Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

# Detección de inconsistencias en una base de datos siguiendo unas reglas de negocio

Autor: César David Zapata Guano
Tutora: Isabel Román Martínez

**Dpto. Ingeniería Telemática**
**Escuela Técnica Superior de Ingeniería**
**Universidad de Sevilla**

Sevilla, 2024

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de la Telecomunicación

# Detección de inconsistencias en una base de datos siguiendo unas reglas de negocio

Autor:
César David Zapata Guano

Tutora:
Isabel Román Martínez
Profesora colaboradora

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2024

Trabajo Fin de Grado: Detección de inconsistencias en una base de datos siguiendo unas reglas de negocio

Autor: César David Zapata Guano

Tutora: Isabel Román Martínez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2024

El Secretario del Tribunal

*A mi familia*

*A mis amigos*

*A mis compañeros*

*A mis maestros*

# Resumen

El crecimiento del IoT (Internet of Things) está transformando la manera en la que vivimos y trabajamos. Ha habido un incremento exponencial en la cantidad de dispositivos conectados a internet que interactúan entre sí y con sistemas centrales. Esto ha transformado industrias, hogares y ciudades debido al avance de nuevas tecnologías y cambios en las necesidades de los usuarios. Esto ha generado una oportunidad de negocio que se ha visto reflejado en la empresa Giesecke & Devrient con un aumento significativo de la demanda de activos que permiten la conectividad IoT.

IotSuite es la plataforma ofrecida por esta empresa para los clientes, es una aplicación para la gestión y administración de la conectividad de SIMs y eSIMs entre otros productos. Es fundamental que esta aplicación sea robusta y fiable por lo que continuas mejoras son implementadas a diario. Este proyecto va a enfrentar la problemática de la inconsistencia de los datos, para ser concreto, en una base de datos no relacional. Distintas causas provocan errores en el manejo y persistencia de los datos, desde el punto de vista del equipo técnico es necesario identificar y monitorizar tales inconsistencias.

Para asegurar la integridad de la aplicación y sus registros, este trabajo aborda el diseño e implementación de un microservicio que sea capaz de detectar las posibles inconsistencias en la base de datos, detectando entradas incompatibles con unas reglas de negocio que deben de seguir las estructuras de los datos. Estas serán reportadas al equipo técnico para su revisión y resolución.

No solo se abordará el diseño e implementación de la solución, se explicarán y aplicarán técnicas de despliegue para automatizar la configuración y puesta en marcha del nuevo servicio en la infraestructura actual.

Mencionar que ciertas características se excluyen para no violar la política de privacidad de la empresa y que se intenta dar contexto y explicar procesos de lógica de negocio sin entrar mucho en detalle.

# Abstract

The growth of IoT (Internet of Things) is transforming the way we live and work. There has been an exponential increase in the number of internet-connected devices that interact with each other and with central systems. This has transformed industries, homes and cities due to the advance of new technologies and changing user needs. This has generated a business opportunity that has been reflected in the company Giesecke & Devrient with a significant increase in demand for assets enabling IoT connectivity.

IotSuite is the platform offered by this company to customers, it is an application for managing and managing connectivity of SIMs and eSIMs among other products. It is essential that this application is robust and reliable so continuous improvements are implemented daily. This project will address the problem of data inconsistency, to be concrete, in a non-relational database. Different causes cause errors in the handling and persistence of data, from the point of view of the technical team it is necessary to identify and monitor such inconsistencies.

To ensure the integrity of the application and its records, this work addresses the design and implementation of a microservice that is able to detect possible inconsistencies in the database, detecting entries incompatible with business rules that the data structures must follow. These will be reported to the technical team for review and resolution.

Not only will the design and implementation of the solution be addressed, deployment techniques will be explained and applied to automate the configuration and commissioning of the new service in the current infrastructure.

Mention that certain features are excluded because they violate the company's privacy policy and that it tries to give context and explain business logic processes without going into much detail.

# Índice

# ÍNDICE DE FIGURAS

# 1 INTRODUCCIÓN

*En el mundo de las API web, las bases de datos no relacionales proporcionan la flexibilidad necesaria para manejar estructuras de datos dinámicas y cargas de trabajo impredecibles.- Todd Hoff-*

Las bases de datos son esenciales en cualquier aplicación web. La base de datos que explicaremos en esta memoria maneja un volumen alto de datos, estructurados pero constantemente fluctuantes. Para cualquier empresa, mantener la fiabilidad de los datos resulta algo imprescindible para la confianza y satisfacción de los clientes en bien del negocio.

## 1.1 Contexto

Podgroup, una compañía recientemente adquirida por G+D (Giesecke+Devrient), se dedica a facilitar la conectividad IoT alrededor del mundo entero. Es un vendedor de tecnología SIM (Subscriber Identity Module) y eUICC (Embedded Universal Integrated Circuit Card) SIM, en adelante, eSIM, que proporciona un portal para el manejo de sus activos e integra proveedores de conectividad de más de 100 países para asegurar el acceso a la red de cualquier solución IoT. Como explica su página web [1]:

> Giesecke+Devrient (G+D) es una empresa global SecurityTech con sede en Múnich. Como socio de confianza para clientes con las más altas exigencias, G+D hace que la vida de miles de millones de personas sea más segura con sus soluciones.
>
> La conectividad celular altamente segura ha sentado las bases para los miles de millones de dispositivos, vehículos, máquinas e incluso fábricas enteras que están en línea en todo el mundo hoy en día. Y más de 5.000 nuevos dispositivos están conectados a Internet - cada minuto. El Internet de las cosas (IoT) ya es la máquina más grande jamás construida por la humanidad. Liberamos el potencial de negocio para nuestros clientes a través de la innovación IoT con tecnología de seguridad integrada. (Giesecke+Devrient DmbH, 2024)

Entre los productos que se proporcionan destacan las tarjetas Iot SIM y las eSIM. Los clientes que contratan los servicios de conectividad IoT, manejan sus activos (propios o de terceros) desde el portal Iot Suite (https://iotsuite.gi-de.com/). Esta es una aplicación web de gestión, donde se pueden controlar las subscripciones a la red, el uso de datos, de SMSs, diagnósticos de la conexión, facturación, creación y provisionamiento de clientes propios entre otras muchas funciones.


Figura 1. Web de acceso a Iot Suite

Todas las características disponibles desde la UI (User Interface) también son accesibles desde una API (application programming interface) REST (representational state transfer, un estilo de arquitectura de software) que se pone a disponibilidad de los clientes para la integración con sus propias aplicaciones o automatización de tareas.

El stack tecnológico se basa en Javascript, se usa Angular [18] para el desarrollo FE (Front End) y la implementación de la UI. Para el BE (Back End) y la implementación de la API y servidor se usa NodeJs [4] y el Framework Restify [19]. La persistencia de datos usa una base de datos NoSQL llamada MongoDB [2].

Usa una arquitectura de microservicios para poder implementar todas las funciones necesarias para satisfacer la necesidad de adaptación de nuevas tecnologías o necesidades de los clientes. Muchos proveedores de servicios usan tecnologías distintas, la arquitectura de microservicios facilita la integración con la aplicación principal. Están desplegados principalmente en la nube de AWS (Amazon Web Services) y algunos microservicios en la nube de Azure.

## 1.2 Motivación

El mercado de los dispositivos IoT no ha parado de crecer, un estudio en Mayo 2022 por IoT Analytics [20] ya aventuraba un gran salto, como muestra la figura 2.

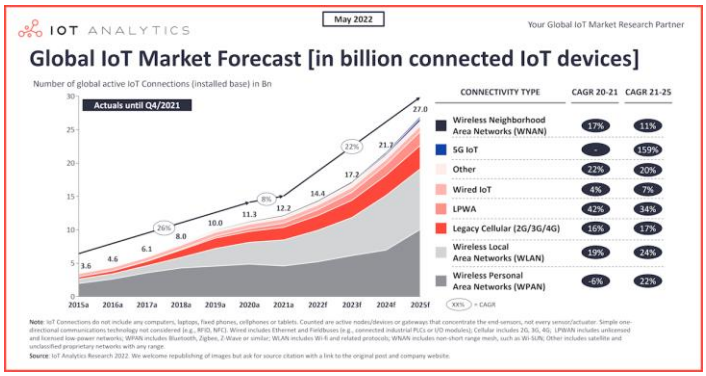



Figura 2. Evolución de dispositivos IoT conectados

Esto se reflejó en un aumento de los clientes de nuestra empresa y se tuvieron que tomar medidas para garantizar

la escalabilidad de la base de datos. Obtener la confianza de los clientes es clave y para ello es necesario garantizar la integridad de los datos. Cualquier problema detectado por un cliente conlleva un sistema de escalación buscando soporte, incrementando mini tareas no automatizadas y aumentando la carga en el equipo de desarrollo. Además de la correspondiente insatisfacción de los usuarios, se pone en duda la fiabilidad de la aplicación y por tanto del equipo de mantenimiento y desarrollo.

Este trabajo aborda el diseño e implementación de un microservicio que sea capaz de detectar las posibles inconsistencias en la base de datos, detectando entradas incompatibles con distintas reglas de negocio.

Algunas de las causas de inconsistencias pueden ser:

- **Condiciones de carrera**: un conjunto de operaciones de escritura concurrentes en las que el manejo no es adecuado, puede llevar a que dos o más operaciones intenten modificar la misma colección de datos.
- **Validaciones insuficientes**: una validación no correcta en la entrada de datos erróneos o contradictorios.
- **Errores de código**: cualquier bug en la aplicación que maneje la lectura y escritura de los datos puede causar que se introduzcan datos inconsistentes.
- **Migraciones o actualizaciones mal planificadas:** cambios en los esquemas de los datos o actualización de colecciones de documentos sin un plan adecuado puede provocar un estado inconsistente de los datos.

## 1.3 Objetivos

Desde el punto de vista técnico del mantenimiento de la aplicación web, la iniciativa de este proyecto parte del equipo de desarrollo. La intención es adelantarse al reporte de incidentes por parte de clientes o equipos de soporte, manteniendo la robustez de la base de datos, siguiendo unas reglas de negocio conocidas o informadas por el equipo de producto en el desarrollo de nuevas funciones.

Para ello la intención es implementar un microservicio que además de automatizar la detección de los desajustes en la base de datos, los notifique, en modo de informe que debe identificar la regla de negocio incumplida y el nivel de impacto que puede provocar el problema detectado.

Aquí listamos los requisitos a cumplir dentro del microservicio:

- Utilizar la tecnología existente para un mejor mantenimiento e integración con la aplicación.
- Mantenerlo simple para un uso de recursos mínimo y que no afecte el rendimiento actual de la aplicación.
- Establecer un sistema de cambios sencillo para la integración de nuevas detecciones.
- Generar un reporte claro, accesible e intuitivo para cualquier persona.
- Hacerlo flexible para poder usarlo en cualquier base de datos.

## 1.4 Estructura del documento

Este documento se organiza de la siguiente forma:

- Tecnologías usadas: explicamos las tecnologías estudiadas para este proyecto.
- Herramientas: hacemos una breve introducción a las herramientas que se necesitan para el desarrollo de la tarea.
- Desarrollo: describimos como organizamos el proyecto:
    - Análisis de los requisitos: explicamos el problema y planteamos la solución.
    - Diseño de la solución: descripción de la solución y como se implementará.
    - Implementación: añadimos una explicación de la elaboración de la solución.
    - Validación: describimos como validamos que la solución da los resultados esperados.

- Despliegue: explicamos la configuración de la puesta en producción de nuestro proyecto siguiendo la práctica CI/CD (Continous Integration/ Continous Delivery)

# 2 TECNOLOGÍAS USADAS

*"El desarrollo web es una sinfonía de tecnologías: bases de datos como base, lenguajes de programación como melodía, y API como conexión armoniosa, todos reunidos para crear sinfonías digitales de innovación". - Tim Berners-Lee*

Las tecnologías usadas para la plataforma IoT fueron seleccionadas para crear una APP web robusta, rápida, sencilla y escalable . El stack principal se basó en el uso de Javascript [3] y NodeJs [4] para la creación de la API web y desarrollo de la lógica BE. La interfaz de usuario y FE son integrados con angular, un framework para el uso de Javascript. La persistencia de datos usa bases de datos no relacionales, como MongoDB, cuyas ventajas en la integración del lenguaje de programación se explicarán más adelante.

En este apartado se presentan las principales tecnologías usadas para el proyecto y se mencionarán algunas de las ventajas de la integración conjunta.

## 2.1 MongoDB

"MongoDB es una base de datos basada en documentos, no relacional. Es de uso gratuito, sus licencias anteriores a octubre de 2018 se publican bajo la licencia AGLP. Las posteriores se harán bajo la licencia *Server Side Public License* (SSPL)" [2].


Figura 3. Logo de MongoDB

Entre sus principales características destacan, el almacenamiento de documentos similares a JSON, indexación y agregaciones en tiempo real y la arquitectura distribuida en su núcleo. [2]

Alguna de la ventajas de usar Mongo con Javascript:

- Documentos BSON (Binary JSON): esto se alinea bien con el formato de datos nativo de Javascript, lo

que facilita el trabajo con los datos de MongoDB directamente dentro de las aplicaciones Javascript.

- Base de datos NoSQL: MongoDB es una base de datos NoSQL, lo que significa que ofrece flexibilidad en el diseño de esquemas y escalabilidad. Esta flexibilidad puede ser particularmente ventajosa en aplicaciones JavaScript donde las estructuras de datos pueden evolucionar con el tiempo o variar entre diferentes partes de la aplicación.
- Naturaleza asíncrona: JavaScript es inherentemente asíncrono, y las operaciones de E/S no bloqueantes de MongoDB encajan bien con este modelo de programación. Esto permite a los desarrolladores de JavaScript realizar operaciones de base de datos de forma asíncrona sin bloquear el bucle de eventos, mejora el rendimiento y la capacidad de respuesta en las aplicaciones web.
- Escalabilidad y rendimiento: la arquitectura distribuida de MongoDB soporta un escalado horizontal adecuado para manejar grandes volúmenes de datos y aplicaciones de alto tráfico. Los desarrolladores de JavaScript pueden aprovechar las características de escalabilidad de MongoDB para crear aplicaciones web robustas y escalables.

## 2.2 JavaScript

Javascript [3] (abreviado como JS) como indica su web de referencia:

> Es un lenguaje ligero, interpretado o compilado con funciones de primera clase. Si bien es más conocido como el lenguaje de programación para páginas web, se usa también en muchos entornos fuera del navegador, tales como NodeJs. Se trata de un lenguaje basado en prototipos, multiparadigma, de un solo hilo, dinámico, con soporte a programación orientada a objetos, imperativa y declarativa. [3]


Figura 4. Logo de JavaScript

Una característica avanzada y a tener en cuenta de JS es su **asincronía**. El manejo de esta se basa en el uso de *promises*, promesas. Las promesas en JavaScript son estructuras que permiten manejar operaciones asíncronas de manera más organizada y comprensible. Inicialmente, una promesa está en estado pendiente hasta que la operación se completa. Las promesas fueron una respuesta a mejorar la legibilidad y estructura del lenguaje anteriormente basado en funciones *callback*.

Una manera de trabajar con las promesas se basa en el uso de las palabras reservadas *async* y *await*. *Async* permite declarar funciones síncronas que con el uso de la palabra *await* detendrán la ejecución del programa hasta la finalización de una promesa cuando sea necesario.

El uso de JavaScript con MongoDB puede ofrecer varias ventajas debido a su soporte e integración nativa, proporciona una manera fluida y potente de interactuar con la base de datos. Ofrece flexibilidad, consistencia y la capacidad de aprovechar todas las capacidades de JavaScript para el desarrollo del lado del cliente y del lado del servidor.

## 2.3 NodeJs

NodeJs [4] es un entorno de ejecución que como dicta su web:

> JavaScript de código abierto y multiplataforma. Ejecuta el motor JavaScript V8 fuera del navegador. Esto permite que Node.js sea muy eficiente. Una aplicación Node.js se ejecuta en un solo proceso, sin crear un nuevo subproceso para cada solicitud. Node.js proporciona un conjunto de primitivas de E/S asíncronas en su biblioteca estándar que impiden que el código JavaScript se bloquee y, en general, las bibliotecas en Node.js se escriben utilizando paradigmas que no bloquean, haciendo que el comportamiento de bloqueo sea la excepción en lugar de la norma. Esto permite a Node.js manejar miles de conexiones concurrentes con un solo servidor sin introducir la carga de administrar la concurrencia de subprocesos, lo que podría ser una fuente significativa de errores. También, los nuevos estándares ECMAScript se pueden usar sin problemas, se decide qué versión ECMAScript usar cambiando la versión Node.js. [4]

Figura 5. Logo de NodeJS

Una de las características a destacar de NodeJs es NPM, *node package manager* [21]. Es el gestor de paquetes que permite la fácil instalación, actualización y manejo de las librerías Javascript. Hay una larga colección de paquetes disponibles lo que permite a los desarrolladores reutilizar código y acelerar el desarrollo. Podemos hacer especial mención de *Moongose* [22], una popular biblioteca de ODM (*Object Data Modeling*) para MongoDB y NodeJs que facilita la interacción con la base de datos a partir de definir esquemas y validación de datos. Así, se estructura y organiza mejor el código de la base de datos.

## 2.4 Docker

La tecnología de contenedores Docker [5] se lanzó en 2013 como un motor de contendores de código abierto, para entender mejor lo que es un contenedor hacemos referencia a su web:

> Un **contenedor** Docker es una unidad estándar de software que empaqueta código y todas sus dependencias para que la aplicación se ejecute de forma rápida y fiable de un entorno informático a otro. Una imagen contenedora de Docker es un paquete de software ligero, independiente y ejecutable que incluye todo lo necesario para ejecutar una aplicación: código, runtime, herramientas del sistema, bibliotecas del sistema y configuración. El software en contenedores siempre funcionará igual, independientemente de la infraestructura. Los contenedores aíslan el software de su entorno y garantizan que funciona de manera uniforme a pesar de las diferencias, por ejemplo, entre el desarrollo y el despliegue. [5]


Esta tecnología permite a cualquier desarrollador crear la imagen Docker de una aplicación, permitiendo que sea portable y ligera para su despliegue y ejecución en cualquier entorno que ya haya instalado esta tecnología. Así, se facilita el diseño de microservicios, haciéndolos consistentes e inmutables pues una vez creada una imagen Docker, el software será predecible.

## 2.5 YAML

YAML [6] que proviene de las siglas de "YAML *Ain't Markup Language*" cuya traducción literal sería "YAML no es un lenguaje de marcado" y diseñado por tres autores en 2001: Clark Evans, Ingy döt Net y Oren Ben-Kiki. Como reza la especificación creada por estos:

> YAML es un lenguaje cruzado, amigable para los humanos, basado en Unicode, un lenguaje de serialización de datos diseñado alrededor de los tipos de datos nativos comunes de los lenguajes de programación dinámica. Es ampliamente útil para funciones de programación que van desde los archivos de configuración, mensajería a través de Internet, la persistencia de objetos, la auditoría de datos y visualización. [6]


Logo YAML

Destacamos la sencillez de aprendizaje, lectura y escritura de este lenguaje que complementa perfectamente la automatización de tareas en CI/CD o el uso de Dockers.

# 3 HERRAMIENTAS

*Una máquina puede hacer el trabajo de cincuenta hombres ordinarios. Ninguna máquina puede hacer el trabajo de un hombre extraordinario.*

*- Elbert Hubbard -*

Las herramientas usadas han sido las manejadas a diario en la empresa, para el trabajo usual. Todas ellas se usan en su versión gratuita, a excepción de una herramienta de sistema de envío de mails y el alojamiento del código. Se instalan en un entorno de trabajo Windows si es necesario o se usa su versión web y en su versión más reciente, no siendo necesario usar alguna versión concreta de las herramientas.

## 3.1 NoSQLBooster

NoSqlBooster [7] es una herramienta visual para la conexión a bases de datos MongoDB. Proporciona una interfaz intuitiva que facilita la interacción con MongoDB a través de un *debugger* para scripts, servicios de monitorización, consultas SQL y mucho más.

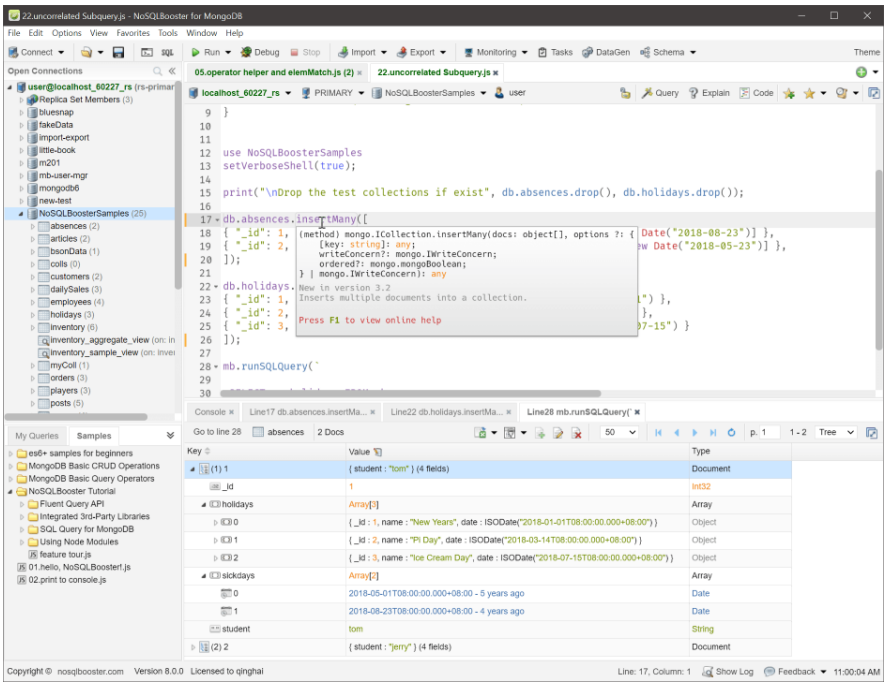


Figura 6. Interfaz de usuario de NoSQLBooster

## 3.2 Visual Studio Code

Visual Studio Code [8] es un editor de código fuente desarrollado por Microsoft que es una de las herramientas más populares entre los desarrolladores. Contiene soporte incorporado para Javascript, TypeScript y NodeJs, y cuenta con una larga lista de extensiones para otros lenguajes y herramientas de programación.

Entre las extensiones que usaremos para el desarrollo tenemos ESlint y GitLens. La primera para establecer un estándar en la escritura del código y la segunda para una mejor visualización del control de versiones.

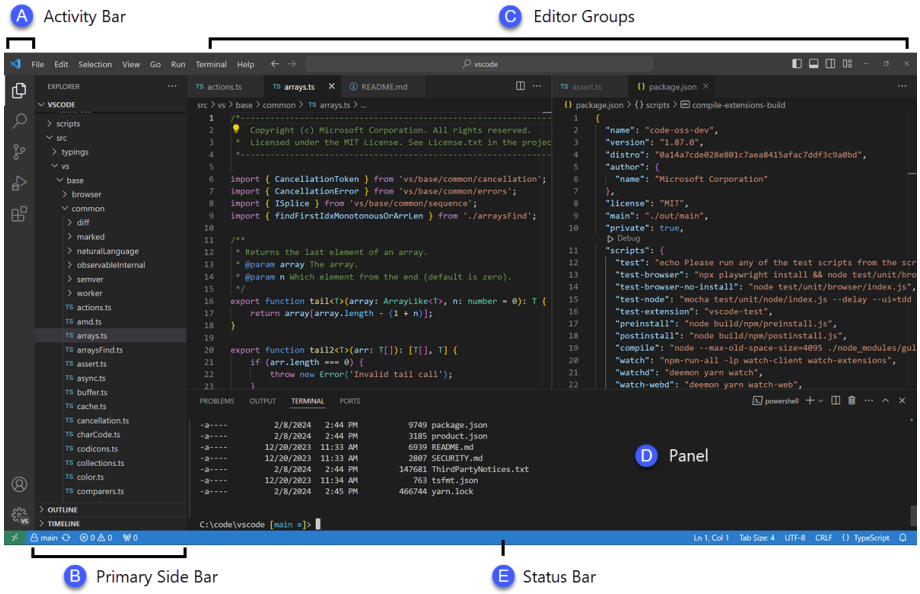



Figura 7. Interfaz de usuario de Visual Studio Code

## 3.3 Bitbucket

Bitbucket [9], desarrollada por Atlassian, es una plataforma de gestión y colaboración basada en Git para el

desarrollo software. Proporciona las herramientas necesarias en la nube para la creación de código pudiendo automatizar test y desplegar código de manera fiable y segura. Entre sus características principales, destacamos:

- PR, Pull Request: característica principal para la revisión y discusión de cambios antes del lanzamiento. Permite tanto la revisión del código, como un control de calidad, documentación, gestión de la integración o notificaciones y seguimiento.
- Integración con Atlassian Suite: es posible integrar otras herramientas como Jira para la gestión de proyectos o Confluence para la documentación.
- Pipelines para integración continua: permite al equipo automatizar la construcción, prueba y despliegue del código desde Bitbucket, facilitando la integración y entrega continua.

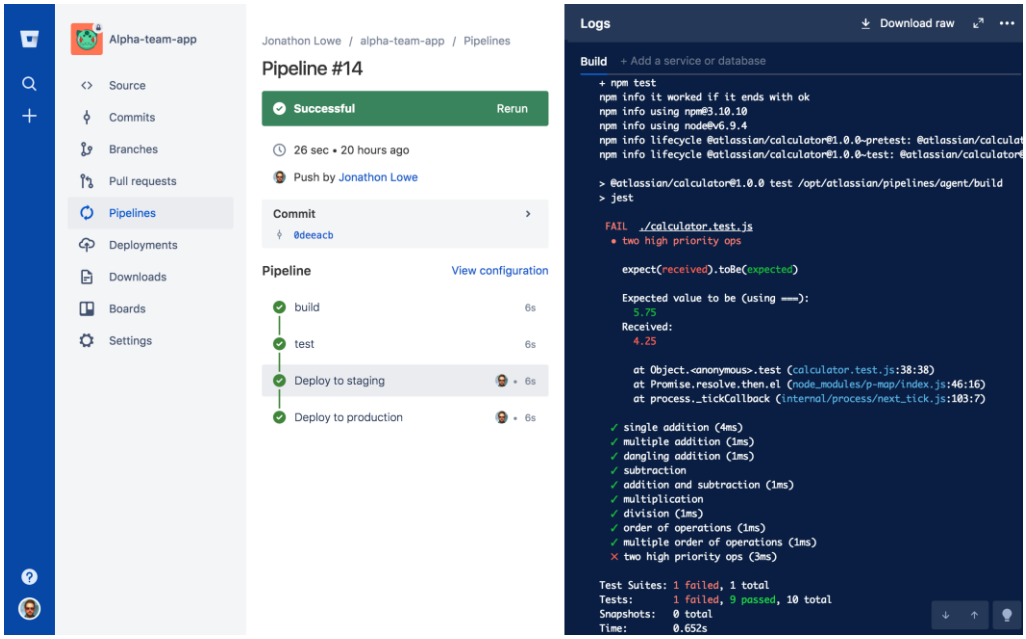



Figura 8. Interfaz de la web de Bitbucket

## 3.4 Docker Desktop

Docker Desktop [5] es una aplicación desarrollada por Docker que permite a los desarrolladores construir, compartir y ejecutar contenedores en su entorno de desarrollo.

La configuración es rápida y sencilla conectándose a recursos remotos. Permite la consistencia y portabilidad, además aísla los entornos y consigue un eficiente uso de los recursos. Todo esto garantiza que las aplicaciones implementadas se ejecuten de manera consistente en diferentes entornos.

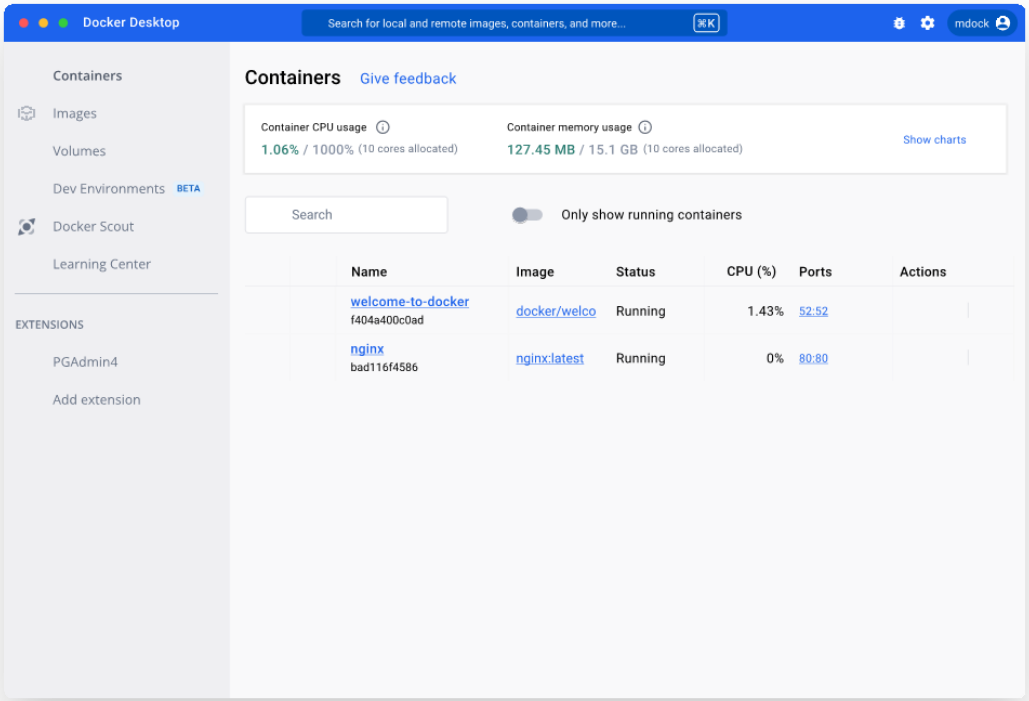



Figura 9. Interfaz de usuario para Docker Desktop

## 3.5 Mailgun

Mailgun [10] es una plataforma de pago diseñada para enviar, recibir y rastrear correos electrónicos en las aplicaciones y servicios web. Es una solución robusta que permite a los desarrolladores integrar fácilmente capacidades de correo electrónico.

Se caracteriza por su posibilidad de envío a alta escala y el seguimiento y analítica de las entregas. Proporciona una API RESTful y SDKs para varios lenguajes de programación, facilitando la integración en aplicaciones web y móviles.


Figura 10. Logo de Mailgun

# 4 DESARROLLO

*Los programas deben ser escritos para que la gente lea,*
*y solo incidentalmente para que las máquinas ejecuten.*

*- Harold Abelson-*

Los requisitos para este proyecto son impuestos por el mismo equipo de desarrollo que lo propone, son mínimos y solo las reglas de negocio pueden ser debatidas con el departamento de producto. Esto es porque la identificación de las incongruencias es responsabilidad del equipo técnico a través de su conocimiento de la lógica de negocio y de la investigación de posibles defectos en el código. Partiendo de esta identificación se confirma con el equipo de producto que se incumple una regla de negocio.

Quiero destacar el concepto de regla de negocio en el que se basa mi trabajo. Cuando hablemos de regla de negocio, nos referimos a una norma especificada por la lógica de negocio que debe cumplir nuestra aplicación. Existen distintas normas según las necesidades del cliente en las características de la aplicación o implícitas en el uso de tecnología SIM y eSIM. No existe un listado de reglas concretas, sino que las identificaremos según los requisitos de la aplicación y la relacionamos con la inconsistencia encontrada. Por ejemplo: según la lógica de negocio un tipo *X* de SIM no está a la venta en Estados Unidos, si en la base de datos un cliente de EEUU tiene esta SIM nos encontramos ante una inconsistencia.

El diseño y optimización para la tarea queda totalmente a elección del desarrollador, dentro de los marcos y el conjunto tecnológico en el que ya se encuentra el equipo.

Este proyecto se desglosa en las siguientes subtareas:

- Análisis de requisitos.
- Diseño de la solución
- Implementación.
- Validación

## 4.1 Análisis de los requisitos

El objetivo principal será la automatización de la identificación de las incongruencias en la base de datos **sin afectar al rendimiento del clúster (sistema de procesamiento)**. Esta identificación ya se hace con el uso de consultas MongoDB basado en documentos JSON:

- *Find*: operaciones utilizadas para recuperar documentos de una colección.
- *Aggregate*: operaciones de agregación para analizar datos y con una estructura más compleja que el *find*.

Estas consultas se suelen ejecutar a través de algún IDE (integrated development environment) para la gestión de MongoDB como NoSQLBooster o a través de la creación de scripts durante la resolución e investigación de fallos. Ahora, todas estas consultas se automatizarán para ser aplicadas por el servicio a desarrollar y que este a su vez sea de fácil configuración y escalabilidad para futuras integraciones. Además debe generar un reporte o

alerta al equipo de desarrollo.

Ahora mismo, la infraestructura en la que se ejecuta el BE en comunicación con la base de datos es la siguiente:

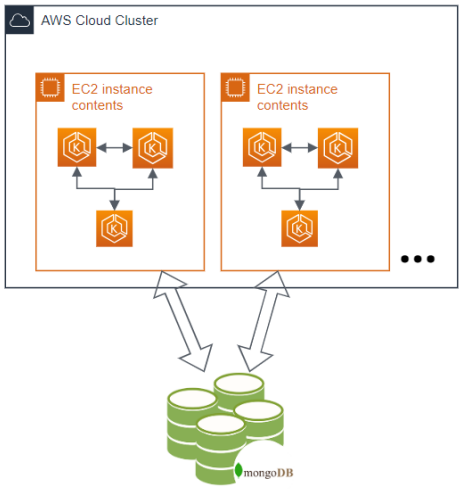



Figura 11. Infraestructura del Clúster

Se trata de una aplicación en contenedores Docker corriendo en la nube de Amazon Web Services y orquestados por Kubernetes [16] (software de código libre que permite el manejo y automatización del despliegue, escalabilidad y manejo de contenedores). Todos los servidores conforman el clúster de Kubernetes principal y su configuración se hace a través de la configuración del Docker y distintos archivos de configuración de Kubernetes.

Nuestro servicio no será necesario que permanezca corriendo constantemente, solo que se ejecute en un instante determinado. Se elegirá la solución, CronJob: una tarea programada para ejecutarse en un tiempo específico y regular. Esto nos ofrecerá flexibilidad y precisión en la ejecución de la lógica desarrollada.

Nuestra solución será la encargada de la ejecución de las consultas a la base de datos y de la generación de un informe. Este informe, incluirá el número de incongruencias encontrados para las normas de negocio conocida, **siendo 0 lo correcto y cualquier otro número, un motivo de investigación para resolver la incongruencia**.

Para el envío de este informe se usará la misma tecnología que ya se usa en el stack de la aplicación, Mailgun. Se enviará vía mail y para su fácil interpretación, en lugar de crear un archivo y adjuntarlo, se planea **generar una tabla HTML** que es fácilmente interpretada por cualquier aplicación de lectura de mails o vía web.

Por tanto, crearemos la imagen Docker de nuestro CronJob y añadiremos el contenedor al clúster como un componente de la aplicación que se ejecutará una vez a la semana, quedando integrado como muestra la siguiente imagen:

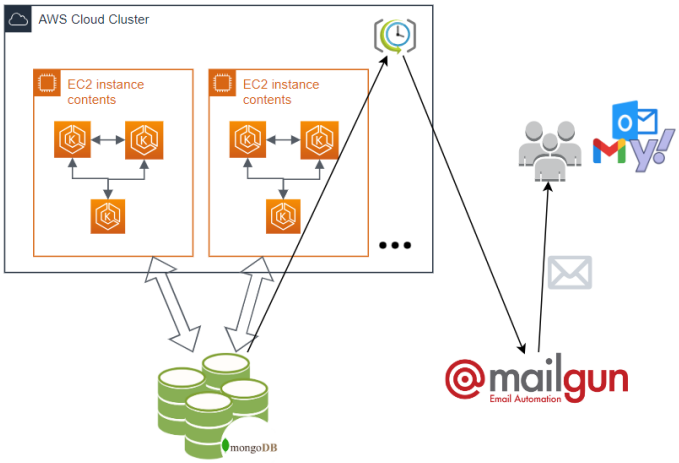



Figura 12. CronJob integrado en la infraestructura

## 4.2 Diseño de la solución

Vamos a hacer el diseño lo más simple posible. Como contamos con dos tipos de consultas a la base de datos, diseñamos dos tipos de modelos para la lectura de la misma. MongoDB se compone de documentos organizados en colecciones (equivalente a las tablas de las base de datos SQL). Las reglas de negocio pueden afectar a una colección o a su relación con otras colecciones, pero las consultas MongoDB siempre tendrán de base los documentos de una colección. Por tanto añadiremos en el modelo el nombre de la colección de la que partimos para conseguir flexibilidad a la hora de integrar nuevas consultas y abarcar cualquier colección.

Cada modelo deberá contener:

- Descripción: una pequeña explicación sobre la inconsistencia que se está consultando, debe hacer referencia a la regla de negocio que se incumplió y se intenta evitar, un tipo *string*.
- Colección: la colección analizada, un tipo *string*.
- Consulta: la consulta a realizar, un objeto en el caso de *find* o un tipo *array* en el caso del *aggregate*.

Para facilitar la reutilización, todas las instrucciones se encontrarán dentro de un array de objetos, uno por cada instrucción para la detección de la inconsistencia, indexadas por la descripción y compuesto por un array. Este array incluye dos elementos, el nombre la colección y la consulta a realizar. Cada modelo se especifica en un archivo que se exportará como un módulo de NodeJs (*module.exports*) para su uso en las clases definidas.

Esta será por tanto la estructura para los modelos:

- Consultas find:

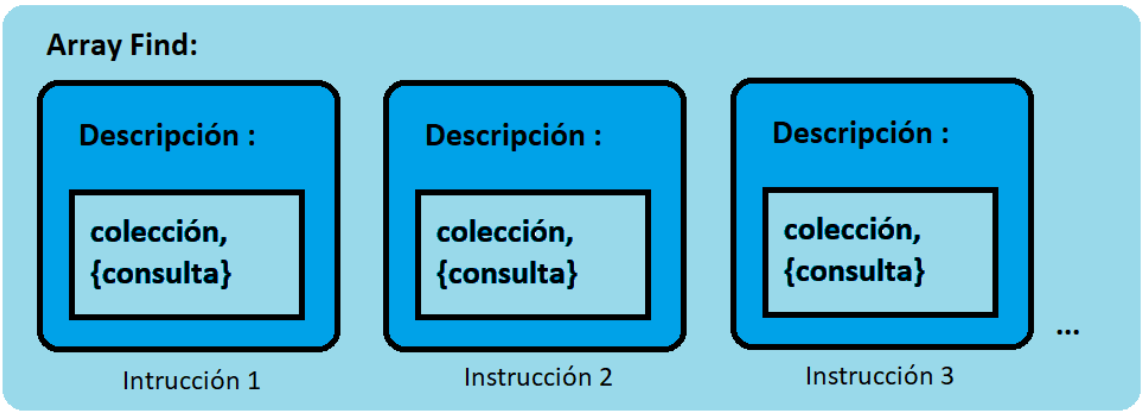



Figura 13. Modelo de consultas find

- Consultas aggregate:

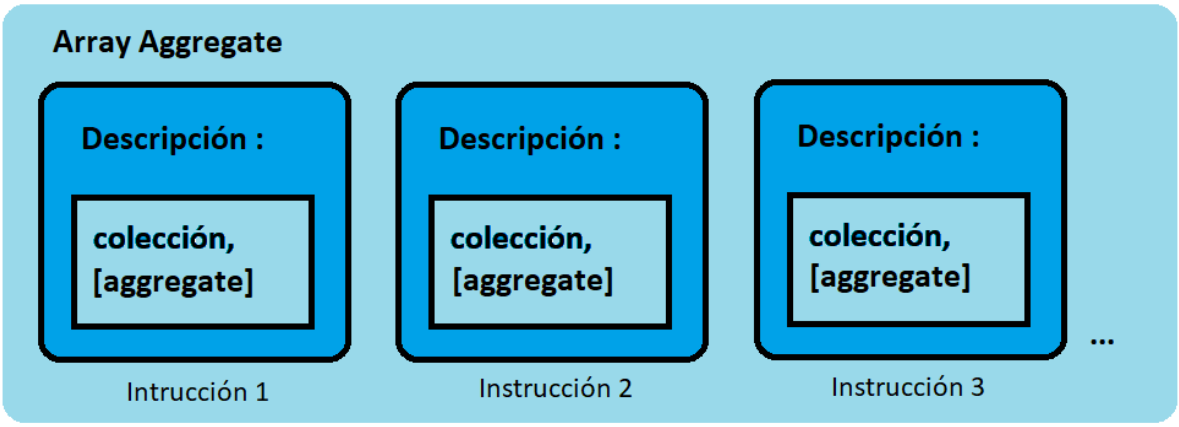



Figura 14. Modelo consultas aggregate

La estructura del modelo se elige así para que cuando sea necesaria la codificación de nuevas detecciones, se pueda añadir como una instrucción adicional en el array correspondiente, según la consulta propuesta, junto a la descripción que la identifica y la colección a la que afecta. También, el Array permitirá al controlador del CronJob poder iterar fácilmente con las instrucciones y manejar los resultados de manera sencilla.

Para el servicio de mail y el de base de datos se crearán dos clases con los atributos y métodos mínimos para la creación del reporte. Estos hacen usos de librerías externas y cuya relación con cada clase se explica a continuación.

Clase Database y su uso de la dependencia MongoClient [11]:

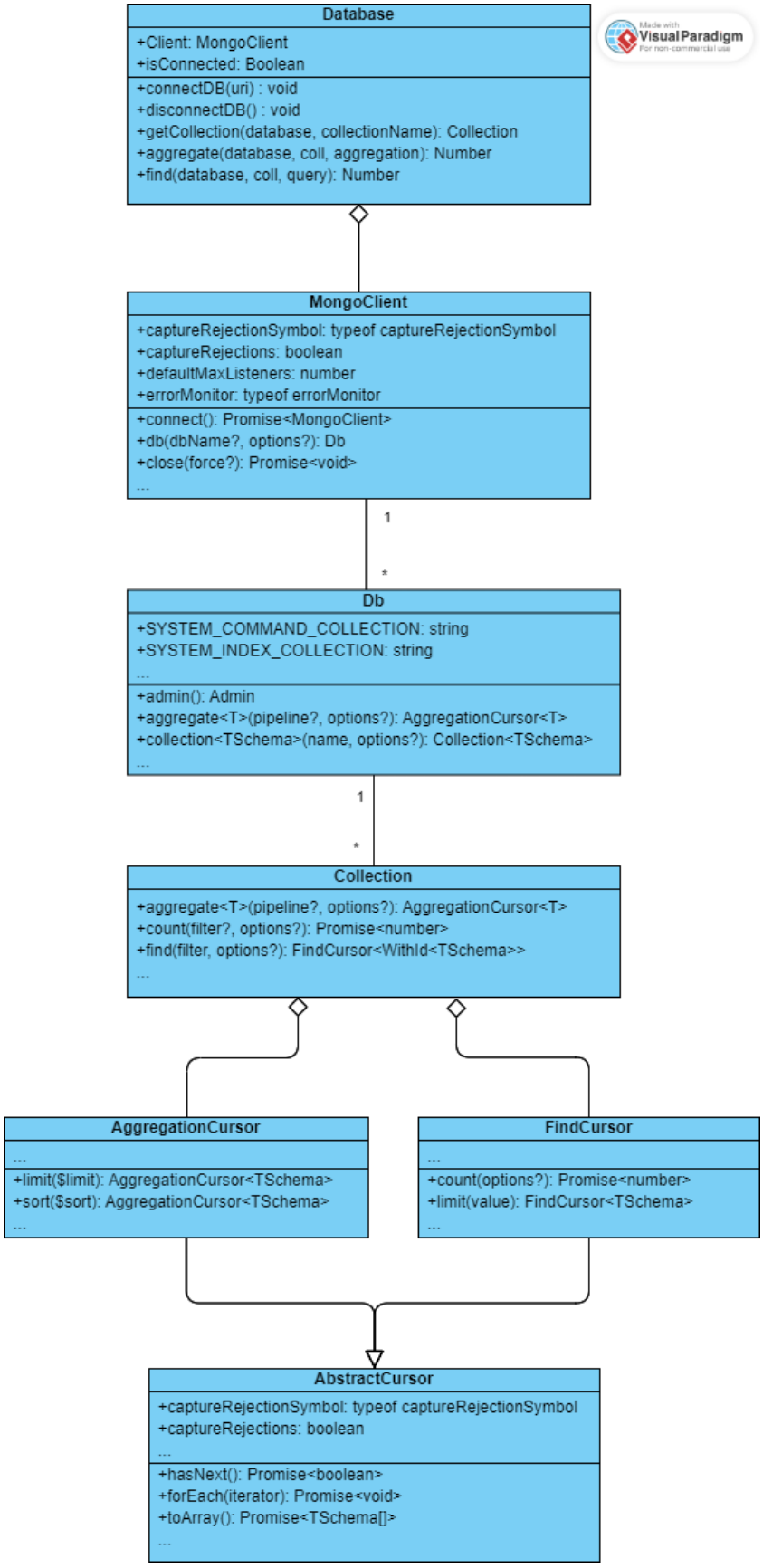



Figura 15. Diagrama de clases de la clase Database.

Hay que tener en cuenta que algunas de las clases que se usan son complejas y se ha evitado añadir todos los detalles para mantener la claridad del diagrama, reemplazando información sin relevancia para este proyecto por tres puntos suspensivos.

Clase Mail y el uso de las dependencias axios [12] y form-data [13]:

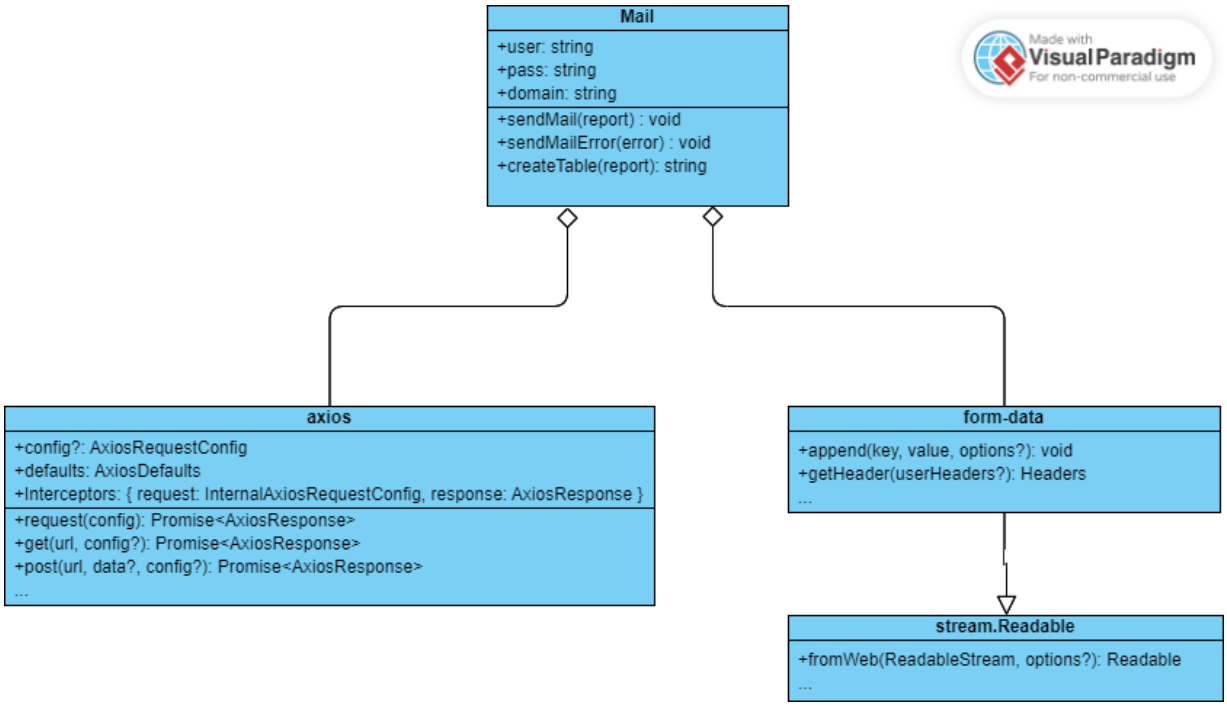



Figura 16. Diagrama de clases para Mail.

Tanto los modelos como las clases serán usados por un controlador quien aplicara la lógica para el CronJob. Mostramos como interactúan en el siguiente diagrama de secuencia:

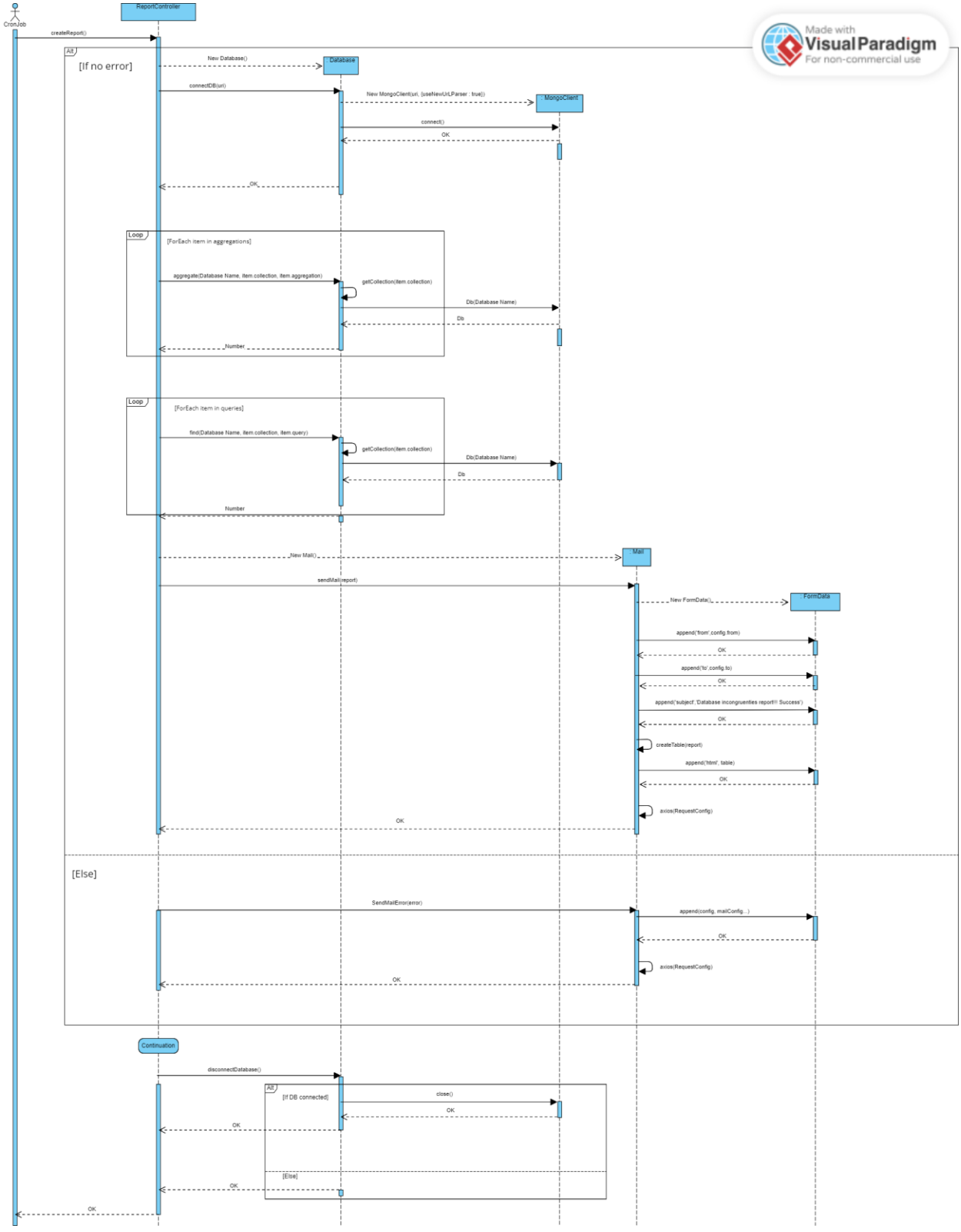



Figura 17. Diagrama de secuencia del controlador

Como podemos observar las tareas que se completan, si no hubiera ningún error, cuando comienza la tarea programada son:

- Conectarse a la base de datos usando la clase Database que a su vez instancia la clase MongoClient.
- Para cada una de las agregaciones, lanzamos la consulta a la base de datos y guardamos el resultado.
- Para cada una de las consultas simples, lanzamos la consulta a la base de datos y guardamos el

resultado.

- Enviamos el mail con la tabla resultado usando la clase Mail que instancia la clase formData para configurar los datos del e-mail.
- Finalmente, si estamos conectados a la base de datos, cerramos al conexión.

Si hay algún error durante el proceso, se envía un e-mail informando del error encontrado.

## 4.3 Implementación

Gracias al soporte nativo de Visual Studio Code para NodeJs, con el instalador de paquetes NPM, podemos usar el comando:

```
Npm init
```

Te ayuda a rellenar el archivo, package.json, con la información y configuración del proyecto. Entre ellas:

- Scripts: npm scripts para lanzar el proyecto con distintos comandos para fines diferentes, como lanzar las pruebas de integración.
- Dependencias: librerías usadas en el proyecto y que en su instalación serán necesarios.
- *devDependencies*: librerías usadas para el desarrollo pero que no son necesarias para ejecutar el proyecto.

Dependiendo de la versión npm, te pedirá la siguiente información:

- package name: (test) databaseincongruenties
- version: (1.0.0) 1.0.0
- description: Incongruenties in the hummingBird database, report sent via mail
- entry point: (index.js)
- test command
- git repository
- keywords:
- author: cesar.david@podgroup.com
- license: (ISC)

Previamente se creó un nuevo repositorio en Bitbucket para este proyecto y el directorio raíz se clonó en nuestro entorno de desarrollo. Gracias al controlador de paquete NPM, cualquier librería necesaria e instalada se añadirá al *package.json*.

Nuestra estructura de directorios será:

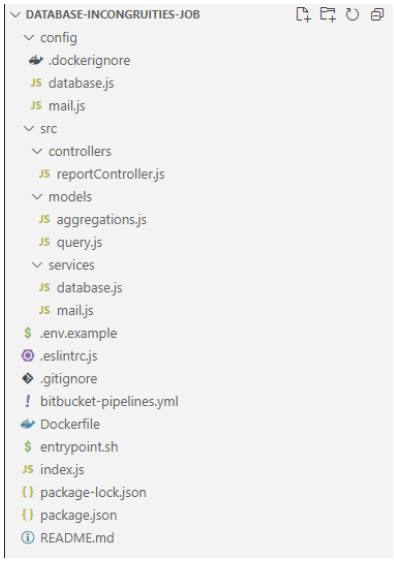


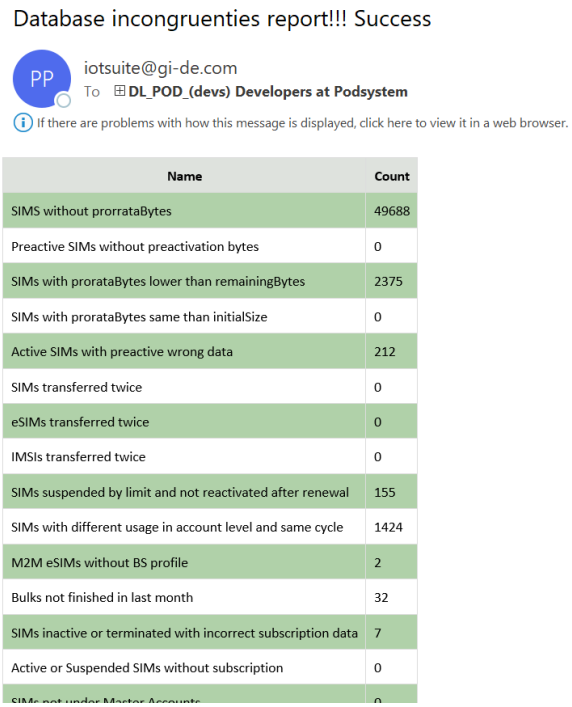

Figura 18. Estructura de los directorios.

Siendo index.js el script de partida y añadido en el *package.json* como "start" para el comando:

```
Node index.js
```

En el directorio ***/src*** incluimos el código principal organizados en el subdirectorio ***/controllers*** para el controlador del reporte, ***/models*** para los modelos de las agregaciones y las consultas y ***/services*** para las dos clases: *database* y *mail*. Algunos detalles sobre el código implementado:

- Todas las funciones usadas serán asíncronas, permitiendo el uso de *await* para la finalización de alguna acción específica.
- Los métodos *aggregate* y *find* de la clase *Database*, devuelven el número de incongruencias encontrado.
- Para el reporte en el mail, se crea una tabla HTML con el reporte pasado por el controlador (array del par descripción y numero de incongruencias) y que muestra un interlineado de color para una mejor apreciación del contenido.



Database incongruenties report!!! Success

PP iotsuite@gi-de.com
To DL_POD_(devs) Developers at Podsystem
If there are problems with how this message is displayed, click here to view it in a web browser.

| Name | Count |
|---|---|
| SIMS without prorrataBytes | 49688 |
| Preactive SIMs without preactivation bytes | 0 |
| SIMs with prorataBytes lower than remainingBytes | 2375 |
| SIMs with prorataBytes same than initialSize | 0 |
| Active SIMs with preactive wrong data | 212 |
| SIMs transferred twice | 0 |
| eSIMs transferred twice | 0 |
| IMSIs transferred twice | 0 |
| SIMs suspended by limit and not reactivated after renewal | 155 |
| SIMs with different usage in account level and same cycle | 1424 |
| M2M eSIMs without BS profile | 2 |
| Bulks not finished in last month | 32 |
| SIMs inactive or terminated with incorrect subscription data | 7 |
| Active or Suspended SIMs without subscription | 0 |
| SIMs not under Master Accounts | 0 |

Figura 19. Tabla con el reporte de las inconsistencias

- El controlador maneja cualquier error y en ese caso, se envía un correo con la excepción encontrada.

La complejidad y el mayor tiempo invertido para el desarrollo se encuentra en la creación de las consultas y las agregaciones que se sumarán a las ya existentes para detectar inconsistencias comunes. A tener en cuenta sobre estas últimas, se trata de una consulta elaborada, que puede constar de una o varias etapas. Todas ellas se enfocan siguiendo un modelo de pipeline (tubería), donde los documentos pasan y sufren una transformación de acuerdo con la tarea específica de esa etapa. Entre las etapas más comunes que usaremos encontramos:

- **$match**: Filtra documentos según los criterios especificados (similar a una consulta con *find*).
- **$group**: Agrupa documentos por un campo especificado y puede realizar operaciones agregadas como sumas y promedios.
- **$project**: Selecciona y reformatea los campos que se van a devolver.
- **$limit**: Limita el número de documentos que se devuelven.
- **$unwind**: Descompone un array en documentos individuales.
- **$lookup**: Realiza un *join* con otra colección.
- **$count**: Cuenta el número de documentos.

El resultado de nuestras consultas a la base de datos debe ser el número de inconsistencias encontradas. Esto implica que para las consultas tipo agregación la última etapa siempre será:

```
{
    $count : 'total',
},
```

Siendo *total* el campo esperado por el controlador para formalizar el reporte (recordemos compuesto por la descripción de la inconsistencia y el número de ocurrencias) y que procesará la clase mail. La consultas sencillas o *query* invocarán al final de la llamada a la base de datos, el método *count()* para obtener el número de documentos retornados.

Antes de comentar las instrucciones diseñadas y las reglas de negocio identificadas. Mencionamos algunos conceptos de la empresa:

- Cliente: particular o empresa que contrata nuestros servicios y obtiene una cuenta en la plataforma IotSuite. Existen dos tipos, vendedores y clientes, teniendo los vendedores la posibilidad de ofrecer cuentas en la plataforma bajo su marca.
- Cuenta: acceso personalizado de la plataforma IotSuite con los permisos necesarios para la gestión de los activos contratados.
- Activo: recursos que vende la empresa a los clientes. Principalmente dos componentes físicos, SIM y eSIMs.
- Producto: relativo a la conectividad, es el plan que contrata un cliente para aplicar lo a un activo y que se configura de acuerdo a las necesidades del cliente.

Cualquiera de nuestros activos pueden pertenecer a los clientes de nuestra empresa pero estas pueden actuar a su vez como vendedoras y proveer activos a sus propios clientes usando la plataforma bajo su control.
Por ello, los modelos de las SIMs y eSIMs mantienen campos como un array de objetos identificados por el Id de un cliente y manteniendo la jerarquía de pertenencia. El FE usa estos campos para mostrar información según la cuenta que está siendo manejada en la plataforma. Para un mejor entendimiento, la siguiente ilustración muestra un hipotético escenario cumpliendo la jerarquía de cuentas:

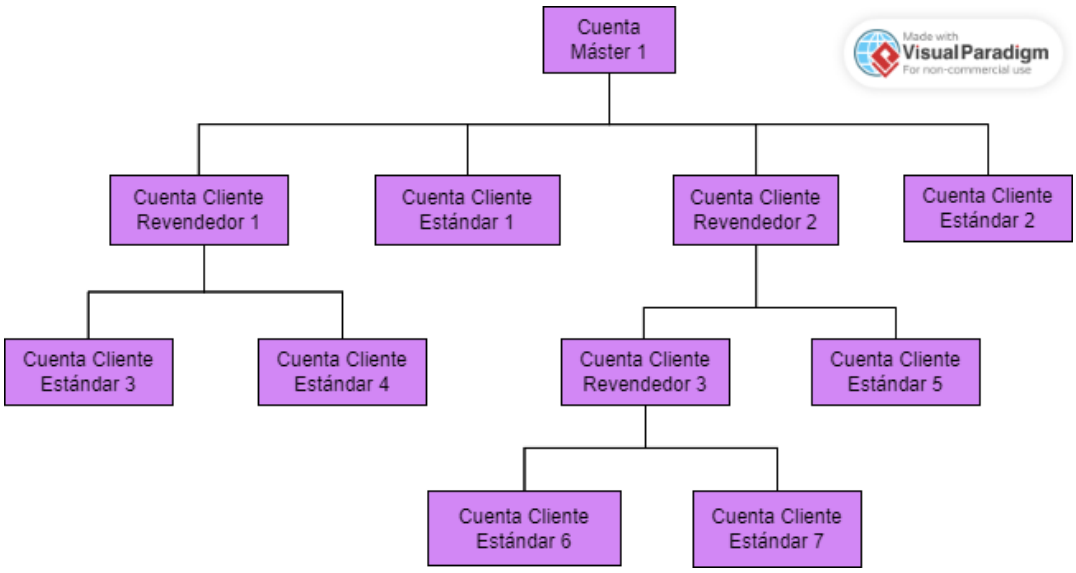



Figura 20. Organigrama de cuentas ejemplo.

Una SIM perteneciente a la cuenta ‘Cuenta Cliente Estándar 6’ se creó en ‘Cuenta Máster 1’ y se provisionó a la una cuenta hija hasta ser provisionada en el cliente final.

Algunos modelos de las colecciones de nuestra base de datos que son importantes para entender nuestras consultas son:

- AssetSimCard:

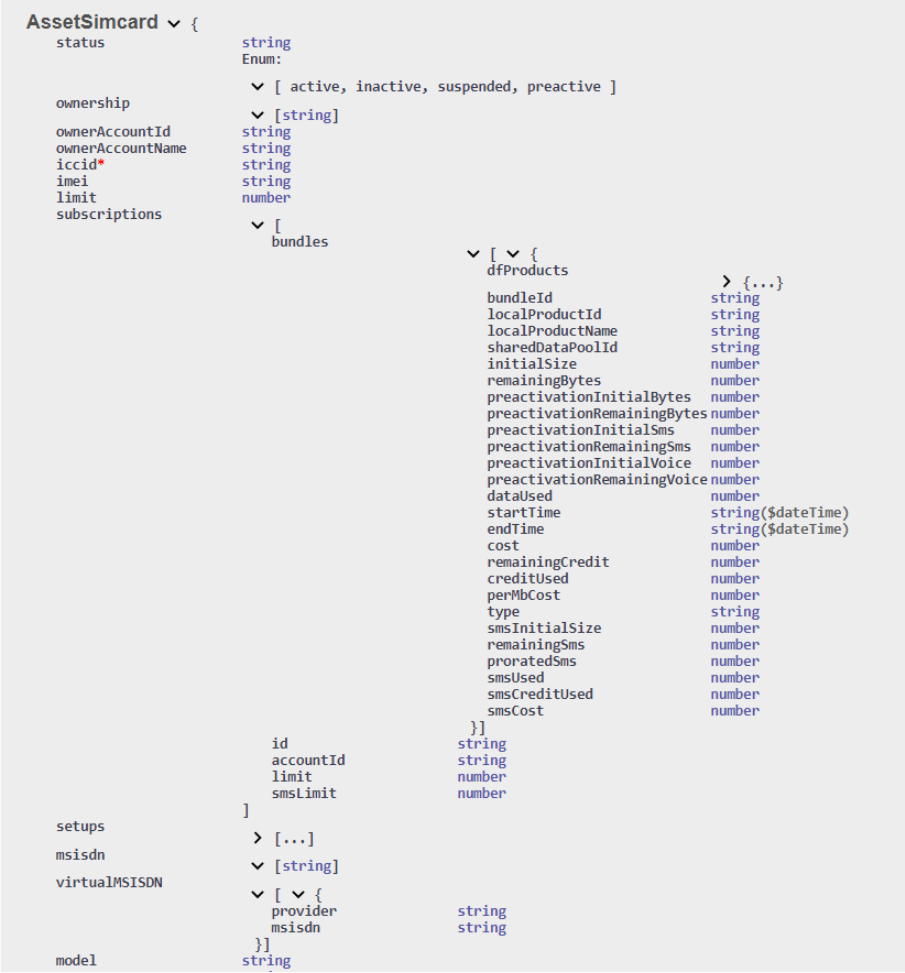



Figura 21. Modelo AssetSimCard

Donde destacamos la complejidad del campo *subscriptions*, un array de objetos con información referida al producto asignado a un activo para su uso manteniendo la jerarquía de cuentas. A su vez, guarda otro array, *bundles*, que conserva detalles más concretos del producto (por ejemplo, el uso de datos).
Cada documento indica un activo del tipo SIM para distintos modelos.

- Bulk:

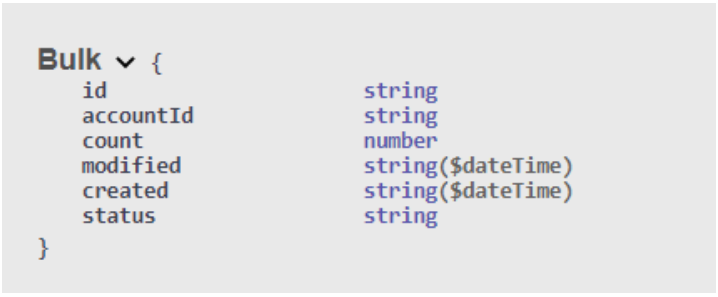



Figura 22. Modelo Bulk

- Imsi:

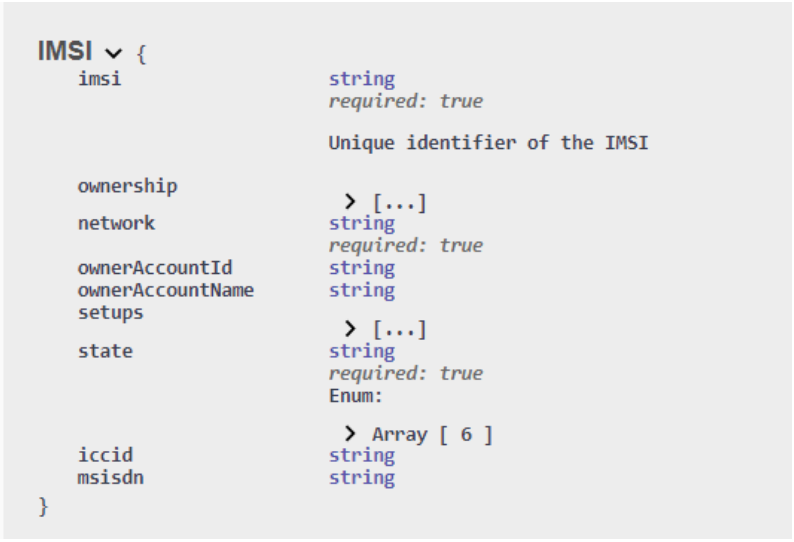



Figura 23. Modelo IMSI

Comentamos algunas de las reglas de negocios y las peticiones a la base de datos creadas para verificar la consistencia conforme a las mismas, añadiremos una diferenciación entre las reglas impuestas por el equipo de producto y las añadidas por el equipo de desarrollo:

- **Active SIMs with preactive wrong data:**

  Regla de producto: "Una SIM activada debe tener 0 *preactive* bytes disponibles".

  Una SIM puede suscribirse con un producto configurado con *preactive bytes*, esto es un plan de datos disponible antes de la facturación y que se traduce como un estado anterior al *active, preactive.* Cuando abandone su preactivación y pase a estado *active* tiene que reiniciar los campos configurados para el estado *preactive*.

  Para ello usaremos una agregación, partimos de una etapa *match* y la colección *assetsimcards*, donde obtenemos todos los documentos:

  - Cuyo estado sea *active*.
  - Se encuentre en un plan con datos *preactive* (comprobando la existencia del campo *'subscriptions.bundles.dfProducts.PreActivationProduct'* ).
  - Disponga de bytes de preactivación disponibles y el producto se encuentre en preactivación. (comprobamos los detalles del producto en el campo '*subscriptions.bundles'* para los datos '*preactivationRemainingBytes*' e '*inPreactivation'*)

  Estos filtros podrían ser suficientes pero tenemos que tener en cuenta la jerarquía de pertenencia, una cuenta puede haber configurado el plan preactivo pero su hija no. Debemos hacer una nueva comprobación sobre la información para cada posible cuenta.

  Agregamos una etapa *project* para limpiar datos que no necesitemos y aplicaremos la etapa *unwind* para el campo '*subscriptions'*, desglosando el array de subscripciones con la información del producto en cada cuenta. Para terminar con el desglose, añadimos una etapa *unwind* sobre el campo '*subscriptions.bundles'*.

Por último, aplicamos la etapa *match* de nuevo confirmando que obtenemos los documentos afectados a nivel de cuenta y no hay una mezcla de los datos en la primera consulta.

```
[
    {
        '$match' : {
            'status' : 'active',
            'subscriptions.bundles.dfProducts.PreActivationProduct' : {
                '$exists' : true,
            },
            '$or' : [
                {
                    'subscriptions.bundles.preactivationRemainingBytes' : {
                        '$ne' : Number(0),
                    },
                },
                {
                    'subscriptions.bundles.inPreactivation' : {
                        '$ne' : false,
                    },
                },
            ],
        },
    },
    {
        $project: {
            'iccid': 1,
            'subscriptions': 1,
        },
    },
    {
        '$unwind' : {
            'path' : '$subscriptions',
        },
    },
    {
        '$unwind' : {
            'path' : '$subscriptions.bundles',
        },
    },
    {
        '$match' : {
            'subscriptions.bundles.dfProducts.PreActivationProduct': {'$exists': true},
            '$or' : [
                {
                    'subscriptions.bundles.preactivationRemainingBytes' : {
                        '$ne' : Number(0),
                    },
                },
                {
                    'subscriptions.bundles.inPreactivation' : {
                        '$ne' : false,
                    },
                },
            ],
        },
    },
    {
        $count : 'total',
    },
]
```

- **SIMs transferred twice:**

Regla de desarrollo: “el identificador de cuenta en el campo de pertenencia de una SIM debe ser único y no puede repetirse.”

Una SIM vendida a un cliente tiene que ser provisionada en su cuenta con el uso de una característica llamada transferencia. Se transfiere desde la cuenta raíz a la cuenta hijo, haciendo disponible el activo para esta última. Este acto de transferencia añade un objeto de información, identificado por el Id del cliente, a algunos campos del tipo array. El campo *ownership* puede presentar duplicidad de elementos usando el mismo identificador debido a una condición de carrera y es necesario identificarlas.

Creamos una agregación partiendo de la colección *assetsimcards*, aplicamos una primera etapa, *match*, para filtrar todas las SIMs que hayan sido vendidas y no pertenezcan solo a la cuenta de la empresa (es decir, existe más de un elemento de pertenencia en el campo *ownership*). A continuación, limpiamos la información que no necesitemos con *$project* y desglosamos el array de pertenencia con *$unwind.* A partir de aquí tenemos un documento con el ICCID y un identificador de la cuenta, si los agrupamos para saber las veces que se repite esta relación obtendremos los duplicados. Usamos la etapa *$group* por el campo *iccid* y *ownership* añadiendo la ocurrencias a un nuevo campo llamado *count* y por último aplicamos un filtro *$match* para obtener aquellos documentos donde la ocurrencia del agrupamiento ha sido mayor que uno.

```
[
    {
        $match: {
            'ownership.2': { $exists: true },
        },
    },
    {
        $project: {
            'iccid' : 1,
            'ownership' : 1,
        },
    },
    {
        $unwind: {
            path: '$ownership',
        },
    },
    {
        $group: {
            _id: {
                'account': '$ownership',
                'iccid': '$iccid',
            },
            count: { $sum: 1 },
        },
    },
    {
        $match: {
            count: { $gt: 1 },
        },
    },
    {
        $count : 'total',
    },
]
```

- **SIMs inactive or terminated with incorrect subscription data:**

Regla de producto: **“**una SIM inactiva o terminada no debe contener ninguna traza de un producto que pudo aplicársele en algún momento de su ciclo de vida.”

El ciclo de vida de una SIM parte desde su estado *inactive* para la venta y después de su uso puede volver a reiniciarse a este estado o aplicarse el estado *terminated* y dar fin a su vida útil.

Usaremos una agregación partiendo de la colección *assetsimcards*, donde la primera etapa será un filtrado *$match* para obtener las SIMs con estado *inacive* y *terminated*. Añadimos una etapa $unwind para desglosar el array de subscripciones donde guardamos la información relativa a los productos por cuenta y volvemos a aplicar un filtro $match para obtener los documentos que no tengan el campo *bundles* como un array vacío.

```
[
    {
        $match: {
            'status': {$in : ['inactive', 'terminated']}, //indexed

        },
    },
    {
        $unwind : {
            'path' : '$subscriptions',
        },
    },
    {
        $match : {
            'subscriptions.bundles': {$ne : []},
        },
    },
    {
        $count : 'total',

    },

]
```

- **SIMs with different usage in account level and same cycle:**

Regla de desarrollo: "el uso de una SIM es el mismo para todas las cuentas si el ciclo de facturación es el mismo."

SIMs con diferentes clientes en su jerarquía de pertenencia, guardan el uso de datos para cada una de sus cuentas en la información relativa al producto aplicado en ellas (campo *bundles*). Este uso de datos podría ser distinto si el ciclo de facturación de una cuenta a otra es distinto pero si es el mismo, el uso debe ser el mismo para todas ellas.

Este caso implica el uso de una agregación, partiendo de la colección *assetsimcards,* donde aplicando una etapa *$match* vamos a obtener las SIMs activadas y que pueden crear consumo. Desglosamos con *$unwind* el array de subscripciones para cada cuenta y después lo repetimos con el array *bundles* con los detalles del producto. Limpiamos la información que no necesitamos y liberamos espacio de procesamiento usando *$project.*

Los documentos obtenidos los agruparemos, etapa *$group,* usando como criterio el *iccid, startDate* y *endDate,* siendo los dos últimos campos los que marcan el ciclo de facturación. En la agrupación crearemos un nuevo campo llamado *distinctValues* y que añadirá los distinto valores del uso de datos en cada cuenta.

Por lo tanto, tenemos para las SIMs con un mismo ciclo de facturación, un array con los distintos valores del uso de datos. Una SIM incorrecta presentará en este array un tamaño mayor que uno. Para obtener el tamaño del array y aplicarle esta condición, usaremos la etapa *$project,* donde añadimos un nuevo

campo, *hasDifferentValues*, con una condición de mayor que 1 que devolverá verdadero o falso si el operador *$size* para la longitud de un array lo cumple. Por último, filtramos con *$match* todos aquellos documentos que hayan establecido el último campo creado como *True*.

```
[

    {
        '$match': {
        'status' : 'active'}} // indexed
    ,
    {
        '$unwind': {
            'path': '$subscriptions',
    }},
    {
        '$unwind': {
            'path': '$subscriptions.bundles',
    }},
    {
        $project:{
            iccid: 1,
            'subscriptions.bundles.dataUsed' : 1,
            'subscriptions.bundles.startTime' : 1,
            'subscriptions.bundles.endTime' : 1,
        },
    },
    {
        $group: {
             _id:{
                id : '$iccid',
                startDate : '$subscriptions.bundles.startTime',
                endDate : '$subscriptions.bundles.endTime',
             },
        distinctValues: {
            $addToSet: '$subscriptions.bundles.dataUsed' ,
        },
    },
    },
    {
        $project: {
            _id: 1,
            distinctValues : 1,
            hasDifferentValues: { $gt: [{ $size: '$distinctValues' }, 1] },
    },
    },
    {
        $match:
        { hasDifferentValues: true },
    },
    {
        $count : 'total',
    },
 ],
```

- **Bulks not finished in last month:**

Regla de desarrollo: “una acción en *bulk* debe terminar, ya sea como éxito o con error.”

Algunas de las acciones que puede realizar una SIM, pueden lanzarse por lotes evitando tener que ejecutar una acción por elemento. Para una lista de SIMs o filtro, puedes aplicar la misma acción siempre y cuando te lo permita la plataforma. Estas acciones conllevan una notificación vía mail al terminar. Las acciones en *bulk* no contienen ninguna restricción de limite sobre los activos afectados, es una acción que corre en segundo plano. Si el hilo de la operación se rompe en el servidor por cualquier razón, puede provocar que la acción quede en un estado incompleto sin notificar. Cuando se comienza a realizar, la

*bulk action* tiene un estado *pending*, una vez finalizado se aplica el estado *success* si todo fue correcto y en caso contrario *error* si hubo alguna operación que falló.

Usamos una agregación partiendo de la colección *bulks* para poder usar dinámicamente la creación de una fecha, clase *Date()*, y algunos de sus métodos que aplicaremos en una etapa *$match* filtrando las operaciones creadas en el último mes que se encuentran aún en estado pendiente.

```
[
    {
        $match : {
            created: { $gt : new Date(new Date().getFullYear(), new Date().getMonth() - 1)}
            'status': 'pending',
        },
    },
    {
        $count : 'total',
    },
]
```

- **IMSIs with same MSISDN and IMSI.**

Regla de producto: "el valor de la IMSI y el MSISDN deben ser distintos para una SIM."

Un activo concreto perteneciente a la SIM, el IMSI, se ha detectado que presenta incongruencias en su creación, puede venir de servicios externos y no existe un control de los datos específico. Usamos una consulta simple, *query* para encontrar en la colección de activos IMSI, cualquier documento que tenga igual valor en el campo *imsi* y *msisdn*, para ello aplicamos el operador de igualdad: *$eq*.

```
{
    $expr:
    { $eq: ['$imsi', '$msisdn'] },
}
```

- **SIMs with the contract expired.**

Regla de producto: "una SIM con una fecha de fin de contrato anterior que la fecha actual debe estar en estado *inactive*."

Un producto puede marcar el final del uso de una SIM y su vuelta al estado inactivo a través de la configuración de un tiempo de contrato. Cumplido este contrato, la SIM retorna a su estado inactivo.

Realizaremos una consulta simple, *query*, y filtraremos todas las SIMs que no se encuentren en un estado relacionado a la aplicación de un producto y cuya campo *endTimeContractLength*, con la fecha de expiración del contrato, sea inferior a la fecha actual en la que se realiza la consulta.

```
{
    'status' : {$in : ['active', 'suspended', 'preactive']} ,
    'endTimeContractLength': { $lt: new Date() },
}
```

Por último, comentamos que hay variables para las clases que han sido configuradas como variables de entorno y que se añadirán en el momento de despliegue cuando se ejecute la imagen del cronJob a añadir en el clúster. Entraremos más en detalle sobre la configuración del entorno en el apartado de despliegue. Para obtenerlos en el código se ha añadido en una carpeta llamada */config* y que por cada clase exportará una variable configurada con sus variables de entorno correspondientes:

Clase Mail config:

```
module.exports = {
    api_key: process.env.MAILGUN_KEY,
    domain: process.env.DOMAIN,
    from: process.env.EMAIL_FROM,
    to: process.env.EMAIL_TO,
}
```

Clase Database config:

```
module.exports = {
    uri: process.env.MONGO_URI,
    database: process.env.DATABASE,
}
```

## 4.4 Validación

Validaremos cada agregación y consulta diseñada con la herramienta NoSqlBooster [7] y con DockerDesktop [5] lanzaremos una base de datos local de MongoDB. Para ello, después de la instalación de Docker en el PC de desarrollo, descargaremos e instalaremos una imagen de MongoDB versión 6.0. Gracias a los *backups* de la empresa para las colecciones de la base de datos, reconstruiremos una copia en nuestra base de datos local y configuramos NoSqlBooster para conectarse a ella.

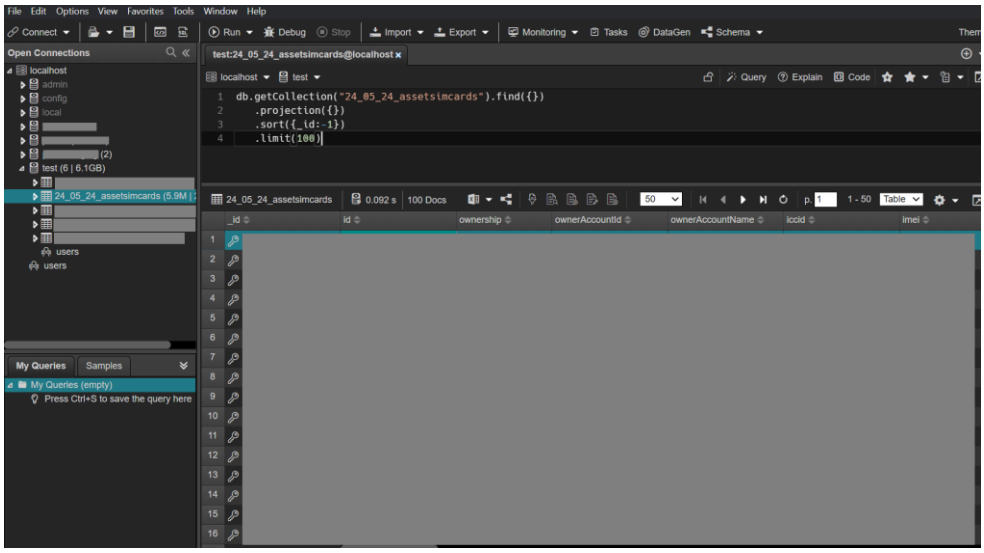



Figura 24. NoSQLBooster conectado a la base de datos local

Una vez que se ha confirmado que las consultas funcionan y dan el resultado esperado (simulamos incongruencias modificando los documentos de nuestra copia en local), conectamos NoSqlBooster con nuestra base de datos de producción. Los permisos del usuario con el que accedemos son de solo lectura, evitando comportamientos inesperados.

Estudiamos en estas pruebas, el rendimiento de nuestras consultas, para ello abrimos como administradores de

la base de datos, la consola de administración de MongDB, Mongo Atlas, que nos proporciona herramientas de monitorización para observar comportamientos anómalos que pudieran generar nuestras consultas.


Figura 25. Métricas de operaciones sobre la base de datos principal.

Como las consultas pueden diseñarse de muchas formas, para aumentar el rendimiento y la optimización de ellas seguiremos dos criterios fundamentales:

- Tiempo de respuesta: ante consultas con diferentes soluciones, agregaciones o consultas simples. Se probarán todas y usaremos la más eficiente.
- Uso de índices: MongoDB permite crear índices para campos, o combinaciones de ellos, que se usen en consultas recurrentes. Como indica la documentación de Mongodb, "Los índices apoyan la ejecución eficiente de consultas. Sin ellos, la base de datos debe escanear cada documento de una colección o tabla para seleccionar aquellos que coincidan con la declaración de la consulta" [2] Hay que tener en cuenta que el uso de consultas configuradas con distintos campos, implica el uso de índices compuestos. Siempre que lo permita y sea posible se creará un índice para la consulta diseñada y si no es posible, se tratarán de reducir los campos usados, permaneciendo el que use un índice y aplicaremos el criterio de tiempo de respuesta para elegir la mejor consulta.

Configuramos las instrucciones y procedemos a validar el código completo. Como hemos mencionado hay variables configuradas en el entorno y para simularlo en local usaremos la librería DotEnv [23], la cual configura en el *process.env* las variables que encuentre en un archivo *.env* donde hemos añadido los valores. Se aplica esta librería en los archivos de configuración. Usamos ya variables de producción, pero enviamos el reporte a nuestro correo corporativo. Procedemos a ejecutar el código con el comando principal que inicia el Job.

Se corrigen los posibles errores encontrados y se perfecciona la tabla HTML del reporte hasta obtener el resultado deseado.

También añadimos en el archivo *README* los pasos para ejecutar el código en local y cómo deben configurarse los modelos de consulta y agregación para la adición de nuevas instrucciones. Por lo diseñado, esto no será más que añadir en el array de cada uno de los modelos un nuevo objeto con la información necesaria según la inconsistencia que se quiera vigilar.

Como seguimiento del estándar para buenas prácticas de nuestro departamento, integramos en nuestro repositorio, EsLint. Una herramienta que analiza el código de manera dinámica para identificar y modificar automáticamente problemas en el código. En este caso configuramos el archivo:

*.eslintrc.js*

Añadimos reglas ya estipuladas por el equipo de desarrollo, todas ellas orientadas a mantener el código limpio y las convenciones establecidas. Para aplicarlo en todo el código del repositorio, bastará con correr el siguiente comando:

```
npm run lint-fix
```

Una vez aceptado el código del proyecto por el equipo, se añade al repositorio por primera vez como rama máster y definiendo la versión 1.0.0 de este CronJob.

# 5 DESPLIEGUE

"El despliegue continuo no es solo una mejora técnica, sino también una mejora en la calidad de vida de los desarrolladores."

*Gene Kim*

Este proyecto nace como iniciativa del departamento de desarrollo para la integración de nuevas instrucciones en la detección de inconsistencias, una recodificación es necesaria. Para facilitar el despliegue de los cambios, se automatizan las tareas que implican su puesta en marcha.

En este apartado vamos a describir las técnicas usadas para la configuración de la imagen Docker, la creación del contenedor con la imagen construida y su configuración, con la colaboración del equipo de infraestructura. Los archivos necesarios para automatizar el despliegue serán:

- Dockerfile : archivo con las instrucciones y configuración necesarias para la construcción de la imagen Docker.
- Manifest.yml : archivo de configuración para el clúster de Kubernetes. Incluye la configuración de los recursos del CronJob.
- Bitbucket-pipelines.yml : archivo de automatización de tareas cuando el código se sube al repositorio.

Una vez se ha implementado el código, se hace un commit a la rama de desarrollo con los cambios y se sincroniza el código a Bitbucket. Con la rama se crea una PR, pull request, hacia la rama donde queremos fusionar. Esta nos permite solicitar la revisión del código por el líder técnico.

Una vez aprobados los cambios, se procede a la fusión de las ramas desde Bitbucket, esta función provoca la ejecución de un pipeline, configurado en el bitbcuket-pipeline.yml. Esta configuración contiene los pasos necesarios para crear una nueva imagen Docker con los cambios y cargarla en el repositorio AWS ECR (AWS Amazon Elastic Container Registry).

Con la nueva imagen, el equipo de infraestructura despliega el nuevo contenedor aplicando la configuración del manifest.yml que habremos subido previamente en un repositorio de configuración aparte. Esto provoca que el CronJob quede finalmente desplegado en el clúster de Kubernetes.

Una representación del proceso explicado se encuentra en la siguiente figura:

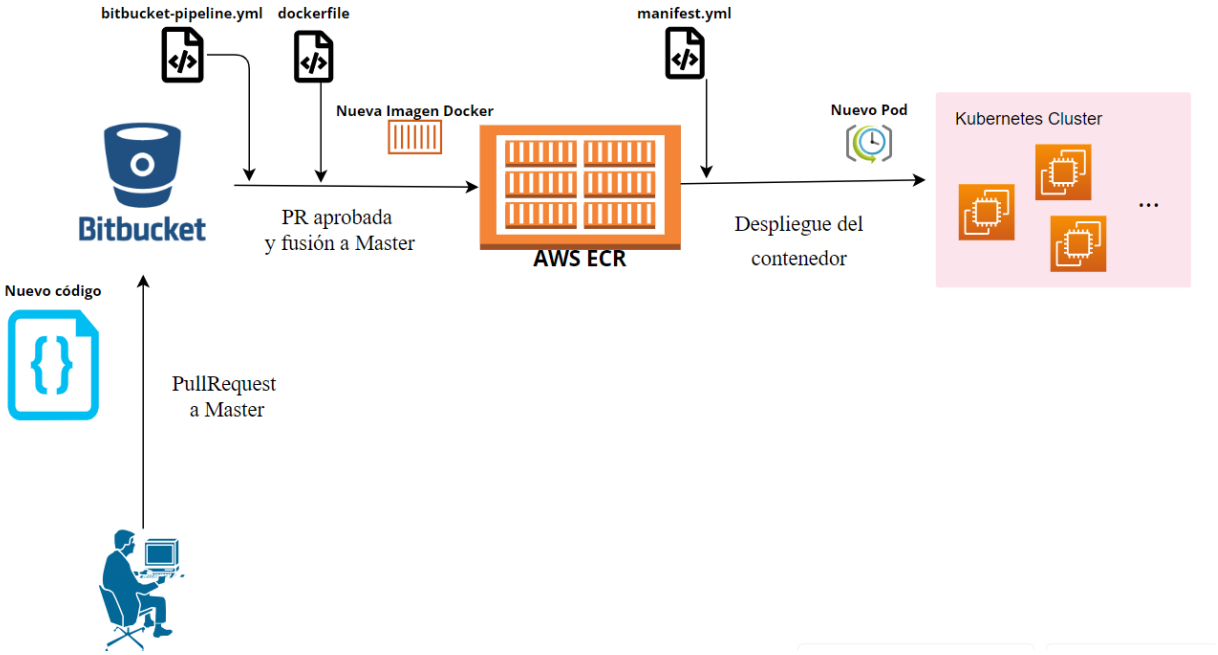



Figura 26. Proceso de despliegue

## 5.1 Definición de la imagen

Para crear la imagen de un contenedor para una aplicación es necesario definir el *Dockerfile* [14]

El formato del documento es el siguiente:

```
# Comentario
INSTRUCCION argumentos
```

La instrucción no distingue entre mayúsculas o minúsculas, por convención se aplicarán siempre mayúsculas.

Todas las instrucciones se leen en orden pero siempre debe empezar por la instrucción `FROM`, que se explica más adelante, y podría ser precedido por directivas del analizador, comentarios e instrucciones `ARG`.

Las directivas del analizador son opcionales y afectan a cómo se interpretan las líneas. No son instrucciones de la construcción de la imagen, deben estar al principio del archivo y son escritas en formas de comentario: `# directive=value`. Encontramos dos:

- Syntax: declara que versión del intérprete del Dockerfile que se tiene que usar. Si no se especifica por defecto se usa una versión empaquetada de la interfaz.
- Escape: se usa para configurar el carácter de escape, por defecto `/`. Escapa tanto caracteres como nuevas líneas.

Se pueden usar variables de entorno en las instrucciones siguiendo esta notación:

`$nombre_variable` o `${nombre_variable}`

También se permite el uso de modificadores para definir valores por defecto según la existencia de la variable o reemplazos en su valor siguiendo un patrón(para una versión del interpretador del Dockerfile concreta.)

Por ejemplo:

```
${variable:-pordefecto}
```

Usa el valor de la variable si existe y sino usa *pordefecto*.

El uso de variables esta soportado por la mayoría de instrucciones pero hay que tener en cuenta algunos detalles para RUN, CMD, and ENTRYPOINT. Estos soportan dos tipos de formatos:

- Tipo exec:

  ```
  INSTRUCTION ["executable","param1","param2"]
  ```

  El comando se especifica como un array, con el primer elemento siendo el ejecutable y el resto cualquier parámetro. La ejecución del comando no invoca un Shell por lo tanto es mas eficiente y evita problema de interpretación de caracteres.

- Tipo Shell:

  ```
  INSTRUCTION command param1 param2
  ```

  El comando se especifica como una cadena de texto. Docker lo ejecuta dentro de un Shell y es útil cuando se requiere usar alguna característica de los Shell.

Algunas de las instrucciones más relevantes:

- FROM: establece la imagen base desde la que se parte para la construcción del contenedor.
- RUN: ejecuta un comando y crea una nueva capa sobre la imagen actual.
- CMD: define el comando por defecto que se usa cuando se corre el contenedor de la imagen actual. Solo puede haber una.
- LABEL: añade metadatos a la imagen en forma de etiqueta. Son pares clave-valor.
- WORKDIR: establece el directorio de trabajo para cualquiera de las siguientes instrucciones que lo use.
- ARG: define variables solo están disponibles en tiempo de construcción del contenedor.
- COPY: copia archivos o directorios desde un directorio fuente a un destino específico.
- ENV: estable variables de entorno. Se definen como un par clave-valor.
- ENTRYPOINT: configura un comando que se ejecutará cuando se inicie el contenedor.
- EXPOSE: especifica los puertos abierto en los que escucha el contenedor en tiempo de ejecución.

Por último, se puede añadir un fichero *.dockerignore* donde añadir archivos o directorios de nuestro directorio de trabajo que serán excluidos en el contexto de construcción.

Nuestro archivo Dockerfile ha sido configurado de la siguiente manera:
Partimos de la imagen base, en este caso usamos node:20-alpine, basado en el proyecto: Alpine Linux Project y que incluye la versión 20 de NodeJS. Es una distribución mucho más optimizada y ligera para pequeñas imágenes.

```
FROM node:20-alpine
```

Definimos el directorio de trabajo donde se encontrarán los archivos de nuestro proyecto. Copiamos los archivos de configuración del código y comenzamos la instalación de dependencias con el uso de NPM. A partir de eso copiamos todos los archivos del código restantes al contenedor.

```
WORKDIR /usr/src/app
COPY package.json package-lock.json ./
RUN npm install
COPY . .
```

Hay variables de entorno para el acceso a AWS que se guardan en secretos y el equipo de infraestructura usa una solución personalizada basada en el script *entrypoint.sh*. No se entrará en ningún detalle de esta solución, pero su script es necesario añadirlo en el directorio de la aplicación. Configuramos los permisos de este script:

```
RUN chmod +x /usr/src/app/entrypoint.sh
```

Corremos una actualización de la distribución y configuramos variables de entorno para el manejo de los secretos en AWS.

```
RUN apk update && apk add --no-cache aws-cli jq
ARG SECRET_ARN
ENV SECRET_ARN=${SECRET_ARN}
```

Finalmente configuramos el script para que se aplique al correr el contenedor.

```
ENTRYPOINT ["/usr/src/app/entrypoint.sh"]
```

Finalmente, configuramos el comando para levantar el servicio:

```
CMD [ "npm", "start" ]
```

Donde se lanzará el servicio con el comando final a su vez definido en el archivo *package.json*. Adicionalmente añadimos al archivo *.dockerignore* el directorio *node_modules* con las dependencias del proyecto. Con el fin de mantenerlo limpio y actualizado en versiones posteriores, se crean las dependencias junto al Docker.

Comprobamos que el contenedor se crea usando el comando:

```
Docker build
```

La imagen creada ya contiene todo lo necesario para correr el servicio diseñado, puede ser añadida al AWS ECR y configurada en el clúster como un CronJob manualmente, explicaremos en el siguiente punto como con el uso de YAML hemos automatizado estas tareas.

## 5.2 Manifiesto de Kubernetes

Como ya hemos mencionado en el diseño de la solución, el CronJob se va a configurar en el clúster de la aplicación. Este clúster desplegado en AWS se administra usando el software de código abierto Kubernetes. Como explica la documentación de AWS sobre Kubernetes [15]:

> Kubernetes administra un clúster de instancias de informática y programa contenedores para que se ejecuten en el clúster en función de los recursos informáticos disponibles y de los requisitos de recursos de cada contenedor. Los contenedores se ejecutan en agrupaciones lógicas llamadas pods y es posible ejecutar y ajustar la escala de uno o más contenedores juntos como un pod. [15]

El clúster de Kubernetes se organiza en nodos que corren los contenedores de la aplicación. Estos nodos los componen los Pods, los componentes de la carga de trabajo de la aplicación. Te proporciona una plano de control que dispone de una API para las tareas de administración de los Nodos y los Pods del clúster.

Entre las principales características de Kubernetes [16] destaca:

- La distribución de la carga según el tráfico de un contenedor.
- Gestión y configuración de secretos para actualizar la configuración de aplicaciones sin tener que volver a construir la imagen.
- Escalado horizontal, ya sea de manera manual o automatizado.
- Reinicio automático de contenedores ante cualquier fallo.

Ya hemos creado la configuración de la imagen del contenedor, ahora lo usaremos para crear el Pod del contenedor en el clúster y establecerlo como un CronJob creando el manifiesto de configuración. Kubernetes nos permite desplegar contenedores con el uso de archivos YAML que dictan sus características.

Partiendo de la plantilla que ofrece la documentación de Kubernetes para la automatización de tareas como CronJob:

application/job/cronjob.yaml

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox:1.28
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

Figura 27. Plantilla para el manifiesto CronJob

Mantenemos la versión de la API de Kubernetes que se usa y el tipo de contenedor:

```
apiVersion: batch/v1
kind: CronJob
```

Configuramos el nombre del contenedor y una etiqueta para organizar Pods de su misma naturaleza:

```
metadata:
  name: alerts-database-incongruities
  namespace: scripts
```

La siguiente configuración es más específica del comportamiento del CronJob, usamos Schedule para definir la repetición de este, la regla es construida y comprobada fácilmente con una web externa llamada *crontab guru* (https://crontab.guru/):

```
spec:
  schedule: "00 03 * * 01"
```

Esto significa que el Job se lanzará a las 3 de la mañana todos los lunes.

Definimos la plantilla del *job* sobre *containers* con el nombre del contenedor y la imagen que se deberá usar en el contenedor. Esta será siempre la última, dado que se especifica la etiqueta *latest.* Añadimos los argumentos que inician el CronJob, *args.*

```
    jobTemplate:
      spec:
        template:
          spec:
            containers:
            - name: alerts-database-incongruities
              Image :     XXXX.XX.amazonaws.com/database-incongruities-
job:latest
              args:
                - "node"
                - "index.js"
```

Ahora una parte importante es la configuración del entorno, como mencionamos anteriormente, algunas de las variables que usamos son añadidas como variables de entorno. La principal razón, es que estas contienen información sensible que no queremos añadir en el código y se encuentran en algún contenedor de secretos. Otra es que la información podría cambiar en el futuro sin afectar a la lógica del código, por lo que es más fácil añadirlo en la configuración del CronJob y aplicarla sin recrear la imagen. Añadimos así en el apartado *env:*

```
              env:
                - name: MAILGUN_KEY
                  valueFrom:
                    secretKeyRef:
                      key: API_KEY_MAILGUN
                      name: xxxxxxxxxx
                - name: MONGO_URI
                  valueFrom:
                    secretKeyRef:
                      key: MONGO_URI
                      name: xxxxxxxxxxx
                - name: DATABASE
                  value: xxxx
                - name: EMAIL_TO
                  value: xxx@podgroup.com
                - name: EMAIL_FROM
                  value: xxxx@podgroup.com
                - name: DOMAIN
                  value: podgroup.com
```

Por último, configuramos dos campos, uno para que el cron se actualice siempre con la última imagen agregada en el ECR que tenga asociado y otra para que este se reinicie en caso de fallo:

```
              imagePullPolicy: Always
            restartPolicy: OnFailure
```

Una vez ya configurado el manifiesto el equipo de infraestructura lo aplica en el clúster con el uso de Kubctl, la herramienta de línea de comandos para Kubernetes.

## 5.3 Pipeline de Bitbucket

Falta automatizar nuevas subidas de código al repositorio, la correspondiente creación de la nueva imagen del contenedor y su subida en el AWS ECR, para ello se sigue la misma lógica interna ya usada en otros entornos, configuramos una pipeline para nuestro repositorio, archivo bitbucket-pipelines.yml.

Creamos el archivo con el asistente de Bitbucket para la creación de pipelines:

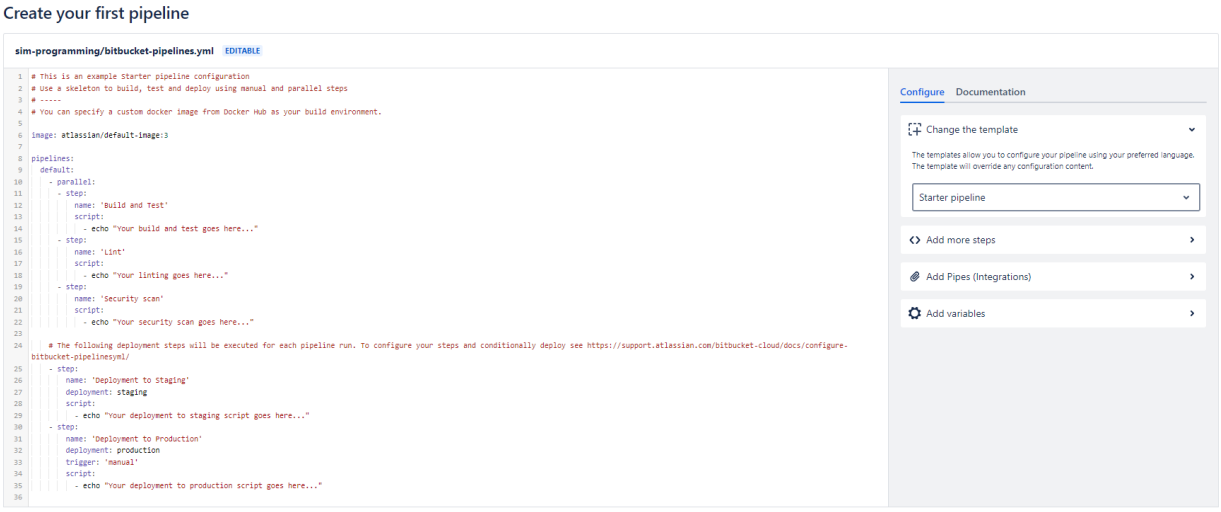



Figura 28. Asistente de Bitbucket para pipelines

Para la creación de la pipeline hay que conocer las propiedades que admite Bitbucket [17] para el archivo YAML.

La propiedad padre es *pipelines* y engloba la configuración de los pasos de la pipeline. Para los pasos tenemos que configurar dos características:

- Condición de comienzo: permite configurar cuando se lanza la pipeline. Puedes definirlo en el momento de fusión de ramas, creación de PR y manual entre otras. Nosotros la lanzaremos cuando se fusione sobre la rama Master, por tanto usaremos la propiedad *branches* y *master*.
- Pasos a seguir: se definen usando la propiedad obligatoria *step* y que define una unidad de ejecución de compilación, son 100 como máximo y engloban la propiedad obligatoria *script*. Durante cada paso se lanza un contenedor Docker que ejecuta los comandos configurados en *script*. Alguna de las propiedades más importantes que se pueden incluir son:
  - *Name*: nombre del paso.
  - *Runtime*: modificar entorno de ejecución para el paso.
  - *Caches*: define una caché personalizada o predefinida para evitar descargar dependencias de la construcción.
  - *Artifacts*: indica archivos o directorios que contengan paquetes necesarios en la construcción.
  - *Image*: configura la imagen Docker de la pipeline.
  - *Conditions*: para el control del flujo, añade condiciones que previene ejecutar un paso si no se satisfacen.

  Varios pasos se pueden organizar con la propiedad *stage.*

Según nuestra configuración cuando se realice una fusión a la rama Master se ejecutará el paso “Create image for Prod and push to ECR”:

```
pipelines:
  branches:
    master:
      - step:
          name: Create image for Prod and push to ECR
```

Para este paso se usa una imagen con la interfaz por línea de comando para AWS configurada y usaremos una caché definida para Docker.

```
          caches:
            - docker
          image: atlassian/pipelines-awscli
```

Describimos rápidamente la sección *script* donde está la lógica del pipeline:

- Creación de un script para hacer login en la plataforma de AWS.

```
script:
  - echo $(aws ecr get-login --no-include-email --region eu-west-1)  > login.sh
```

- Ejecución del script.

```
  - sh login.sh
```

- Construcción de la imagen del contenedor con el nombre formado por el *path* al repositorio ECR y la variable imagen del Docker. Usamos el commit de Bitbucket como tag.

```
  - docker build -t ${DOCKER_ECR_REPO_URL}/${DOCKER_IMAGE_NAME}:${BITBUCKET_COMMIT} .
```

- Creamos el tag *latest* para la imagen creada.

```
  - docker tag ${DOCKER_ECR_REPO_URL}/${DOCKER_IMAGE_NAME}:${BITBUCKET_COMMIT}
${DOCKER_ECR_REPO_URL}/${DOCKER_IMAGE_NAME}:latest
```

- Subimos al ECR la imagen con la etiqueta del *commit* y también lo hacemos con la etiqueta *latest*:

```
  - docker push ${DOCKER_ECR_REPO_URL}/${DOCKER_IMAGE_NAME}:${BITBUCKET_COMMIT}
  - docker push ${DOCKER_ECR_REPO_URL}/${DOCKER_IMAGE_NAME}:latest
```

Por último, se pueden añadir opciones globales con el uso de la propiedad *options*, nosotros solo configuraremos *docker* como verdadero para usar los servicios de Docker en todos los pasos de la pipeline:

```
options:
  docker: true
```

Con este archivo completamos la automatización de la subida de nuevas imágenes al AWS ECR desde Bitbucket.

# 6 CONCLUSIONES

*Lo más importante en cualquier base de datos es la integridad, no el rendimiento. La velocidad es importante, pero nunca a costa de la precisión.*

*-Donald Knuth-*

La solución se ha intentado mantener sencilla desde el punto de vista del código, definiendo y simplificando la modificación de los modelos, lo que facilita la integración de futuras alertas o comprobaciones, que se quieran hacer de la base de datos. Se ha intentado mantener variables de configuración que permitan, partiendo de la misma lógica del código, integrar la detección de inconsistencias en distintas bases de datos, siempre basadas en la creación de las consultas y agregaciones.

Integrado en el proceso de trabajo del departamento de desarrollo, el reporte creado se revisa cada lunes para confirmar que el recuento de inconsistencias es 0 para todas las instrucciones desarrolladas. Si no lo fuera, se revisa la instrucción y se investiga la consulta contra la base de datos.

Se reduce considerablemente el soporte respecto a problemas en la base de datos o reportes de inconsistencias en los datos de la plataforma. Ahora nos adelantamos a muchos de los problemas que podría presentar el crecimiento de nuestra base de datos y que no han sido tomados en cuenta a la hora de integrar nuevas características de la aplicación.

Desde el punto de vista técnico, también cabe destacar que la integración del script como un contenedor que se configura de manera cronológica para ser ejecutado cada cierto tiempo resulta interesante como patrón paro automatizar pequeñas tareas que puedan resultar tediosas y que no pueden ser integradas en la aplicación.

# REFERENCIAS


[1] Giesecke+Devrient DmbH. (2024), Digital Security, Connectivity IoT. https://www.gi-de.com/

[2] MongoDB, Inc. (2024), ¿Qué es MongoDB? . MongoDB. https://www.mongodb.com/

[3] Fundación Mozilla. (2024), JavaScript. MDN dev docs. https://developer.mozilla.org/

[4] OpenJs Foundation (2024), Introduction to Node.js. NodeJs. https://nodejs.org/

[5] Docker Inc. (2024), Use containers to Build, Share and Run your applications. Docker. https://www.docker.com/

[6] Ingy et al. (2021), YAML Ain't Markup Language (YAML™) version 1.2. Ingy et al. https://yaml.org/spec/1.2.2/

[7] MongoDB Admin GUI. (2024), The Smartest IDE for MongoDB. NoSQLBooster. https://nosqlbooster.com

[8] Microsoft. (2024), Getting Started. Visual Studio Code. https://code.visualstudio.com/

[9] Atlassian. (2024), A brief overview of Bitbucket. Bitbucket. https://bitbucket.org/

[10] Mailgun. (2024), ABOUT MAILGUN. Mailgun. https://www.mailgun.com/

[11] MongoDB, Inc. (2024), MongoDB Node Driver. MongoDB. https://www.mongodb.com/docs/drivers/node/v6.4/

[12] NPM, Inc. (2024), Axios. Npm Docs. https://www.npmjs.com/package/axios

[13] NPM, Inc. (2024), Axios. Npm Docs. https://www.npmjs.com/package/form-data

[14] Docker Inc. (2024), Dockerfile reference. Docker. https://docs.docker.com/reference/dockerfile/

[15] Amazon Web Services, Inc, (2023), Kubernetes en AWS. AWS. https://aws.amazon.com/es/kubernetes/

[16] The Kubernetes Authors, (2024), Documentation. Kubernetes. https://kubernetes.io/docs/home/

[17] Atlassian. (2024), Bitbucket Pipelines configuration reference. Bitbucket. https://support.atlassian.com/bitbucketcloud/docs/bitbucket-pipelines-configuration-reference/

[18] Google. (2024), Overview. Angular. https://angular.dev/overview

[19] Restify team. (2024), Meet restify. Restify. http://restify.com/

[20] IoT analytics GmbH. (2024), Number connected IoT devices. IoT Analytics. https://iot-analytics.com/

[21] NPM, INC. (2024), NPM. NPM. https://www.npmjs.com/

[22] Mongoose team. (2024), mongoose. Mongoose. https://mongoosejs.com/

[23] NPM, Inc. (2024), DotEnv. Npm Docs. https://www.npmjs.com/package/dotenv

# GLOSARIO

| | |
|---|---|
| *G+D* | Giesecke+Devrient |
| IoT | Internet of Things |
| SIM | Subscriber Identity Module |
| eUICC | Embedded Universal Integrated Circuit Card |
| eSIM | eUICC SIM |
| UI | User Interface |
| API | Application programming interface |
| REST | Representational state transfer |
| FE | Front End |
| BE | Back End |
| JS | JavaScript |
| NPM | Node Package Manager |
| ODM | Object Data Modeling |
| PR | Pull Request |
| BBDD | Base de datos |
| IDE | Integrated development environment |
| AWS | Amazon web Services |
| ECR | AWS Amazon Elastic Container Registry |