Documentación de Análisis y Diseño de la Arquitectura

Documentación de Análisis y Diseño de la Arquitectura
PMPV

- Introducción
  - Propósito
  - Alcance
- Contexto del Proyecto
  - Descripción del Proyecto
  - Stakeholders
- Descripción de la Arquitectura
  - Tipo de arquitectura
  - Metodología
  - Requisitos y características de la arquitectura
    - Requisitos
    - Características
  - Vistas de la arquitectura
    - Vista de Contexto y Lógica.
    - Vista de Despliegue.
- Decisiones de Diseño
  - Principios de diseño
  - Estilos arquitectónicos
  - Tecnologías, plataformas y protocolos de comunicación.
- Validación de la Arquitectura
  - Estrategia de Validación
  - Resultado de Validación

Introducción

Propósito

El objetivo de esta documentación es describir el análisis y diseño de la arquitectura del sistema Planificación Mantenimiento Preventivo Vehicular (PMPV).

Alcance

La documentación aborda el análisis y diseño de la arquitectura.

Contexto del Proyecto

Descripción del Proyecto

El sistema de planificación de mantenimiento preventivo vehicular tiene como objetivo principal gestionar el mantenimiento programado de una flota de vehículos. La aplicación facilitará la gestión de las tareas de mantenimiento proporcionadas por el fabricante como de aquellas generadas por el propio operador. Además, se gestionarán las revisiones de las tareas, notificando cualquier cambio a los ingenieros responsables de la gestión y planificación del mantenimiento de vehículos.

Los ingenieros crearán planes de mantenimiento para vehículos similares que al aplicarlos generarán, automáticamente, órdenes de trabajo. Estas, serán gestionadas por los mecánicos, los cuales también podrán crear nuevos trabajos debido a la detección de errores durante la realización de sus trabajos. Además, la aplicación gestionará los diferentes tipos de usuario (ingenieros, mecánicos, administradores, etc..) que podrán acceder de forma segura a las diferentes funcionalidades de esta.

Cabe destacar que el plan de mantenimiento preventivo contribuye a prevenir fallos y garantizar el correcto funcionamiento de los vehículos. De hecho, la aplicación se deberá diseñar para que gestione un crecimiento y sea escalable y esté en la nube, con procesos automatizados para minimizar errores y garantizar su calidad.

Stakeholders

Administradores

Descripción de la Arquitectura

Tipo de arquitectura

Microservicios

Metodología

Metodología ágil

Requisitos y características de la arquitectura

Requisitos

- PMPV-106: US-2023-Como arquitecto quiero identificar medidas de seguridad FINALIZADA
- PMPV-103: US-2023-Como arquitecto quiero evaluar y seleccionar las tecnologías y las herramientas que se utilizarán FINALIZADA
- PMPV-27: US-2023-Como arquitecto quiero definir la estructura de la aplicación FINALIZADA
- PMPV-105: US-2023-Como arquitecto quiero definir los flujos de datos y comunicación FINALIZADA
- PMPV-104: US-2023-Como arquitecto quiero establecer patrones de diseño FINALIZADA
- PMPV-107: US-2023-Como arquitecto quiero definir la estrategia de integración continua y despliegue continuo FINALIZADA
- PMPV-109: US-2023-Como arquitecto quiero definir la estrategia de automatización de pruebas FINALIZADA

Características

Características	Detalle
Seguridad	El sistema debe ser seguro y solo permitir acceso a usuarios autorizados.
Escalabilidad	El sistema debe ser escalable para solventar problemas de rendimiento en el futuro.
Tolerancia a fallos	El sistema debe ser tolerante a fallos para garantizar la disponibilidad y fiabilidad.
Separación por funcionalidades	El sistema debe separarse por funcionalidades para garantizar la escalabilidad, testeabilidad y mantenibilidad dada la complejidad de la aplicación.
Interoperabilidad	El sistema debe integrarse con otros sistemas externos.
	El sistema debe incluir la capacidad de realizar notificaciones.

Características de la arquitectura

Vistas de la arquitectura

Vista de Contexto y Lógica.

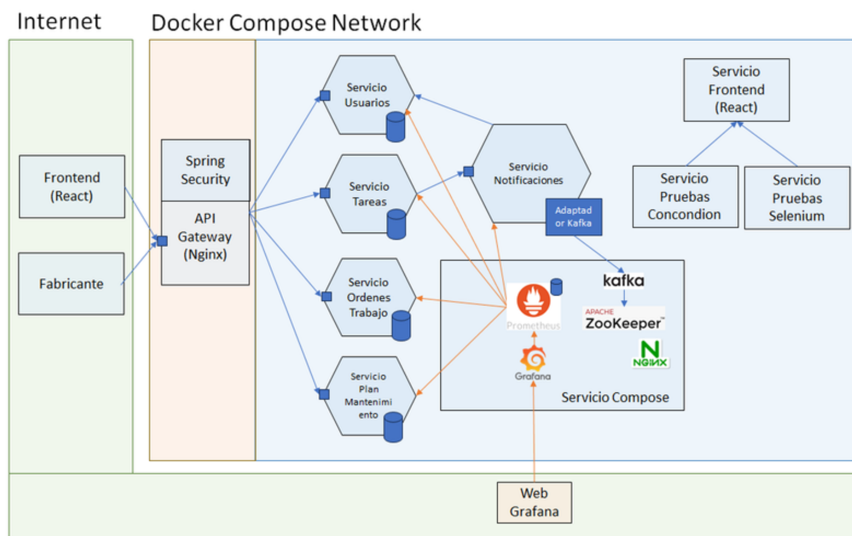
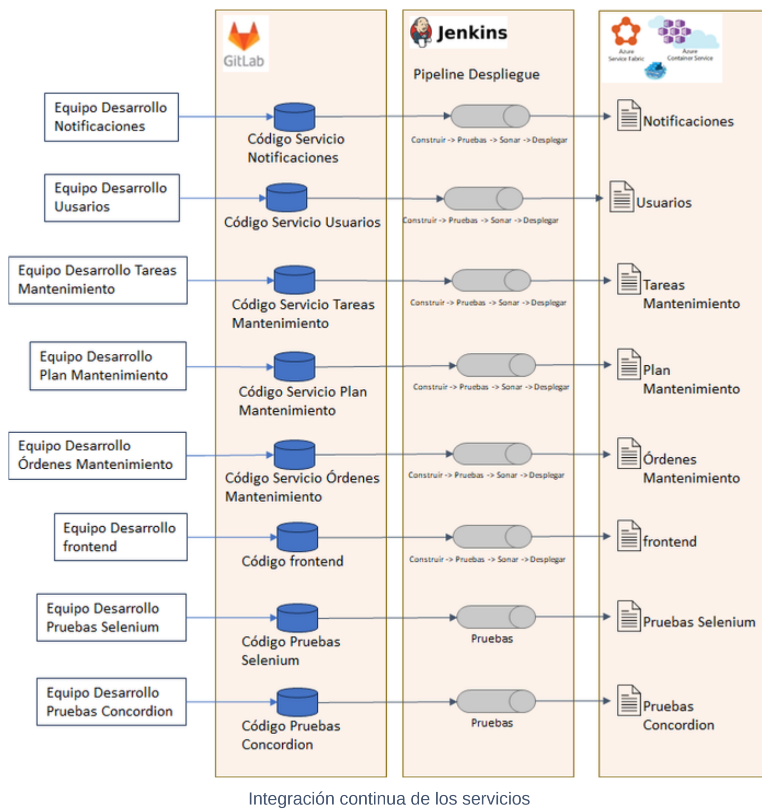


Diagrama de componentes de la aplicación

Vista de Despliegue.



Integración continua de los servicios

Decisiones de Diseño

Principios de diseño

A continuación, se establecerán los principios y pautas que posibilitarán la creación y organización de la arquitectura teniendo en cuenta las características que se han extraído de los requisitos.

Arquitectura

- Se debe mantener una documentación actualizada sobre la arquitectura.
- Es preferible realizar una división por funcionalidades.
- Es preferible la escalabilidad horizontal ya que facilita el crecimiento.
- Es preferible utilizar mensajes asíncronos entre los servicios ya que aumenta la tolerancia a fallos.

Interfaz de Usuario

Es preferible que la interfaz de usuario sea intuitiva y reactiva.

Seguridad

- Es preferible que la comunicación entre los diversos servicios se realice a través de una *API Rest*, con interfaces y contratos claros y estandarizados.
- Es preferible incorporar medidas de seguridad que permitan gestionar los accesos a la aplicación.

Integración Continua y Despliegue Continuo (CI/CD)

- Es preferible que las diferentes funcionalidades de la aplicación implementen una funcionalidad específica, sean autónomas y tengan su propio proceso de integración continua (CI/CD).
- Es preferible que el proceso de integración continua (CI/CD) incluya las siguientes fases: compilación, pruebas, análisis de vulnerabilidades, calidad de código y despliegue.

Monitorización

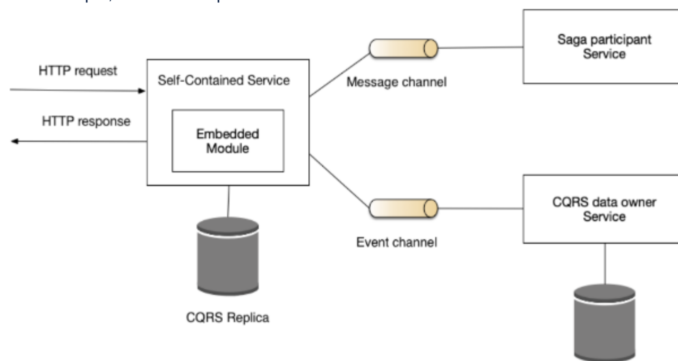
Es preferible monitorizar cada funcionalidad, centralizar la recopilación de datos y visualizarlos en tiempo real para su evaluación.

Automatización de Pruebas

Es preferible realizar la automatización de las siguientes pruebas: unitarias, criterios de aceptación, regresión y reglas de gobernabilidad de la arquitectura[1].

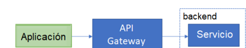
Patrones de diseño

- Arquitectura:
  - Patrón arquitectura de microservicios, permite diseñar una aplicación como una colección de servicios centrados en los dominios del producto proporcionando claridad y calidad en el código, facilita el escalado, servicios independientes los unos de los otros y disponibles para ser comunicados en cualquier momento.
- Descomposición:
  - Patrón descomponer por subdominio, permite definir los servicios correspondientes a subdominios DDD. Este enfoque proporciona servicios cohesivos y débilmente acoplados, y equipos de desarrollo multifuncionales, autónomos y organizados en torno a la entrega del valor del producto.
  - Patrón servicio autónomo, permite, en colaboración con el patrón CQRS y Saga, responder a una solicitud sincrónica sin esperar la respuesta de ningún otro servicio. Este enfoque, aborda la disponibilidad de los servicios.



Solicitud síncrona sin espera

- Gestión de datos:
  - Patrón base de datos por servicio. Cada servicio tiene su propia base de datos privada, lo que permite que estén débilmente acoplados y utilicen la BBDD que mejor se adapte a sus necesidades. En este enfoque, será necesario aplicar los patrones CQRS [2] y Saga para implementar transacciones y consultas que abarquen servicios.
  - Patrón CQRS, separa la parte de comandos de la de consultas. Este enfoque simplifica los modelos de consulta y comando.
  - Patrón Saga, permite mantener la consistencia de datos a lo largo de los servicios.
- Pruebas:
  - Patrón prueba de contrato impulsada por el consumidor, permite verificar a un servicio que las llamadas a otro servicio cumplen con sus expectativas.
  - Patrón prueba de componente de servicio, permite verificar un servicio de forma aislada.
- Despliegue:
  - Patrón instancia de servicio por contenedor, empaqueta y ejecuta el servicio usando contenedores *Docker* independientes.
- API externo:
  - Patrón *API Gateway*, permite ofrecer una pasarela que sirve de punto único de acceso a la aplicación encapsulando toda la arquitectura interna, de modo que no se expongan los datos y operaciones.



API Gateway

- Seguridad:
  - Patrón token de acceso, permite comunicar la identidad del solicitante a los servicios que gestionan la solicitud. Este enfoque garantiza la seguridad y autenticación de las solicitudes, es decir, *API Gateway* autentica la solicitud y pasa un *token* de accesos que identifica de forma segura.

- Comunicación:
  - Patrón mensajería, permitirá enviar mensajes a otros servicios de forma asíncrona y para garantizar la consistencia se utilizará con el patrón bandeja de salida transaccional. Este enfoque permite la colaboración entre los servicios de forma asíncrona. Por ejemplo, *Kafka*.
  - Patrón bandeja de salida transaccional, permite garantizar la consistencia de datos ya que el mensaje solo se envía si la transacción se ha completado correctamente.
- Observabilidad[3]:
  - Patrón agregación de registros, permite centralizar la información registrada en un servicio.
  - Patrón API de comprobación de estado, permite detectar el mal funcionamiento de una solicitud.
  - Patrón métricas de aplicación, permite recopilar métricas específicas de una aplicación.

Estilos arquitectónicos

Esta última dimensión permitirá obtener una visión de la organización y disposición de los componentes en el sistema, incluyendo el estilo arquitectónico, la organización de los módulos y las conexiones entre ellos.

En primer lugar, el sistema seguirá un estilo arquitectónico basado en microservicios y cada servicio adoptará un estilo Hexagonal[4]. Esto permitirá una clara separación de responsabilidades. Además, se utilizará *Docker* para contenerizar cada servicio, de modo que su mantenimiento y despliegue sean más sencillos.

Cada servicio representará un subdominio de la aplicación y tendrá su propia base de datos, sistema de monitorización y una integración continua con pruebas propias. Además, proporcionará acceso a sus funcionalidades a través de *APIs*, y mediante *Docker Compose*, facilitará la comunicación entre los servicios a través de una red compartida entre ellos.

Para gestionar el acceso a las funcionalidades de los servicios, se implementará una fachada que los expondrá a través de una *API Gateway* con autenticación. Esto proporcionará un punto de entrada único y asegurará que los servicios estén aislados del exterior.

Finalmente, cabe destacar que la infraestructura de producción estará alojada en Azure y se habilitará un acceso SSH para conexiones externas. Además, el despliegue se realizará a través de una integración continua con Jenkins que proporcionará una imagen de cada uno de los servicios.

Tecnologías, plataformas y protocolos de comunicación.

En esta dimensión, se expondrán las decisiones que se han tomado para implementar la arquitectura especificando tecnologías, plataformas y protocolos de comunicación.

Documentación

- Realizar la documentación en *Confluence* para mantener la trazabilidad con *Jira* y que todo esté centralizado en un repositorio.
- Los *API Rest* se documentarán en *Swagger*.

Estrategia de versionado

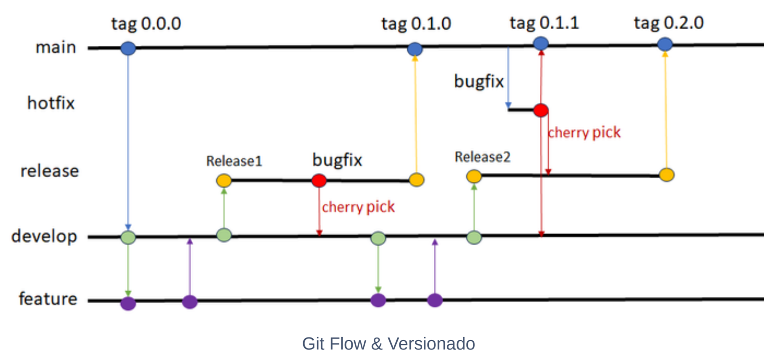
Adoptar el Versionado Semántico [19] para gestionar las versiones de la aplicación con la convención de número de versión MAYOR.MENOR.PARCHE que se incrementa:

- La versión MAYOR cuando realizas un cambio incompatible en el API.
- La versión MENOR cuando añades funcionalidad compatible con versiones anteriores.
- La versión PARCHE cuando reparas errores compatibles con versiones anteriores.

Esta estrategia, mejora la claridad de los cambios en las diferentes versiones.

Estrategia de ramificación (*branching*)

Aplicar el método *Git Flow* planteado por *Vincent Driessen en 2010* [20] para proporcionar una gestión de ramas clara y organizada que facilite el trabajo del equipo de desarrollo.



Git Flow & Versionado

Tecnologías backend

- *Spring Framework* y *Spring Boot*: Marcos de trabajo que facilitan el desarrollo con inyección de dependencias y aseguran el correcto funcionamiento de las solicitudes.
- *Postgress*: Base de datos que asegura la integridad de datos ACID[5] y admite consultas relacionales y no relacionales.
- *Apache Kafka*: Para mensajería asíncrona y procesamiento de eventos con tolerancia a fallos. Además, es perfecto para sistemas distribuidos.
- *Zookeeper*: Servido de código abierto que permite coordinar de manera fiable y consistente procesos distribuidos.

Tecnologías frontend

- *React*: Marco de trabajo que permite crear interfaces de usuario intuitivas y reactivas, fácil de aprender con conocimientos de *JavaScript*.
- *Bootstrap:* Librería de estilos basado en HTML y CSS que facilita el diseño *responsive*.

Infraestructura de despliegue

- *Jenkins*: Servidor enfocado en la automatización de procesos de integración continua, es de código abierto, permite utilizarlo en conjunto con *Docker* y se configura fácilmente.
- *SonarQube*: Herramienta de código abierto que garantiza la calidad del código analizando errores, vulnerabilidades y malas prácticas de programación.
- *Azure*: Proveedor en la nube en el que es fácil cargar imágenes *Docker*, automatizar el proceso con *Jenkins* y levantar sus instancias.
- *Docker*: Permite la contenerización de los servicios proporcionando servicios aislados y portátiles.
- *Kubernetes*: Permite orquestar y gestionar eficientemente las instancias de los contenedores y se integra fácilmente con *Azure*.

Monitorización

- *Micromiter*: Recopila métricas del servicio para evaluarlo.
- *Prometeus*: Herramienta de código abierto que almacena de forma centralizada las métricas.
- *Grafana*: Herramienta web de código abierto que permite configurar y visualizar las métricas.

Pruebas automáticas

- *Selenium:* Permite escribir y gestionar pruebas de aceptación automatizadas en proyectos basados en Java.
- *Concordion*: Herramienta de código abierto que se utiliza para probar aplicaciones web.
- *Junit5:* Marco de trabajo de pruebas unitarias para Java que permite verificar de forma automática la aplicación.
- *ArchUnit*: Librería que permite verificar la estructura del código.

Protocolos de comunicación

- *HTTP y API Rest*: para la comunicación entre los servicios.
- *API Gateway*: para ofrecer un único punto de acceso a la aplicación.

Seguridad:

Spring Security como marco de seguridad para la autorización y autenticación de la aplicación por su robustez, sencillez y flexibilidad en la protección de aplicaciones web y de servicios en *Spring*.

Por último, hay que indicar que, utilizando *Kafka*, *Docker* y *Kubernetes* se garantiza la escalabilidad. Además, *Kafka* y *Zookeeper* proporciona tolerancia a fallos.

Validación de la Arquitectura

Estrategia de Validación

Pruebas automáticas.

Resultado de Validación

Verificar que la estructura del código sigue la gobernabilidad de la arquitectura.

[1] Verificar que la implementación de código es consistente con los principios arquitectónicos definidos. Por ejemplo, si se han tenido en cuenta las buenas prácticas relativas a las dependencias entre los paquetes.

[2] Segregación de responsabilidades de consultas.

[3] Comprender y monitorizar el comportamiento interno de un sistema en tiempo real a través de la recopilación de datos.

[4] La arquitectura hexagonal organiza la lógica en el centro, rodeada de puertos y adaptadores, permitiendo una clara separación entre la lógica de negocio y los componentes externos.

[5] Atomicidad, consistencia, aislamiento y durabilidad.