpUser, task

class API(HttpUser):
    @task
    def get_drivers(self):
        self.client.get("/api/v1/drivers/")

    @task
    def get_stations(self):
        self.client.get("/api/v1/stations/", auth=("admin", "pass"))

```

Listado 4.2: locustfile.py configuración de pruebas.

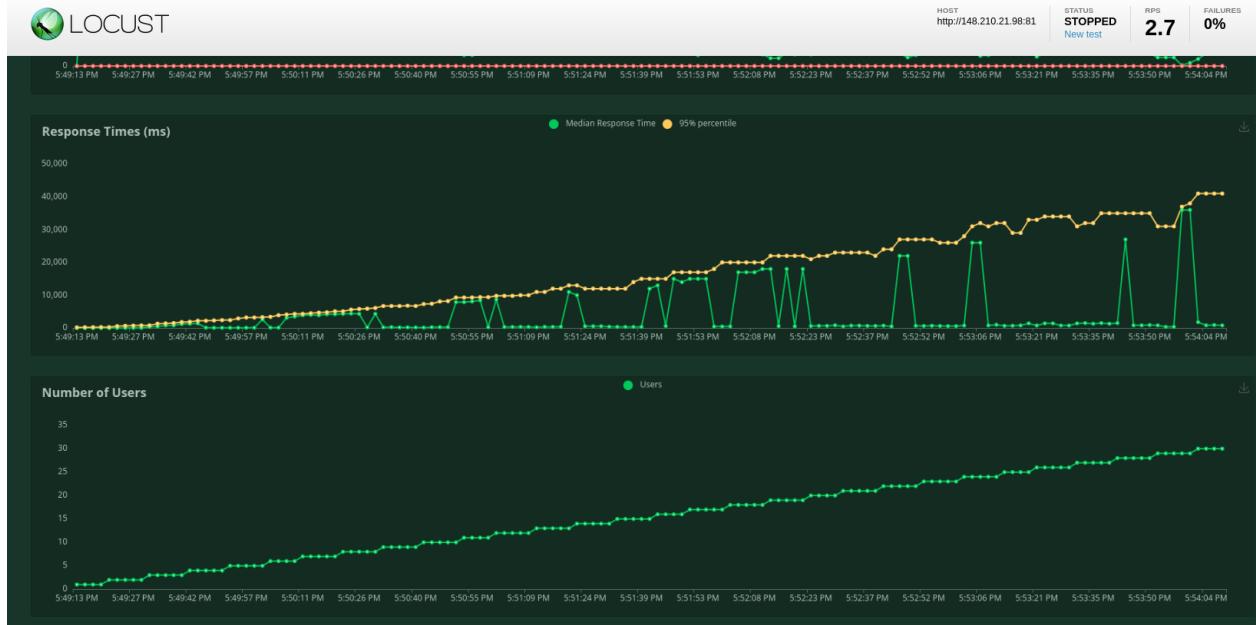


Figura 4.6: Gráfica de usuarios y tiempos de respuesta creada con Locust.

Sin embargo, al revisar el análisis de tiempo de respuesta por ruta presente en la Figura 4.7, es posible observar una diferencia significativa entre el tiempo de respuesta de la ruta `/api/v1/drivers`, con un máximo de 1.825s y la ruta `/api/v1/stations` con un máximo de 41.937s. Esto puede deberse a que la primer ruta mencionada no realiza tareas de lectura

en la base de datos. Esto implicaría que la solución implementada para el almacenamiento de datos, una base de datos MariaDB en Docker en la misma máquina virtual que donde está alojado el API, no es una solución óptima para el problema.

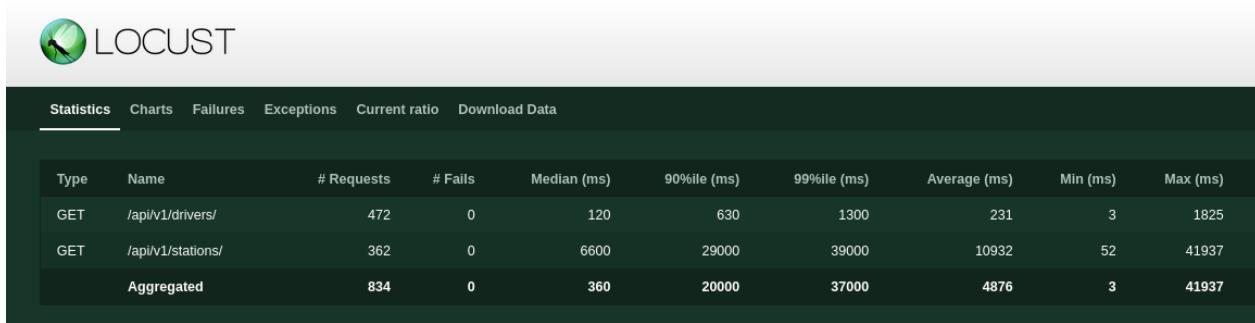


Figura 4.7: Estadísticas de pruebas por ruta.

Capítulo 5

Conclusiones

En este documento se reporta el desarrollo de un sistema de monitoreo y control para estaciones meteorológicas. A través de este apartado se presentan las conclusiones a las que se llegó durante el desarrollo del proyecto, así como las recomendaciones para trabajos futuros.

5.1. Con respecto al objetivo de la investigación

Con el sistema presentado en este documento, se obtiene que es posible realizar una recolección de datos de estaciones meteorológicas y generar un sistema para la solución de problemas de forma remota.

Con respecto al objetivo general del proyecto, que es el desarrollo de un sistema que permita un monitoreo continuo de estaciones meteorológicas, para proveer un mejor mantenimiento preventivo y correctivo, se logró de manera exitosa y funcional en resolver el problema descrito en la sección 1.2, conforme a los resultados presentados en el capítulo 4.

Con la realización del proyecto se lograron los siguientes resultados:

- Se desarrolló un sistema para recolectar los datos del estado de las estaciones, que permitió un mejor control de la información y estado de las mismas.
- Se diseñó un API REST para facilitar la consulta de los datos del estado de las esta-

ciones, documentada con OpenAPI de forma automática y,

- Se generó una interfaz gráfica para la consulta del estado de las estaciones, que permitió una mejor visualización de los datos, integrado con un sistema de notificación de alertas.

5.2. Recomendaciones para futuras investigaciones

El sistema de monitoreo de estaciones Meteoreo fué concebido como un punto de partida para más avances en el mantenimiento y monitoreo de estaciones meteorológicas. La capacidad de extensión y documentación del código tienen la arquitectura necesaria para permitir modificaciones sin alterar el funcionamiento fundamental del programa.

Actualmente, el sistema recolecta información de las estaciones meteorológicas y se permite realizar acciones en base a la información presentada, sin embargo, requiere de interacción para acciones que podrían ser automáticas. Esto abre un área de oportunidad para utilizar la información que el sistema recaba para automatizar la ejecución de las soluciones conforme a los servicios definidos, e incluso, el resolver de forma automática los errores no reconocidos en las estaciones, ya sea con una inteligencia artificial o con un sistema de pesos para una serie predefinida de soluciones comunes.

También se recomienda el realizar un sistema de notificaciones más versátil que pueda alertar a los diferentes involucrados en la administración de las estaciones, ya que actualmente se envía una alerta de error a todos los involucrados en el sistema sin discriminar por parámetros como tipo de falla, o localización de la estación.

Bibliografía

- [1] C. L. Muller, L. Chapman, C. S. B. Grimmond, D. T. Young, and X. Cai, “Sensors and the city: a review of urban meteorological networks,” *International Journal of Climatology*, vol. 33, no. 7, pp. 1585–1600, 2013.
- [2] R. Hayler (2018), “Build your own weather station with our new guide!” [Internet]. Disponible en <https://www.raspberrypi.org/blog/build-your-own-weather-station/>.
- [3] O. Aspiazu Ituarte, *Diseño de un sistema de fertirrigación automático con control telemático y sensores*. PhD thesis, Universitat Oberta de Catalunya, Jun 2019.
- [4] Monitoring Plugins Development Team (2008) , “The monitoring plugins project.” [Internet]. Disponible en <https://www.monitoring-plugins.org/index.html>.
- [5] Nagios (2018), “Guide to configure the restarting of a service in nagios.” [Internet]. Disponible en <https://assets.nagios.com/downloads/nagiosxi/docs/Restarting-Linux-Services-With-NRPE.pdf>.
- [6] F. A. Vázquez-Galvez, F. Estrada-Saldaña, and J. I. Hernández-Hernández, “Red climática y de calidad del aire uacj,” tech. rep., Universidad Autónoma de Ciudad Juárez, 2019.
- [7] M. Massé, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O’reilly, 1 ed., 2012.

- [8] H. Ed-douibi, J. L. Cánovas Izquierdo, and J. Cabot, “Example-driven web api specification discovery,” in *Modelling Foundations and Applications* (A. Anjorin and H. Espinoza, eds.), (Cham), pp. 267–284, Springer International Publishing, 2017.
- [9] OpenAPI Organization, “OpenAPI FAQ.” [Internet]. Disponible en <https://openapis.org/faq>.
- [10] W. Cheng, “Openapi generator · generate clients, servers, and documentation from openapi 2.0/3.x documents,” Apr 2021.
- [11] B. B. Rad, H. J. Bhatti, and M. Ahmadi, “An introduction to docker and analysis of its performance,” *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.
- [12] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and linux containers,” in *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pp. 171–172, IEEE, 2015.
- [13] Codecademy, “Python courses and tutorials.”
- [14] Mozilla, “Javascript documentation.”
- [15] Masonite (2021), “Introduction.” [Internet]. Disponible en <https://orm.masoniteproject.com/>.
- [16] S. Ramírez (2020), “Fastapi.” [Intenret]. Disponible en <https://fastapi.tiangolo.com/>.
- [17] C. Macrae, *Vue.js Up & Running*. O'Reilly Media, Incorporated, 2018.
- [18] MariaDB foundation (2019), “MariaDB Foundation.” [Internet]. Disponible en <https://mariadb.org/>.

- [19] S. Tongkaw and A. Tongkaw, “A comparison of database performance of mariadb and mysql with oltp workload,” in *2016 IEEE Conference on Open Systems (ICOS)*, pp. 117–119, 2016.
- [20] M. Fowler, J. Highsmith, *et al.*, “The agile manifesto,” *Software Development*, vol. 9, no. 8, pp. 28–35, 2001.
- [21] Z. Hussain, M. Lechner, H. Milchrahm, S. Shahzad, W. Slany, M. Umgeher, and P. Wolkerstorfer, “Agile user-centered design applied to a mobile multimedia streaming application,” *Lecture Notes in Computer Science*, p. 313–330, 2008.
- [22] A. Jiménez, J. Nieve, F. Estrada, F. A. Vázquez-Gálvez, and I. Hernández, “Management of heterogeneous data in the red climatológica uacj in a nosql environment,” in *2019 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC)*, pp. 1–6, IEEE, 2019.
- [23] H. Lee, “Justifying database normalization: a cost/benefit model,” *Information processing & management*, vol. 31, no. 1, pp. 59–67, 1995.
- [24] H. Terry, *Object-Role Modeling (ORM/NIAM)*, pp. 81–103. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.
- [25] Docker (2022, Mayo 10), “Run multiple services in a container.” https://docs.docker.com/config/containers/multi-service_container/.