Universidad Técnica Particular de Loja

NodoTech

Documento de Arquitectura de Software

Fecha: 02/02/2025

Aprobador(es) del Documento:

Nombre del Aprobador	Rol
Armando Cabrera	Docente tutor

Revisor(es) del Documento:

Nombre del Revisor	Rol
Carolina Alvarado	Desarrollador UXUUI
Kevin Barrazueta	Administrador de seguridad
Kevin Bustamante	Desarrollador Backend
Byron Castillo	Desarrollador Frontend
Freddy Leon	Lider de Proyecto, Inspector de calidad
Pablo Ramon	Arquitecto de Software

Resumen de cambios:

Versión	Fecha	Creado por	Descripción Breve de los Cambios
0.1	23.10.2024	Pablo Ramon	Creación del Mapa de Capacidades.
0.1	07.11.2024	Carolina Alvarado	Desarrollo del Prototipo inicial.
0.2	21.11.2024	Carolina Alvarado	Finalización del Prototipo final.
0.3	27.11.2024	Pablo Ramon	Diseño de modelos de datos y endpoints para el backend.
0.4	11.12.2024	Kevin Bustamante	Implementación del endpoint de autenticación.
0.5	20.12.2024	Kevin Bustamante	Creación de lógica de negocio para los flujos principales.
0.6	02.01.2025	Kevin Bustamante	Integración de servicios externos.
0.7	09.01.2025	Freddy Leon	Pruebas unitarias de funcionalidad.
0.7	09.01.2025	Kevin Barrazueta	Pruebas y validación de seguridad.
0.8	04.12.2024	Byron Castillo	Creación de componentes clave como formularios y dashboards en el frontend.
0.9	11.12.2024	Byron Castillo	Implementación de autenticación y manejo de sesiones en el frontend.
1.0	02.01.2025	Byron Castillo	Integración de APIs y validación de datos del backend en el frontend.
1.1	16.01.2025	Byron Castillo	Asegurar la responsividad y accesibilidad en el frontend.
1.2	20.01.2025	Byron Castillo/Freddy Leon	Realización de pruebas funcionales y visuales en el frontend.
1.3	27.01.2025	Equipo de Desarrollo	Despliegue final en Google Play Store.
1.4	02.02.2025	Equipo de Desarrollo	Pequeña actualización para corrección de bugs y mejoras menores.

Tabla de contenidos

Contenido

1. INTRODUCCIÓN

1.1. Propósito

Este documento proporciona una visión arquitectónica integral de la aplicación Checklist de Estado de Campus, diseñada para la Universidad Técnica Particular de Loja (UTPL). Utiliza múltiples vistas arquitectónicas para representar diferentes aspectos del sistema, capturando y transmitiendo las decisiones arquitectónicas significativas tomadas durante su desarrollo. El objetivo es asegurar la claridad, mantenibilidad y escalabilidad de la aplicación.

1.2. Alcance

La arquitectura descrita en este documento se refiere a la aplicación Checklist de Estado de Campus, que es una solución basada en la nube para gestionar y monitorear el estado de las instalaciones del campus. El sistema está diseñado para reemplazar los procesos manuales con una solución automatizada, eficiente y escalable. Este documento no cubre los detalles de los procesos internos de la UTPL ni los sistemas de gestión del campus en general, sino que se enfoca en la arquitectura técnica de la aplicación.

1.3. Referencias

#	Documento	Contenido
[REF1]	eDelivery AS4 Profile	El perfil AS4 de eDelivery es un perfil de los estándares OASIS ebMS3 y AS4. Tiene disposiciones para su uso en topologías de cuatro esquinas, pero también puede usarse en intercambios punto a punto.
[REF2]	OASIS AS4 Profile	Perfil AS4 de ebMS 3.0 Versión 1.0. Estándar OASIS, 23 de enero de 2013.
[REF3]	ebMS3 Core	Servicios de Mensajería ebXML de OASIS Versión 3.0: Parte 1, Características Principales. Estándar OASIS. 1 de octubre de 2007.
[REF4]	Domibus plugin cookbook	Manual técnico sobre el desarrollo de plugins de Domibus.
[REF5]	Apache CXF	Apache CXF es un marco de servicios de código abierto. Estos servicios pueden hablar una variedad de protocolos como SOAP, XML/HTTP, HTTP RESTful o CORBA y trabajar sobre una variedad de transportes como HTTP, JMS o JBI.
[REF6]	Apache WSS4J	El proyecto Apache WSS4J™ proporciona una implementación en Java de los estándares de seguridad principales para Servicios Web, a saber, las especificaciones de Seguridad de Servicios Web (WS-Security) de OASIS.
[REF7]	WS-Policy Specification	El marco de Servicios Web proporciona un modelo de propósito general y una sintaxis correspondiente para describir las políticas de las entidades en un sistema basado en servicios web.
[REF8]	Spring Security	Spring Security es un marco que se centra en proporcionar tanto autenticación como autorización a aplicaciones Java. Puede extenderse fácilmente para cumplir con requisitos

		personalizados.
[REF9]	Bcrypt	Provos, Niels; Mazières, David; Talan Jason Sutton 2012 (1999). "Un esquema de contraseñas adaptable al futuro". Actas de la Conferencia Técnica Anual USENIX de 1999: 81–92.
[REF10]	JMS	La API de Java Message Service (JMS) es una API de Java para Middleware Orientado a Mensajes para enviar mensajes entre dos o más clientes.
[REF11]	e-CODEX	El proyecto e-CODEX mejora el acceso transfronterizo de ciudadanos y empresas a medios legales en Europa y, además, crea la interoperabilidad entre autoridades legales dentro de la UE.
[REF12]	Java Servlet 3.0	Un servlet de Java es un programa Java que extiende las capacidades de un servidor. Aunque los servlets pueden responder a cualquier tipo de solicitudes, comúnmente implementan aplicaciones alojadas en servidores web. Estos servlets web son la contraparte en Java de otras tecnologías de contenido web dinámico como PHP y ASP.NET.
[REF13]	Xtext	Xtext es un marco para el desarrollo de lenguajes de programación y lenguajes específicos de dominio.
[REF14]	SOAP	Protocolo de Acceso a Objetos Simples
[REF15]	HTTP Chunking	Un mecanismo por el cual los datos se dividen en varios fragmentos cuando se envían a través de una conexión HTTP.
[REF16]	Documentación de Firebase	Firebase es una plataforma Backend-as-a-Service (BaaS) que proporciona bases de datos en tiempo real, autenticación y funciones en la nube.
[REF17]	Documentación de Docker	Docker es una plataforma para desarrollar, enviar y ejecutar aplicaciones en contenedores.
[REF18]	Documentación de Node.js	Node.js es un entorno de ejecución de JavaScript construido sobre el motor V8 de Chrome, utilizado para construir aplicaciones de red escalables.
[REF19]	Documentación de React Native	React Native es una biblioteca de JavaScript para construir interfaces de usuario, particularmente aplicaciones multiplataforma.
[REF20]	Documentación de Expo	Expo es un marco y plataforma para aplicaciones universales de React, que simplifica el desarrollo móvil.
[REF21]	Directrices para desarrolladores de Google Play	Directrices para publicar y gestionar aplicaciones en Google Play Store.
[REF22]	Documentación de MongoDB	MongoDB es una base de datos NoSQL orientada a documentos que almacena datos en formato JSON/BSON, ofreciendo escalabilidad, flexibilidad y consultas avanzadas.
[REF23]	Documentación de ClickUp	Plataforma de gestión de proyectos y tareas.
[REF24]	Documentación de Git	Sistema de control de versiones distribuido.
[REF25]	Documentación de Gradle	Herramienta de automatización de compilaciones.
[REF26]	Documentación de Jest	Framework de pruebas para JavaScript.
[REF27]	Documentación de JMeter	Herramienta para pruebas de carga y rendimiento.

[REF28]	Documentación de Google Lighthouse	Herramienta para mejorar el rendimiento web.
[REF29]	Documentación de Appium	Framework para pruebas automatizadas en móviles.
[REF30]	Documentación de GitLab CI/CD	Servicio de integración y despliegue continuo.
[REF31]	Documentación de LogRocket	Herramienta de monitoreo y análisis de sesiones.
[REF32]	Discord para Desarrolladores	API y herramientas para integrar Discord.
[REF33]	(eDelivery)(AP)(SAD)(3.4) (1	Guía arquitectónica para la aplicación Checklist de Estado de Campus.

1.4. Resumen del Contenido del Documento

Después de resumir la representación arquitectónica, objetivos y restricciones, este documento describe el sistema utilizando varias vistas arquitectónicas:

- **Vista de Caso de Uso**: Describe los requisitos funcionales y las interacciones entre los usuarios y el sistema.
- **Vista Lógica**: Ilustra la estructura de alto nivel del sistema, incluyendo componentes y sus relaciones.
- **Vista de Procesos**: Muestra el comportamiento en tiempo de ejecución del sistema, incluyendo procesos e hilos.
- **Vista de Despliegue**: Detalla cómo se despliega el sistema en la infraestructura física y en la nube.
- **Vista de Implementación**: Se enfoca en la organización del código y las herramientas de desarrollo.
- **Vista de Datos**: Describe el modelo de datos y el esquema de la base de datos.

El documento concluye con consideraciones sobre el tamaño del sistema, rendimiento y atributos de calidad como escalabilidad, seguridad y mantenibilidad.

2. REPRESENTACIÓN ARQUITECTONICA

En las secciones siguientes del documento se describen los objetivos y las restricciones de la arquitectura.

2.1. Objetivos Arquitectonicos

Los principales objetivos arquitectónicos del sistema **Checklist de Estado de Campus** son:

1. **Escalabilidad**: Garantizar que la aplicación pueda manejar un crecimiento en el número de usuarios y datos sin comprometer el rendimiento.
2. **Disponibilidad**: Asegurar que la aplicación esté accesible en todo momento, especialmente para los usuarios que realizan inspecciones en tiempo real.
3. **Seguridad**: Proteger los datos sensibles de los usuarios y garantizar la integridad de la información almacenada.
4. **Mantenibilidad**: Facilitar la actualización y el mantenimiento del sistema mediante una arquitectura modular y bien documentada.
5. **Interoperabilidad**: Permitir la integración con otros sistemas de la UTPL, como sistemas de inventario o mantenimiento.

2.2. Restricciones Arquitectónicas

Las principales restricciones arquitectónicas son:

1. **Tecnologías específicas**: Uso de Firebase para la base de datos y autenticación, React para el frontend, y Node.js para el backend.
2. **Despliegue en la nube**: La aplicación debe estar alojada en infraestructura en la nube para garantizar accesibilidad y escalabilidad.
3. **Compatibilidad móvil**: La aplicación debe ser compatible con dispositivos móviles a través de Expo y estar disponible en Google Play Store.
4. **Cumplimiento de normativas**: La aplicación debe cumplir con las políticas de seguridad y privacidad de la UTPL.

Los diagramas UML se utilizan sistemáticamente para representar las diferentes vistas del sistema

3. OBJETIVOS Y RESTRICCIONES ARQUITECTÓNICAS

Se han identificado los siguientes requisitos no funcionales que afectan a la solución arquitectónica:

Requisitos no funcionales	**Descripción**
Adaptabilidad	La aplicación debe ser fácil de integrar en los flujos de trabajo existentes de la UTPL, utilizando diferentes protocolos de comunicación y formatos de datos.
Portabilidad	La aplicación debe poder desplegarse en una amplia variedad de sistemas de hardware/software, incluyendo dispositivos móviles y entornos en la nube.
Interoperabilidad	El sistema debe ser interoperable con otras aplicaciones y servicios utilizados por la UTPL, como sistemas de inventario, mantenimiento y gestión de campus.
Escalabilidad	La aplicación debe ser capaz de manejar un crecimiento en el número de usuarios y datos sin comprometer el rendimiento.
Disponibilidad	La aplicación debe estar accesible en todo momento, especialmente para los usuarios que realizan inspecciones en tiempo real.
Seguridad	La aplicación debe proteger los datos sensibles de los usuarios y garantizar la integridad de la información almacenada.
Mantenibilidad	La arquitectura debe facilitar la actualización y el mantenimiento del sistema mediante un diseño modular y bien documentado.
Rendimiento	La aplicación debe responder rápidamente a las solicitudes de los usuarios, incluso bajo cargas de trabajo elevadas.

4. SEGURIDAD

4.1. Introducción

La aplicación "Checklist de Estado de Campus" incorpora medidas de seguridad robustas siguiendo las mejores prácticas de la industria y los estándares de seguridad más recientes. Estas medidas garantizan la protección de los datos sensibles y la integridad del sistema, al mismo tiempo que permiten una integración sencilla con los dominios de seguridad existentes de la UTPL.

4.2. Comunicación entre Componentes

Dado que no se pueden hacer suposiciones sobre la arquitectura de seguridad de los sistemas externos (como los sistemas de inventario o mantenimiento de la UTPL), la aplicación está diseñada para ser flexible y permitir la integración con diferentes esquemas de seguridad a través de plugins o configuraciones personalizadas. Esto asegura que la aplicación pueda adaptarse a las necesidades de seguridad específicas de la UTPL.

4.3. Comunicación Cliente-Servidor

La comunicación entre el cliente (aplicación móvil) y el servidor (backend) cumple con los requisitos de seguridad especificados en los estándares modernos de desarrollo de aplicaciones. Esto incluye:

- **Autenticación**: Uso de MongoDB para gestionar la autenticación de usuarios mediante correo electrónico y contraseña y a futuro proveedores de identidad externos (Azure).
- **Encriptación**: Todos los datos transmitidos entre el cliente y el servidor están encriptados mediante HTTPS (TLS/SSL).
- **Autorización**: Implementación de controles de acceso basados en roles (RBAC) para garantizar que los usuarios solo puedan acceder a las funcionalidades y datos correspondientes a su perfil.

4.4. Comunicación entre Servicios Internos

La seguridad en la comunicación entre los servicios internos (por ejemplo, entre el backend y la base de datos de Firebase) se maneja mediante los mismos mecanismos utilizados en la comunicación cliente-servidor, garantizando la integridad y confidencialidad de los datos en todo momento.

4.5. Acceso a Interfaces Administrativas

El acceso a las interfaces administrativas de la aplicación (como el panel de control para gestionar usuarios y edificios) está protegido con autenticación de usuario y contraseña. Las credenciales se almacenan de manera segura en la base de datos de MongoDB, utilizando una función de hash fuerte (Bcrypt) para garantizar su confidencialidad.

DECLARACIÓN DE SEGURIDAD

Además de las medidas de seguridad integradas en la aplicación "Checklist de Estado de Campus", el usuario final (UTPL) debe implementar medidas de seguridad adicionales de acuerdo con las mejores prácticas y regulaciones aplicables. Esto incluye, pero no se limita a:

- Uso de firewalls para restringir el acceso no autorizado.
- Implementación de listas blancas de IP para limitar el acceso a los servidores.
- Encriptación de archivos y bases de datos para proteger la información sensible.
- Monitoreo continuo de la seguridad para detectar y responder a posibles vulnerabilidades.

.

5. VISTA DE CASOS DE USO

Esta sección proporciona una representación de los casos de uso relevantes para la arquitectura.

5.1. Justificación de la selección

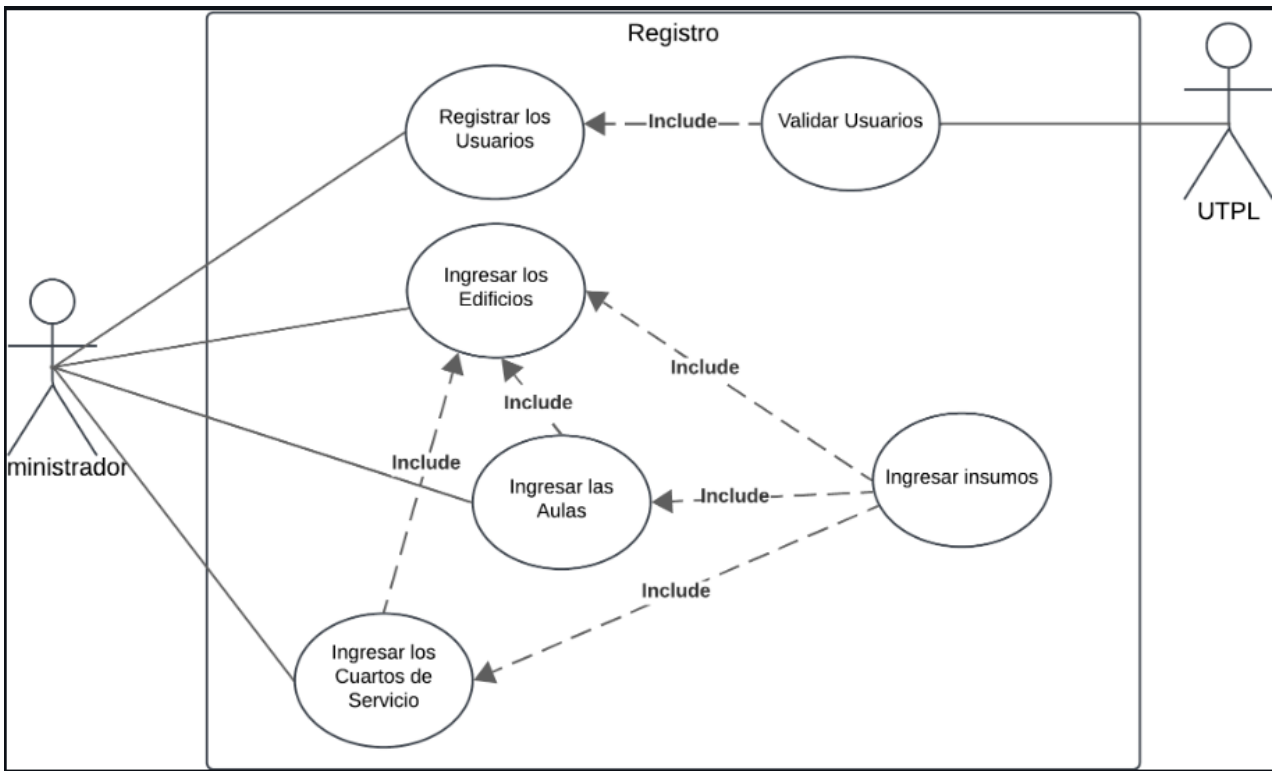
Los casos de uso relevantes para la arquitectura se han seleccionado basándose en los siguientes criterios:

- **Casos de uso que afectan la interacción entre los sistemas externos y la aplicación**: Por ejemplo, la integración con sistemas de inventario o mantenimiento de la UTPL.
- **Casos de uso que representan partes críticas de la arquitectura**: Estos casos ayudan a abordar los riesgos técnicos del proyecto en una etapa temprana.

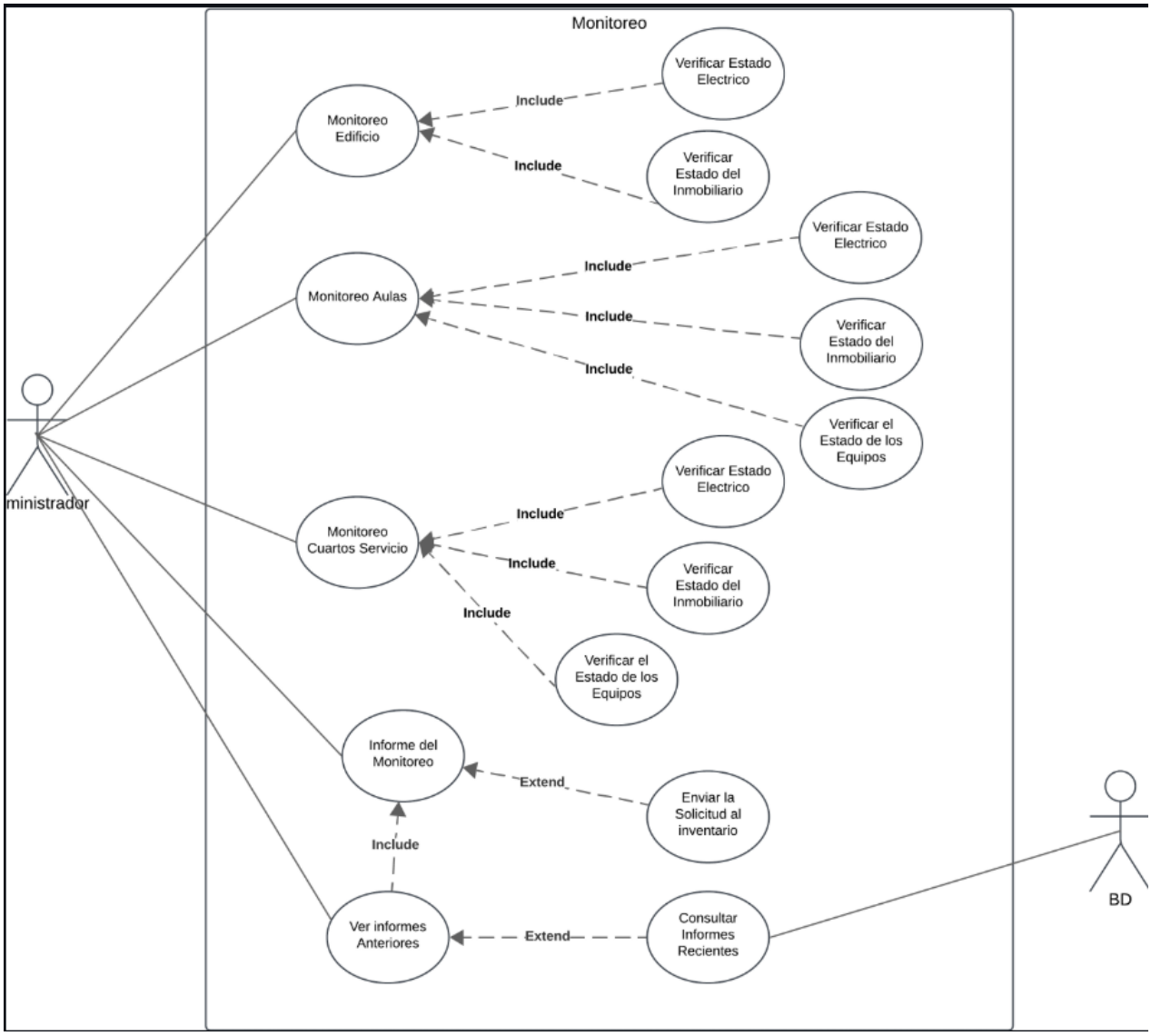
Los siguientes casos de uso han sido seleccionados:

#	Nombre del Caso de Uso	Actores	Flujo Normal	Consideraciones
1	Integración Pull con WebService	- Sistema Back-office - Domibus MSH	1. El Sistema Back-office solicita datos mediante una llamada de WebService. 2. Domibus MSH procesa la solicitud y recupera la información requerida. 3. Domibus MSH devuelve los datos al Sistema Back-office. 4. El Sistema Back-office recibe y procesa la respuesta.	- Comunicación síncrona - Basado en estándares de WebService - Ideal para consultas puntuales y recuperación de información
2	Integración Push con JMS	- Sistema Back-office - Domibus MSH	1. El Sistema Back-office envía un mensaje a través de una cola JMS. 2. Domibus MSH recibe el mensaje de la cola. 3. Domibus MSH procesa y enruta el mensaje. 4. El Sistema Back-office recibe confirmación de entrega.	- Comunicación asíncrona - Basado en tecnología de mensajería JMS - Adecuado para procesamiento de lotes y eventos asincrónicos
3	Gestión Administrativa del Sistema	- Administrador del Sistema	1. El Administrador accede a la interfaz administrativa. 2. Configura parámetros de integración. 3. Gestiona usuarios y permisos. 4. Monitorea el estado de las integraciones. 5. Genera informes de actividad y rendimiento.	- Interfaz centralizada de gestión - Control de configuraciones de integración - Monitoreo

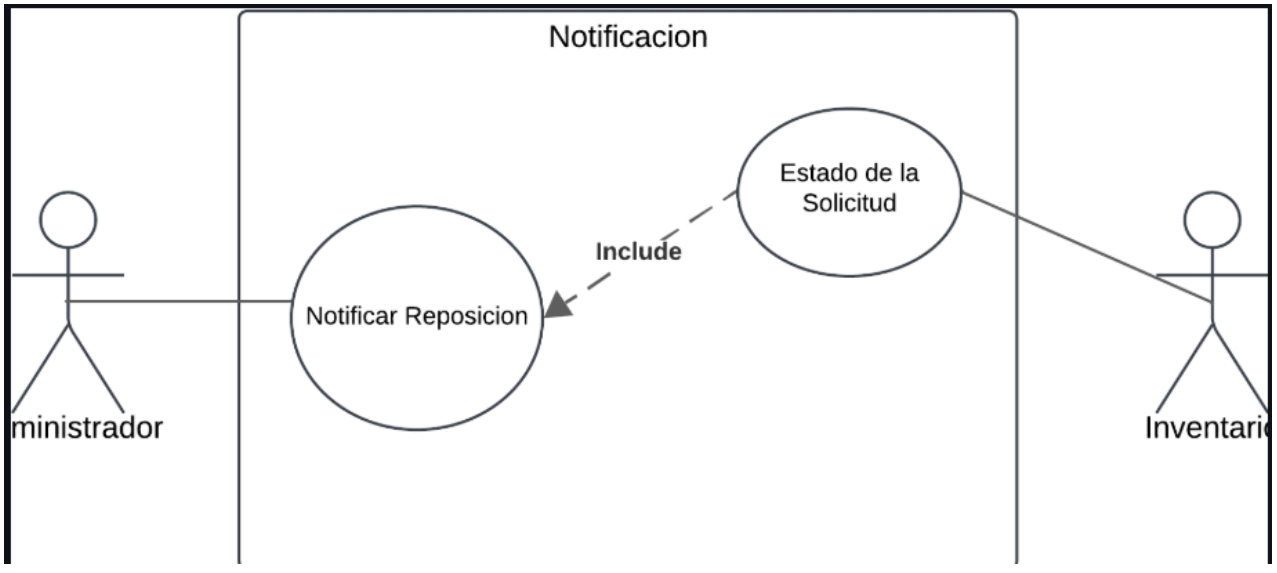
1. Diagrama de caso de uso (Registro)



2. Diagrama de caso de uso (Monitoreo)



3. Diagrama de caso de uso (Notificación)



6. VISTA LÓGICA

6.1. Visión general

Este capítulo describe los principales módulos de la aplicación **Checklist de Estado de Campus**, cómo interactúan entre sí y cómo implementan los requisitos funcionales y no funcionales del sistema. La vista lógica proporciona una representación de alto nivel de la estructura del sistema, mostrando los componentes clave y sus relaciones.

6.2. Diagrama de Clases

Un diagrama de clases es un tipo de diagrama de estructura estática que muestra la estructura de un sistema mostrando las clases del sistema, sus atributos, operaciones (o métodos), y las relaciones entre los objetos. Es la piedra angular del modelado orientado a objetos.

Los diagramas de clases son ampliamente utilizados en el proceso de diseño de sistemas orientados a objetos, ya que permiten a los desarrolladores y otros interesados visualizar:

- Las clases que componen el sistema
- Los atributos y métodos de cada clase
- Las relaciones estáticas entre las clases
- Las restricciones y dependencias del sistema

Este tipo de diagrama es esencial en la fase de diseño del desarrollo de software, ya que proporciona una vista clara de la arquitectura del sistema antes de su implementación. Ayuda a los desarrolladores a comprender la estructura general del sistema y cómo los diferentes componentes interactúan entre sí.

Los diagramas de clases son particularmente útiles para:

- Ilustrar modelos de datos para sistemas de información
- Comprender mejor la estructura general de un sistema
- Documentar y visualizar la arquitectura del software
- Servir como base para la implementación del código

El diagrama muestra la estructura principal del sistema, compuesta por los siguientes módulos principales:

Diagrama de clases

6.2.1. *Clases principals del sistema*

Las clases principales y sus relaciones son:

- **Persona**: Almacena información básica de usuarios
- **Administrador**: Gestiona accesos y operaciones principales
- **Edificio**: Contiene información estructural y referencias a aulas y servicios
- **Aulas**: Gestiona espacios educativos
- **CuartoServicio**: Administra áreas de servicio
- **Informe**: Maneja documentación de inspecciones
- **UTPL**: Provee servicios de validación
- **Inventario**: Controla recursos y solicitudes

El diagrama muestra relaciones de asociación y composición entre las clases, donde el Administrador actúa como punto central de control, conectando con el Edificio y la validación UTPL.

7. VISTA DE PROCESOS

7.1. Visión general

La vista de procesos describe el comportamiento dinámico del sistema, mostrando cómo los diferentes componentes interactúan entre sí a lo largo del tiempo. Esta vista es crucial para entender el flujo de control y datos en el sistema.

7.2. Diagrama de Secuencia

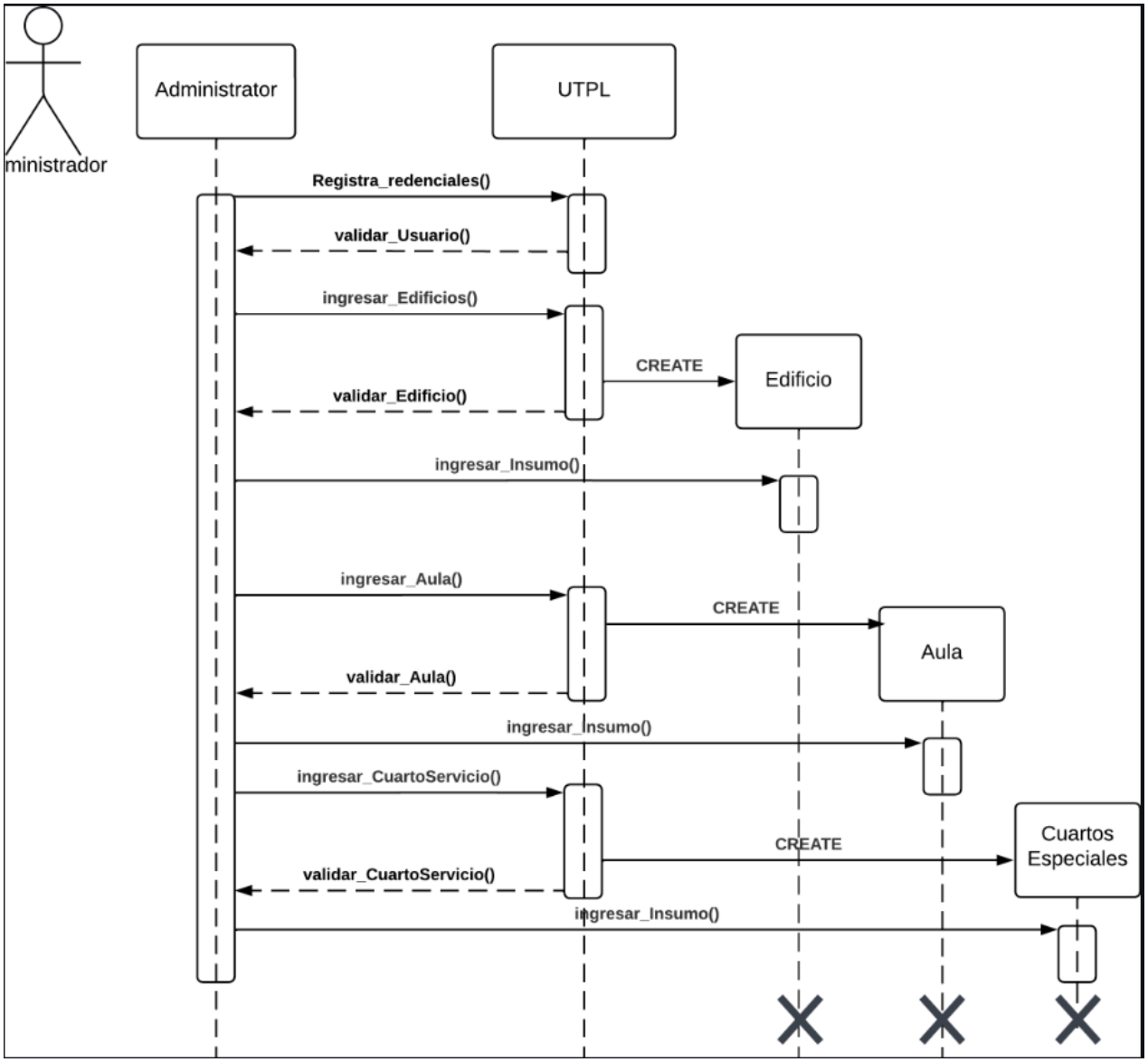
Un diagrama de secuencia es un tipo de diagrama de interacción que muestra cómo los procesos operan entre sí y en qué orden. Ilustra:

- La secuencia temporal de mensajes entre objetos
- Las interacciones de objetos organizadas en orden cronológico
- Los ciclos de vida de varios objetos y componentes del sistema

Elementos principales:

- Línea de vida: Representa el tiempo de vida de un objeto
- Mensajes: Comunicación entre objetos
- Activaciones: Periodos durante los cuales un objeto está ejecutando una operación
- Objetos/Actores: Participantes en la interacción

1. Diagrama de inicio de sesión y registro de edificios

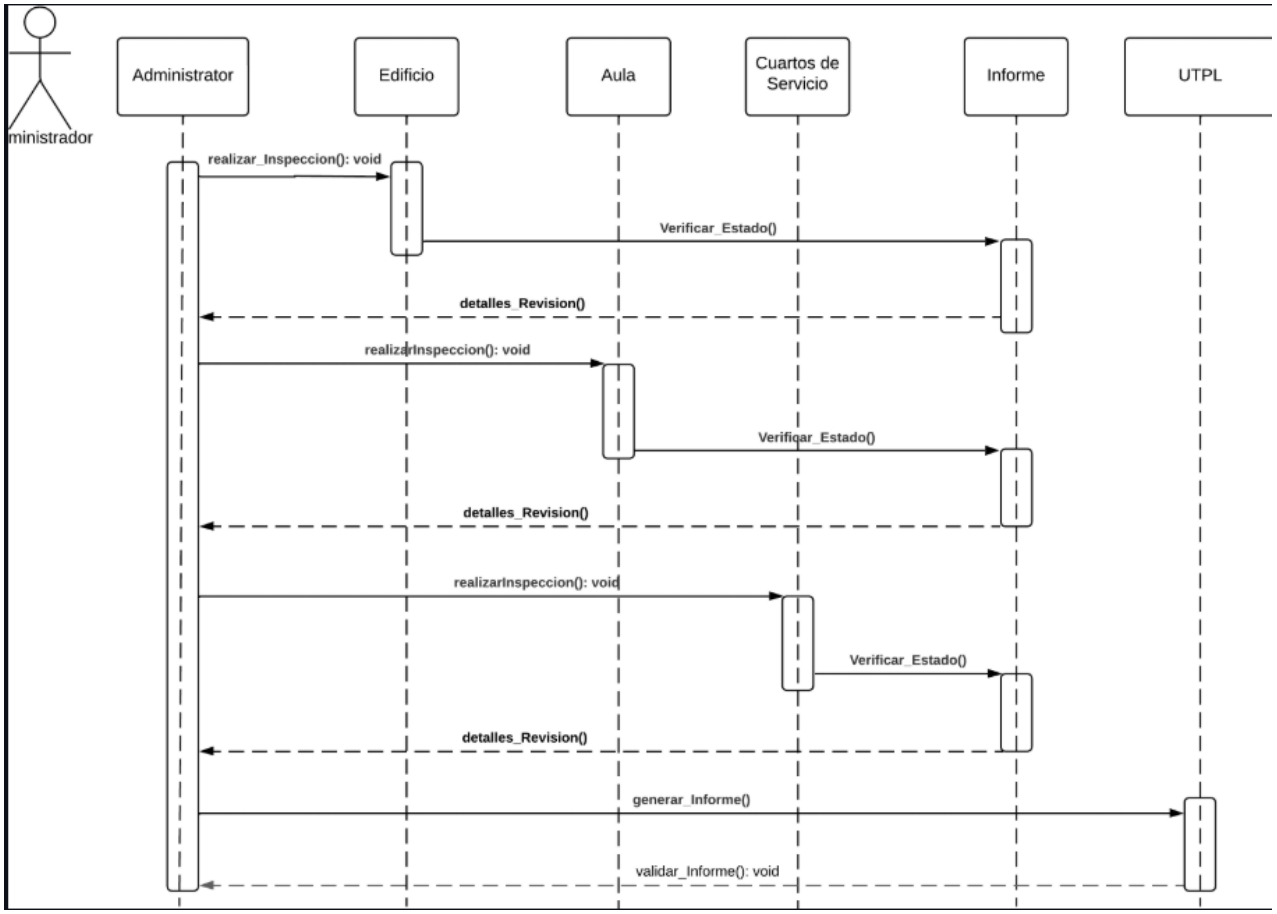


Explicación:
En el primer diagrama de secuencias se describe el flujo de interacción para los encargados de realizar el checklist del estado del campus. El proceso comienza cuando los usuarios inician sesión utilizando sus credenciales institucionales de la UTPL. Una vez ingresados los datos, el sistema valida las credenciales en la base de datos para garantizar la autenticidad del acceso.

Adicionalmente, el diagrama incluye el proceso de registro de nuevos edificios, aulas y cuartos de servicio en el sistema de la UTPL, destacando las interacciones necesarias para completar estas tareas de manera eficiente. El flujo es el siguiente:

- Ingresar Creedenciasles.
- Validar el usuario dentro de la UTPL.
- Registrar los edicios, aulas y cuartos de servicio.
- Agregar los insumos.

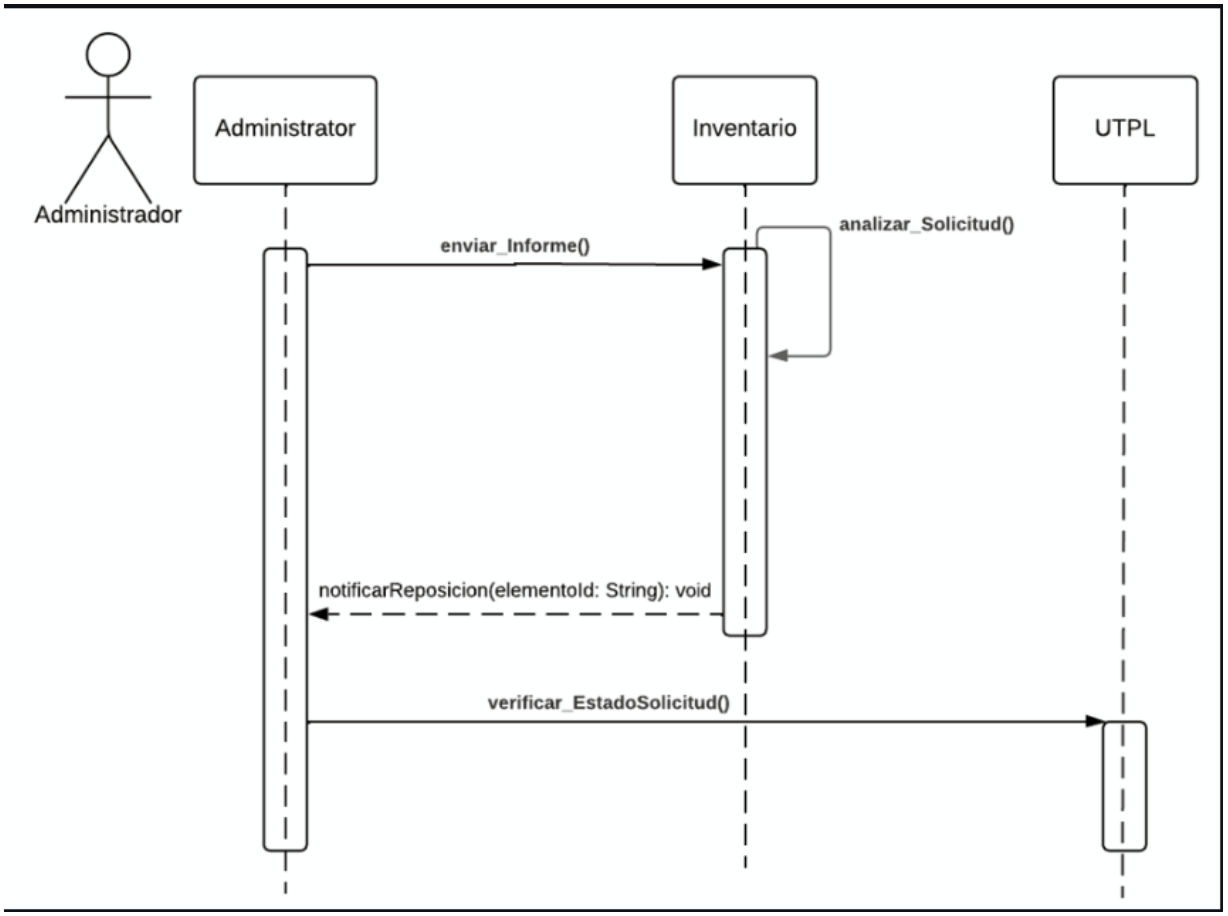
2. Diagrama de monitoreo de edificios



Explicación:
En el segundo diagrama se ilustra el proceso mediante el cual el usuario selecciona la parte específica del edificio que desea monitorear, permitiéndole generar un informe detallado sobre su estado. El flujo es el siguiente:

- Seleccioinar el edificio, aula y cuarto de servicio
- Escojer el apartado que se desea vereificar dentro de la parte seleccionada
- Realizar el proceso de verificacion
- Ver las observaciones guardadas en el informe
- Generar Informe

3. Diagrama de caso de envío de informe y proceso de solicitud



Explicación:
En el último diagrama se detalla el proceso mediante el cual el informe generado por el usuario sobre el estado del edificio es enviado al departamento de inventario. El propósito de este envío es que el equipo de inventario analice la información y proceda a realizar las acciones correspondientes, ya sea el mantenimiento o la reposición de los equipos necesarios. Finalmente, se notifica al usuario sobre la corrección realizada, cerrando el ciclo de seguimiento. El flujo es el siguiente:

- Enviar informe.
- Proceso de solicitud.
- Notificación al usuario.

7.3. Diagrama de Robustez

Un diagrama de robustez es una herramienta de análisis que ayuda a identificar objetos y sus interacciones en un escenario de caso de uso. Sirve como puente entre el análisis y el diseño. Muestra:

Elementos principales:

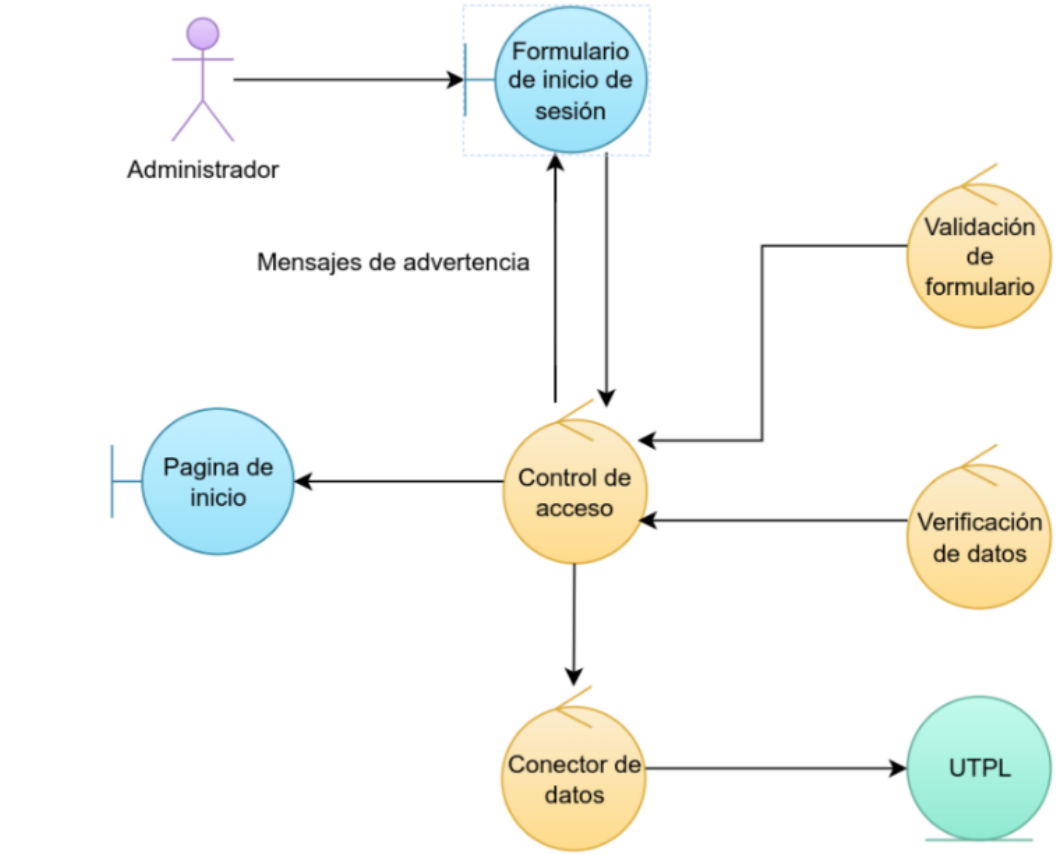
- Objetos Frontera: Representan interfaces con usuarios o sistemas externos
- Objetos Entidad: Representan información almacenada
- Objetos Controlador: Representan la lógica de negocio
- Conexiones: Muestran las relaciones entre objetos

Propósito

- Verificar que los casos de uso sean realizables

- Identificar objetos necesarios
- Validar la arquitectura del sistema
- Reducir la brecha entre análisis y diseño

Diagrama 1: Registro



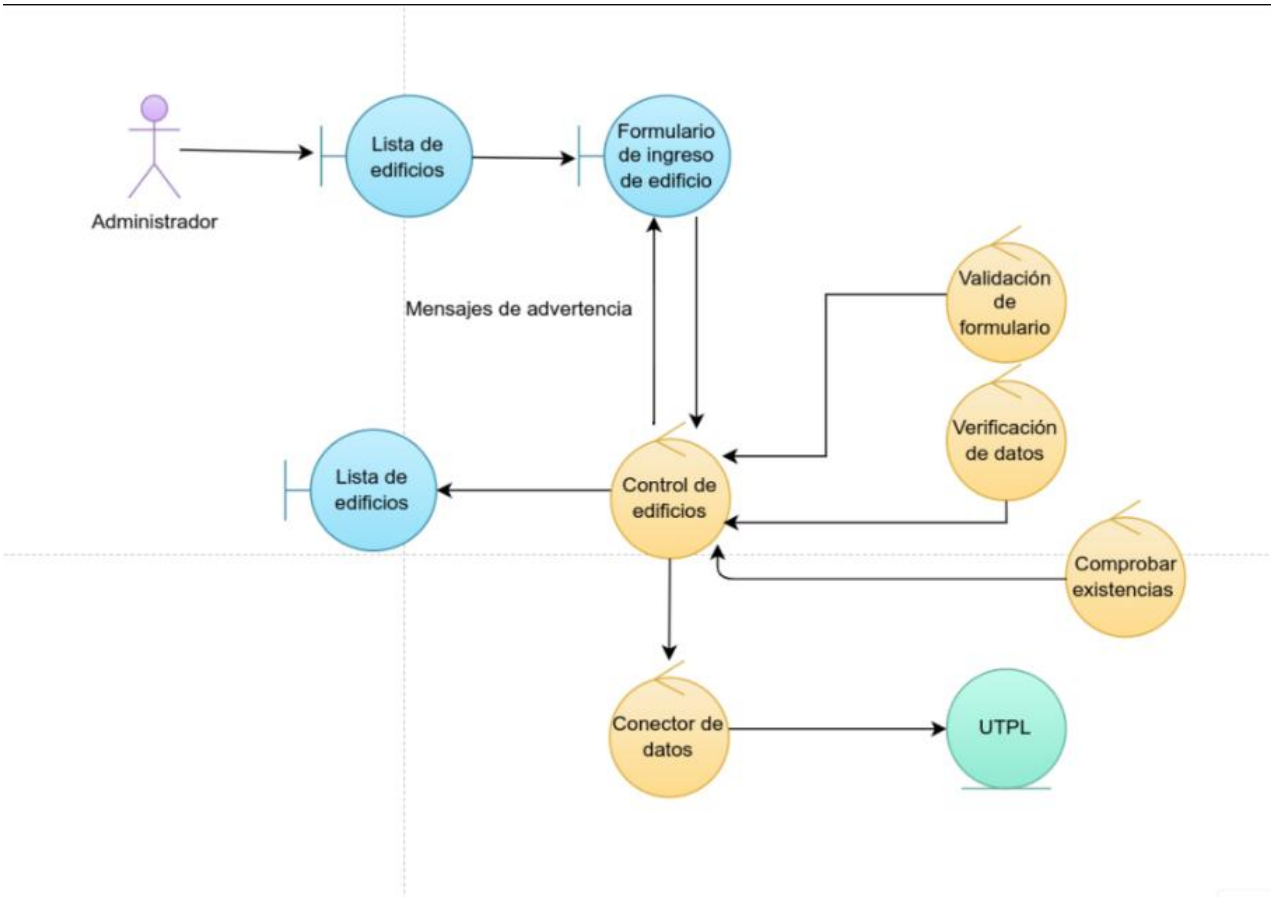


Diagrama 2: Monitoreo de los Edificios

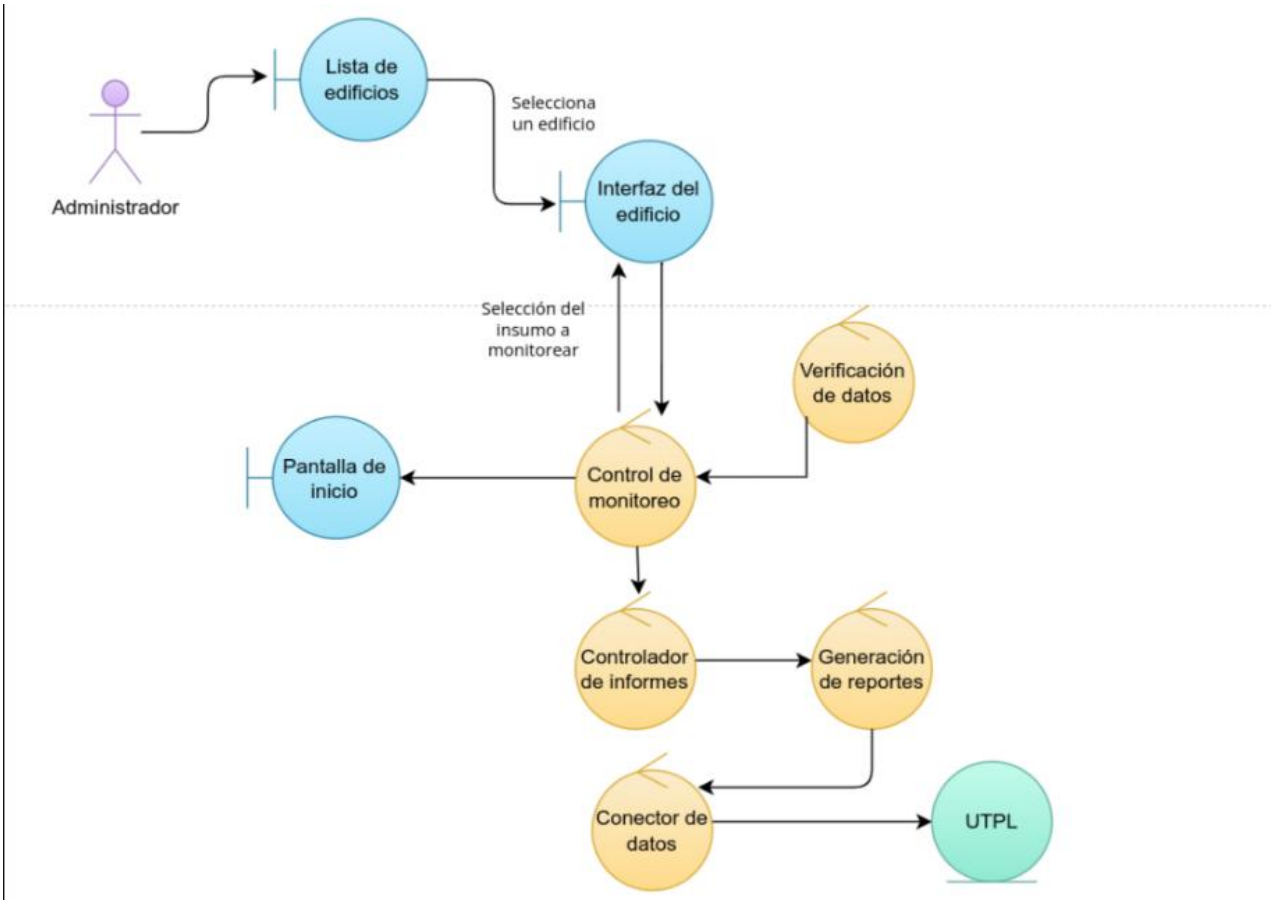
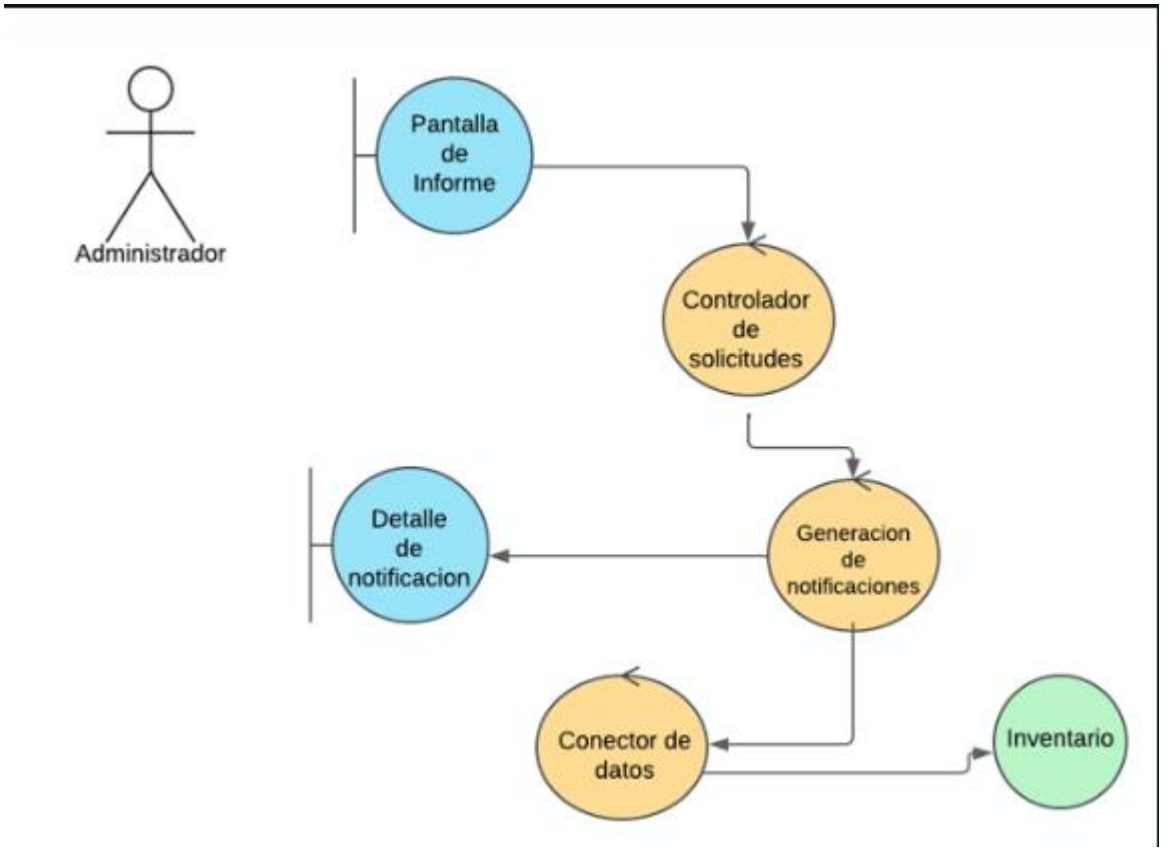


Diagrama 3: Notificaciones



8. VISTA DE DESARROLLO

8.1. Visión general

La vista de desarrollo describe la implementación física del sistema, detallando la arquitectura tecnológica, herramientas, procesos y estrategias de desarrollo utilizadas para construir la aplicación.

8.2. Arquitectura de Desarrollo

El proyecto se estructura en dos componentes principales:

1. **Backend**
    - Tecnologías: Node.js, Firebase
    - Bases de datos: MongoDB, Firebase
    - Enfoque: Desarrollo de servicios, lógica de negocio y gestión de datos
2. **Frontend**
    - Tecnologías: React, Expo, Tailwind CSS
    - Plataforma: Aplicación móvil multiplataforma
    - Enfoque: Interfaz de usuario, experiencia del usuario, integración de APIs

.

8.3. Metodología del Desarrollo

Características principales:

- **Metodología Ágil**: Desarrollo iterativo con sprints
- **CI/CD**: Integración y despliegue continuos
- **DevOps**: Integración de desarrollo y operaciones

¿Qué es CI/CD?

CI/CD, que significa *Integración Continua y Entrega/Despliegue Continuo*, es un conjunto de prácticas que automatizan las etapas de desarrollo, prueba y despliegue del software. Su objetivo principal es mejorar la velocidad, calidad y confiabilidad de las entregas de software.

- *Integración Continua (CI)*:
    - Es el proceso de integrar regularmente el código de diferentes desarrolladores en un repositorio compartido.
    - Incluye la ejecución automática de pruebas para identificar errores rápidamente.
- *Entrega Continua (CD - Continuous Delivery)*:
    - Extiende la CI al automatizar la preparación de entregas del software en cualquier momento.
    - Garantiza que el código esté siempre en un estado listo para producción.
    - Requiere pruebas adicionales y validaciones antes del despliegue.
- *Despliegue Continuo (CD - Continuous Deployment)*:
    - Va un paso más allá y automatiza también el proceso de despliegue en producción.
    - Cada cambio que pasa las pruebas se despliega automáticamente.

El enfoque CI/CD fomenta ciclos de desarrollo cortos, iterativos y más seguros.

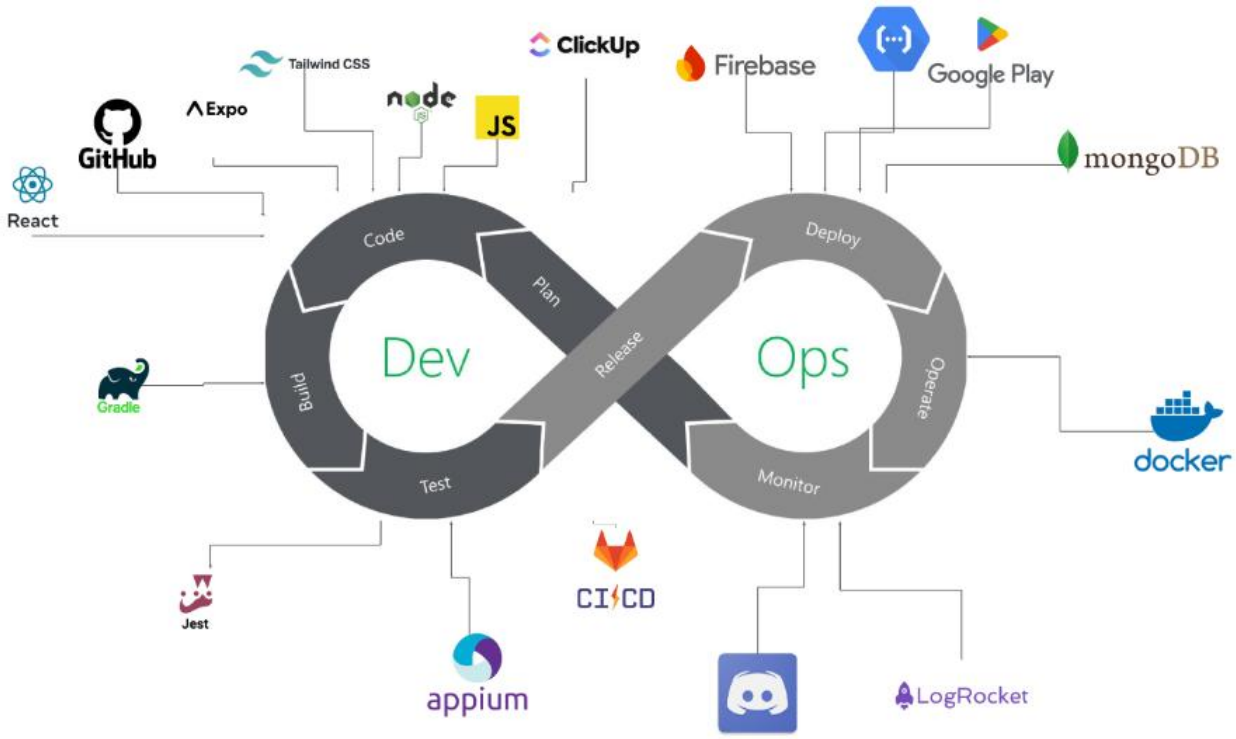
¿Qué es DevOps?

DevOps es una cultura, metodología y conjunto de prácticas que busca integrar los equipos de desarrollo (Development) y operaciones (Operations) para mejorar la colaboración, automatización y entrega continua de software. DevOps no es solo una herramienta, sino una filosofía que combina personas, procesos y tecnologías.

- *Beneficios de DevOps*:
    - Ciclos de entrega más rápidos.
    - Mayor estabilidad y calidad del software.

- Mejor alineación entre objetivos técnicos y empresariales.

8.4. Herramientas y Tecnologías Clave



Fase	Herramientas	Descripción
Plan	ClickUp	Herramienta utilizada para planificar el desarrollo y gestionar las tareas del proyecto. Permite organizar funciones, priorizar tareas y dar seguimiento de sprints en equipos ágiles.
Code	React, JavaScript, Node.js, Tailwind CSS, Expo	**React:** Base para la construcción de la interfaz de usuario de la aplicación. **JavaScript:** Lenguaje principal utilizado en el desarrollo. **Node.js:** Framework utilizado para el backend. **Tailwind CSS:** Facilita el diseño de interfaces con estilos predefinidos. **Expo:** Permite desarrollar aplicaciones móviles rápidamente con React Native.
Build	Gradle	Herramienta de automatización de la construcción, especialmente útil para gestionar componentes nativos y la creación de versiones para Android.
Test	Jest, JMeter, Google Lighthouse, Appium	**Jest:** Pruebas unitarias en JavaScript. **JMeter:** Permite pruebas de carga para evaluar el comportamiento bajo múltiples usuarios. **Google Lighthouse:** Analiza el rendimiento y la accesibilidad de la aplicación. **Appium:** Pruebas funcionales para aplicaciones móviles en múltiples plataformas.
Release	GitLab CI/CD	Se encarga del empaquetado, revisión y despliegue de la aplicación para garantizar su disponibilidad en plataformas como Google Play Store.
Deploy	MongoDB, Firebase,	**MonogoDB:** Maneja la parte de la Autentificación de la aplicación

	Google Play, Google Cloud Functions	**Firebase:** Maneja el backend de la aplicación. **Google Play:** Plataforma de distribución para los usuarios finales. **Google Cloud Functions:** Proporciona lógica de funciones para la aplicación.
Operate	Docker	Garantiza el funcionamiento continuo de la aplicación mediante monitoreo de errores, caídas y rendimiento.
Monitor	LogRocket, Discord	**LogRocket:** Rastrea la actividad de los usuarios y detecta problemas en la interfaz. **Discord:** Permite enviar notificaciones rápidas y colaborar en tiempo real.

9. VISTA DE DESPLIEGUE

9.1. Visión general

La vista de despliegue muestra la distribución física del sistema en el entorno de producción, ilustrando cómo los diferentes componentes de software se distribuyen a través de la infraestructura de hardware y servicios en la nube.

Componentes Principales

1. **Aplicación Cliente**
    - Dispositivo móvil con la aplicación desarrollada
    - Interactúa con la API Gateway
2. **API Gateway**
    - Gestiona el enrutamiento y la autenticación
    - Sirve como punto de entrada único al sistema
    - Maneja las solicitudes HTTP REST
3. **Google Cloud Functions**
    - Aloja las funciones serverless del sistema
    - Implementa la lógica de negocio
    - Incluye componentes para:
        - Autenticación
        - Registro
        - Servicios principales de monitoreo
        - Notificaciones
4. **Servicios Externos**
    - MongoDB Atlas: Para gestión de base de datos
    - Cloud Storage: Almacenamiento en la nube
    - Servicio MongoDB: Para autenticación y datos

Conexiones

- Conexiones HTTP/REST entre componentes
- Comunicación segura mediante protocolos estándar
- Integración con servicios cloud de Google

Aspectos de Despliegue

- Arquitectura serverless
- Escalabilidad automática
- Alta disponibilidad
- Gestión de recursos cloud

10. VISTA DE DATOS

10.1. Modelo de Datos

Un **modelo de datos** es una representación estructurada que define cómo se organizan, almacenan y manipulan los datos dentro de un sistema o base de datos. Es un diseño conceptual que describe:

- **Entidades**: Objetos o conceptos del mundo real, como "Clientes", "Productos" o "Órdenes".
- **Atributos**: Propiedades o características de las entidades, como el "nombre" de un cliente o el "precio" de un producto.
- **Relaciones**: Conexiones o asociaciones entre entidades, como la relación entre "Clientes" y "Órdenes".

El propósito principal de un modelo de datos es facilitar la comprensión y gestión de los datos en sistemas complejos, asegurando que estén organizados de manera lógica y eficiente

10.2. Estructura general de un modelo de datos

Un modelo de datos se compone de los siguientes elementos clave:

1. Entidades

Representan los objetos principales que necesitan ser modelados dentro del sistema. Ejemplo: Un cliente puede ser una entidad en un sistema de ventas.

2. Atributos

Son las características o propiedades que describen una entidad. Ejemplo: Los atributos de un cliente pueden incluir:

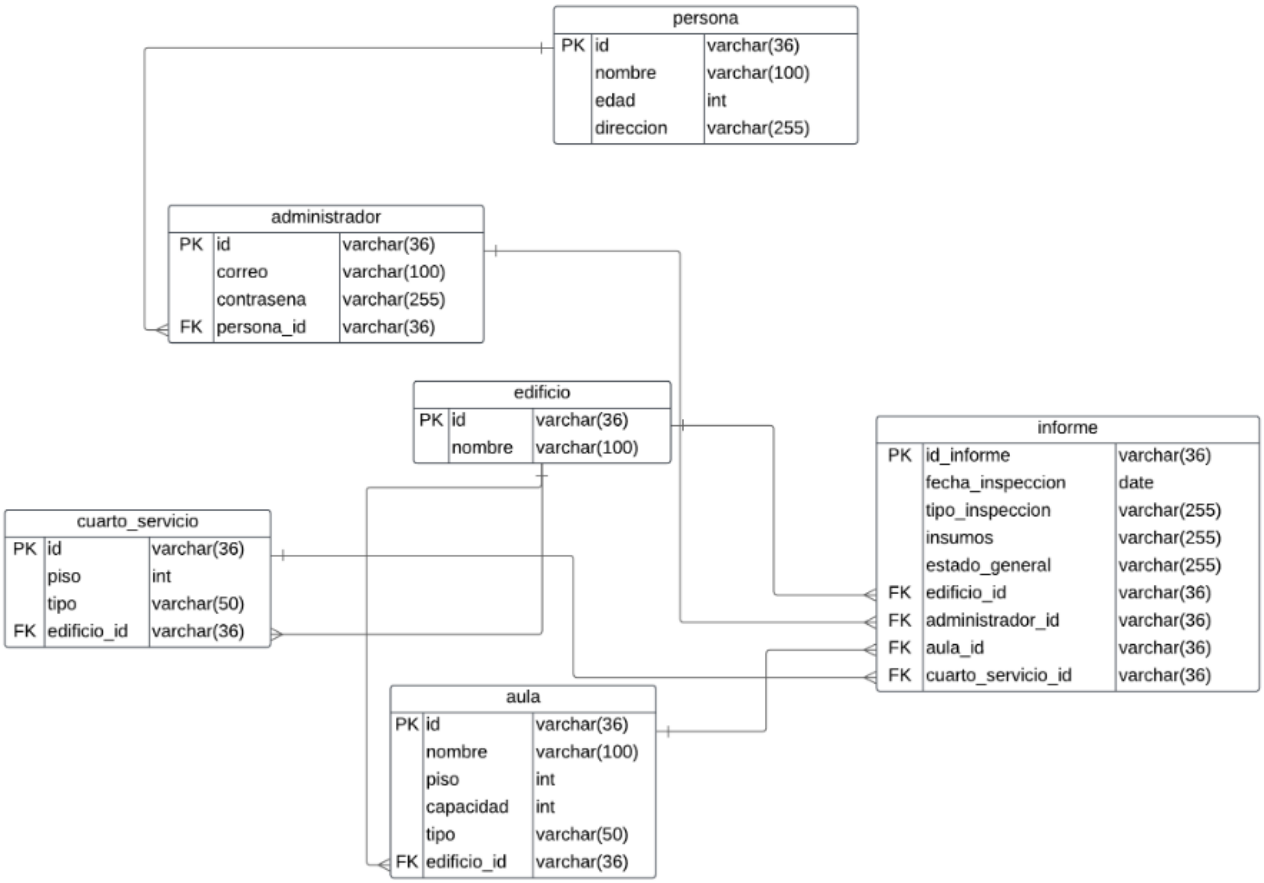
- Nombre
- Dirección
- Teléfono

3. Relaciones

Describen cómo las entidades están conectadas entre sí y cómo interactúan. Ejemplo: Un cliente puede tener una relación de "realiza" con varias órdenes en un sistema de ventas.

Tipos comunes de modelos de datos:

- **Modelo conceptual**: Representación abstracta, como diagramas Entidad-Relación (ER).
- **Modelo lógico**: Especifica estructuras detalladas como tablas y columnas.
- **Modelo físico**: Implementación en un sistema de base de datos específico.


persona
PK id varchar(36)
nombre varchar(100)
edad int
direccion varchar(255)
administrador
PK id varchar(36)
correo varchar(100)
contrasena varchar(255)
FK persona_id varchar(36)
edificio
PK id varchar(36)
nombre varchar(100)
informe
PK id_informe varchar(36)
fecha_inspeccion date
tipo_inspeccion varchar(255)
insumos varchar(255)
estado_general varchar(255)
FK edificio_id varchar(36)
FK administrador_id varchar(36)
FK aula_id varchar(36)
FK cuarto_servicio_id varchar(36)
cuarto_servicio
PK id varchar(36)
piso int
tipo varchar(50)
FK edificio_id varchar(36)
aula
PK id varchar(36)
nombre varchar(100)
piso int
capacidad int
tipo varchar(50)
FK edificio_id varchar(36)

11. Tamaño y rendimiento

11.1. Visión general

Las restricciones de tamaño aplicadas a los datos intercambiados por los sistemas externos (como el sistema de inventario de la UTPL) tienen un impacto en la arquitectura y en la configuración del sistema. A continuación, se describen las consideraciones clave:

1. **Intercambio de Archivos Grandes**:
    - La aplicación **Checklist de Estado de Campus** permite el intercambio de archivos binarios grandes, como imágenes de inspecciones o documentos adjuntos. Para manejar estos archivos de manera eficiente, se utiliza **Firebase Storage**, que permite almacenar y recuperar archivos grandes sin sobrecargar la base de datos.
    - Los archivos se almacenan en Firebase Storage y se referencian en la base de datos mediante URLs, evitando el procesamiento de grandes objetos binarios (BLOBs) en la base de datos.

2. **Limitaciones de Tamaño**:
    - El tamaño máximo de los archivos adjuntos está limitado por la configuración de Firebase Storage y la capacidad de red. Actualmente, el límite es de **100 MB por archivo**.
    - El número máximo de archivos adjuntos por informe o inspección está configurado en **10 archivos**.

3. **Restricciones Adicionales**:
    - Se pueden implementar restricciones adicionales a través de la lógica de negocio, como límites en el tamaño total de los datos enviados en una sola solicitud o el número máximo de inspecciones simultáneas.

11.2. Rendimiento

El rendimiento de la aplicación **Checklist de Estado de Campus** es una prioridad, especialmente considerando la necesidad de manejar múltiples usuarios y grandes volúmenes de datos en tiempo real. A continuación, se describen las decisiones arquitectónicas que benefician el rendimiento del sistema:

1. **Desacoplamiento de Componentes**:
    - La aplicación está diseñada con una arquitectura desacoplada, donde el frontend (aplicación móvil) y el backend (servicios en la nube) interactúan a través de APIs RESTful. Esto permite una distribución eficiente de la carga y una escalabilidad horizontal.
    - El uso de **Firebase Firestore** como base de datos en tiempo real garantiza un acceso rápido a los datos y una sincronización automática entre dispositivos.

2. **Manejo de Solicitudes**:
    - Las solicitudes de los usuarios son manejadas por múltiples instancias del backend, ejecutándose en contenedores Docker y distribuidas mediante un balanceador de carga. Esto mejora el rendimiento y la disponibilidad del sistema.
    - Para operaciones asíncronas, como el envío de notificaciones o la comunicación con sistemas externos, se utiliza un sistema de mensajería (JMS o RabbitMQ), que permite procesar las solicitudes en segundo plano sin bloquear la interfaz de usuario.

3. **Optimización de Recursos**:
    - Los archivos grandes se almacenan en Firebase Storage en lugar de la base de datos, lo que

reduce la carga en el servidor de base de datos y mejora el rendimiento general.
    - Se implementan técnicas de caché para reducir la carga en el backend, como el almacenamiento en caché de datos frecuentemente accedidos (por ejemplo, listas de edificios y aulas).

4. **Pruebas de Rendimiento**:
    - Se realizan pruebas de carga y estrés para garantizar que el sistema pueda manejar un gran número de usuarios simultáneos sin degradar el rendimiento.
    - Se utilizan herramientas como **JMeter** para simular escenarios de alta carga y **Google Lighthouse** para optimizar el rendimiento del frontend

11.3. Consideraciones adicionales

- **Escalabilidad**: La arquitectura en la nube permite escalar horizontalmente los recursos según la demanda, garantizando un rendimiento óptimo incluso durante picos de uso.
- **Disponibilidad**: El uso de servicios gestionados como Firebase y Docker garantiza una alta disponibilidad y tolerancia a fallos.
- **Seguridad**: Aunque no se incluyen en este apartado, las medidas de seguridad (como la encriptación de datos y la autenticación robusta) también contribuyen al rendimiento al prevenir accesos no autorizados y ataques de denegación de servicio (DoS).

12. CALIDAD

La arquitectura de la aplicación **Checklist de Estado de Campus** está diseñada para mejorar la **extensibilidad**, **fiabilidad** y **portabilidad**, garantizando un sistema robusto y adaptable a las necesidades futuras de la UTPL.

12.1. Extensibilidad

La aplicación está diseñada de manera modular, lo que facilita la incorporación de nuevas funcionalidades y la actualización de componentes existentes sin afectar el funcionamiento general del sistema. A continuación, se describen las características clave que contribuyen a la extensibilidad:

1. **Arquitectura por Capas**:
   - El sistema está dividido en capas claramente definidas (frontend, backend, base de datos, servicios externos), lo que permite modificar o reemplazar una capa sin afectar a las demás.

2. **Uso de APIs Bien Definidas**:
   - Las interacciones entre los componentes del sistema se realizan a través de APIs RESTful y mensajes en cola (JMS), lo que facilita la integración con nuevos sistemas o servicios.

3. **Plugin Architecture**:
   - La aplicación podría extenderse para soportar plugins que agreguen funcionalidades específicas sin modificar el núcleo del sistema.

12.2. Fiabilidad

La fiabilidad del sistema se garantiza mediante mecanismos que aseguran la disponibilidad y la integridad de los datos, incluso en caso de fallos parciales. A continuación, se describen las características clave que contribuyen a la fiabilidad:

1. **Desacoplamiento mediante Colas de Mensajes (JMS)**:
   - Las operaciones críticas, como el envío de notificaciones o la comunicación con sistemas externos, se manejan mediante colas de mensajes (JMS o RabbitMQ). Esto permite que el sistema continúe funcionando incluso si un componente falla temporalmente.

2. **Mecanismo de Reintento Automático**:
   - En caso de fallos en la comunicación con servicios externos (por ejemplo, el sistema de inventario), el sistema implementara un mecanismo de reintento automático para garantizar que las solicitudes se procesen correctamente.

3. **Almacenamiento en la Nube**:
   - El uso de Firebase Firestore y Firebase Storage garantiza la disponibilidad y durabilidad de los datos, con copias de seguridad automáticas y replicación en múltiples ubicaciones.

12.3. Portabilidad

La aplicación está diseñada para ser portable y adaptable a diferentes entornos de despliegue, lo que facilita su implementación en diversas infraestructuras. A continuación, se describen las características clave que contribuyen a la portabilidad:

1. **Despliegue en Contenedores (Docker)**:
    - El backend de la aplicación se ejecuta en contenedores Docker, lo que permite su despliegue en cualquier entorno compatible con Docker, ya sea en la nube o en servidores locales.
2. **Independencia de la Base de Datos**:
    - Aunque actualmente se utiliza Firebase Firestore, el diseño modular permite migrar a otras bases de datos (por ejemplo, MongoDB o MySQL) con cambios mínimos en el código.
3. **Compatibilidad con Diferentes Plataformas**:
    - El frontend de la aplicación está desarrollado con **React Native** y **Expo**, lo que garantiza su funcionamiento en dispositivos iOS y Android sin necesidad de modificaciones significativas.
4. **Uso de Estándares Abiertos**:
    - Las APIs RESTful y el uso de formatos de datos estándar (JSON) facilitan la integración con otros sistemas y plataformas.

13. REGISTRO DE LOGS (LOGGING)

El sistema de registro de logs (**logging**) de la aplicación **Checklist de Estado de Campus** se basa en herramientas modernas y servicios en la nube para garantizar un seguimiento detallado y eficiente de las actividades del sistema. Las principales herramientas utilizadas son:

- **LogRocket**: Para el registro de actividades en el frontend (aplicación móvil), incluyendo interacciones del usuario, errores y rendimiento.
- **Discord**: Para las notificaciones de actividad dentro del repositorio de Github
- **Google Cloud Logging**: Para el registro de actividades en el backend, incluyendo solicitudes HTTP, errores de servidor y eventos de autenticación.
- **Firebase Crashlytics**: Para el registro de errores y fallos en la aplicación móvil.
- **MongoDB Logs**: Para el registro de consultas y operaciones en la base de datos.
- **React Native Logs**: Para el registro de eventos y errores específicos de la aplicación móvil.
- **NodeJS Logs**: Para el registro de eventos y errores específicos del backend.

13.1. Categoria de Logs

Los logs se dividen en varias categorías para facilitar su análisis y seguimiento:

- **Seguridad**: Registran eventos relacionados con la autenticación, autorización y acceso a recursos sensibles.
- **Negocio**: Registran eventos relacionados con las funcionalidades principales de la aplicación.
- **Errores**: Registran errores y excepciones que ocurren en el sistema.
- **Rendimiento**: Registran métricas relacionadas con el rendimiento del sistema.

13.2. Estructura de Logs

Cada registro de log sigue un formato estructurado para facilitar su análisis. El formato general es el siguiente:
[Fecha y Hora] [Nivel de Log] [Categoría] [Usuario] [Módulo] [Mensaje]

- **Fecha y Hora**: Marca de tiempo del evento.
- **Nivel de Log**: Registran eventos relacionados con las funcionalidades principales de la aplicación.
- **Categoría**: Tipo de log (Seguridad, Negocio, Errores, Rendimiento).
- **Usuario**: Usuario asociado al evento (si aplica).
- **Módulo:** Módulo o componente del sistema que generó el log.
- **Mensaje:** Descripción detallada del evento.

14. MULTITENANCIA

14.1. General

La aplicación **Checklist de Estado de Campus** está diseñada para soportar **multitenencia**, lo que permite a múltiples "inquilinos" (en este caso, diferentes departamentos de la UTPL) utilizar la misma instancia de la aplicación mientras mantienen sus datos aislados y seguros. Se consideraron varias opciones para implementar la multitenencia, y la seleccionada fue:

1. **Un Esquema por Inquilino**:
    - Los datos de cada inquilino se almacenan en el mismo servidor de base de datos, pero en esquemas separados.
    - Cuando se añade un nuevo inquilino, se crea un nuevo esquema en la misma instancia de la base de datos.
    - Ventajas:
        - Fácil de implementar y mantener.
        - Reutilización del mismo pool de conexiones para todos los inquilinos.
        - Aislamiento de datos entre inquilinos sin necesidad de múltiples instancias de base de datos.

2. **Otras Opciones Consideradas**:
    - **Una Base de Datos por Inquilino**: Cada inquilino tiene su propia base de datos. Aunque ofrece el máximo aislamiento, es complejo de mantener y escalar.
    - **Campo Discriminador**: Todos los datos se almacenan en las mismas tablas, con un campo adicional para distinguir entre inquilinos. Esta opción fue descartada debido a la falta de aislamiento físico y el impacto en el rendimiento.

14.2. Identificación del dominio

Para cada operación en la aplicación, es necesario identificar el dominio (inquilino) asociado. Esto se hace de la siguiente manera:

1. **Mensajes Salientes**:
    - El dominio se identifica basándose en la información de autenticación del usuario (por ejemplo, credenciales de acceso de la UTPL).
    - Ejemplo: Un usuario administrador inicia sesión y realiza una inspección. El sistema identifica automáticamente que pertenece al dominio "Administrador".

2. **Mensajes Entrantes**:
    - El dominio se identifica mediante un parámetro en la solicitud HTTP.
    - Ejemplo: Una solicitud de mantenimiento enviada al backend incluye el parámetro domain=Administrador, lo que permite al sistema procesar la solicitud en el esquema correcto.

14.3. Interfaz de Usuario (UI)

1. **Consola de Administración**:
    - La interfaz de usuario ha sido adaptada para soportar múltiples dominios.
    - Un superadministrador tiene acceso a todos los dominios, pero solo puede gestionar un dominio a la vez.
    - Los usuarios normales solo pueden acceder a los datos de su propio dominio.
2. **Seguridad**:
    - Las solicitudes REST desde la UI están protegidas para garantizar que los usuarios solo puedan acceder a los datos de su dominio.

15. INFORMACIÓN DE CONTACTO

Checklist Support Team