

Universidad Autónoma De La Ciudad De México
“Nada humano me es ajeno”

Profesor: Maxino Eduardo Sanchez Gutierrez

Marquez Madonado Aaron

Soria Guerra Erick

Flores Herrera Fernando

Diseño del Software

Plantel:Cuautepec



5.1 Introducción

El presente Documento de Diseño de Software (SDD) tiene como objetivo principal proporcionar una descripción detallada y estructurada de la arquitectura y diseño de la aplicación web de calculadora científica, siguiendo el estándar IEEE 1016-2021. Este documento servirá como guía técnica para:

Desarrolladores: Para implementar los módulos y componentes del sistema.

Equipo de Calidad: Para validar que el diseño cumple con los requisitos funcionales y no funcionales.

Mantenimiento futuro: Como referencia para actualizaciones o correcciones.

El SDD abordará todos los aspectos de diseño desde múltiples perspectivas (vistas), incluyendo:

- Estructura estática (componentes, clases).
- Comportamiento dinámico (flujos de interacción, estados).
- Interfaces internas/externas (APIs, contratos de UI).
- Patrones y decisiones técnicas (reutilización, escalabilidad).

Alcance del Diseño

El diseño cubre los siguientes elementos clave de la aplicación:

Frontend (Interfaz de Usuario)

Interfaz gráfica: Diseño responsive con HTML/CSS/JavaScript.

Lógica de presentación: Manejo de eventos (clic en botones, validación de entrada).

Comunicación con backend: Llamadas REST/JSON para operaciones avanzadas.

Backend (Procesamiento)

Operaciones básicas: Suma, resta, multiplicación, división (ejecutadas en el frontend).

Operaciones avanzadas: Trigonometría (sin, cos), logaritmos, usando la librería Math.js.

Persistencia: Almacenamiento del historial en SQLite (modo offline) o Firebase (en línea).

Servicios Externos

Math.js: Para cálculos complejos y precisión numérica.

APIs de conversión: Ej. monedas, unidades (opcional).

5.2 punto de vista del contexto

a) Identificación del sujeto de diseño

Sistema: Calculadora científica web.

Propósito: Permitir a los usuarios realizar operaciones aritméticas básicas: Suma, resta, multiplicación y división, también funciones avanzadas: Cálculos trigonométricos (seno, coseno, tangente), logaritmos, exponenciales y raíces cuadradas y Conversión de sistemas de numeración: Binario, octal, decimal y hexadecimal.

Alcance: Solo cálculos matemáticos básicos, operaciones avanzadas y conversión de números, guarda el historial para futuras consultas

b) Actores del sistema

Usuarios:

momentáneo: Persona que accede a la calculadora de vez en cuando para consulta rápida.

Esta puede acceder a sumas básicas en consecuencia del poco uso no guarda el historial a pesar del poco uso que le de el usuario puede y podrá acceder a todas las funciones.

frecuente: usa comúnmente la página, ocupa cálculos avanzados y consulta su historial

El historial se muestra por fecha, esto ayudará a que se le hagan sugerencias con base en su historial al igual que el usuario poco frecuente este podrá acceder a todas las funciones de la página

Sistemas externos:

API de matemáticas avanzadas (si se usa una librería externa para cálculos complejos). Servidor web que aloja parte del código solo para en caso de que el usuario necesite hacer cálculos avanzados. La página sin conexión solo tendrá acceso a realizar operaciones básicas.

c) Entorno del sistema

Plataformas soportadas: Navegadores web en PC y dispositivos móviles.

Requisitos:

Conexión a Internet (si usa una API externa) siempre y cuando desee cálculos avanzados. Compatibilidad con HTML5, CSS y JavaScript.

Restricciones:

No requiere autenticación. Debe ser rápida y ligera para funcionar en cualquier dispositivo.

d) Relación con otros sistemas

avanzada: Puede conectarse a una API externa para cálculos más complejos (ejemplo: Wolfram Alpha o Math.js).

e) Limitaciones y restricciones

sin red puede ser demasiado básica. No cuenta con un formulario. No permite programación ni funciones gráficas. No da pasos detallados del resultado.

5.2.1 Preocupaciones de diseño (conversion"fer", avanzada "aaron", basica "erick", conexión "erick", sin conexión "fer" y previamente cargado "fer")

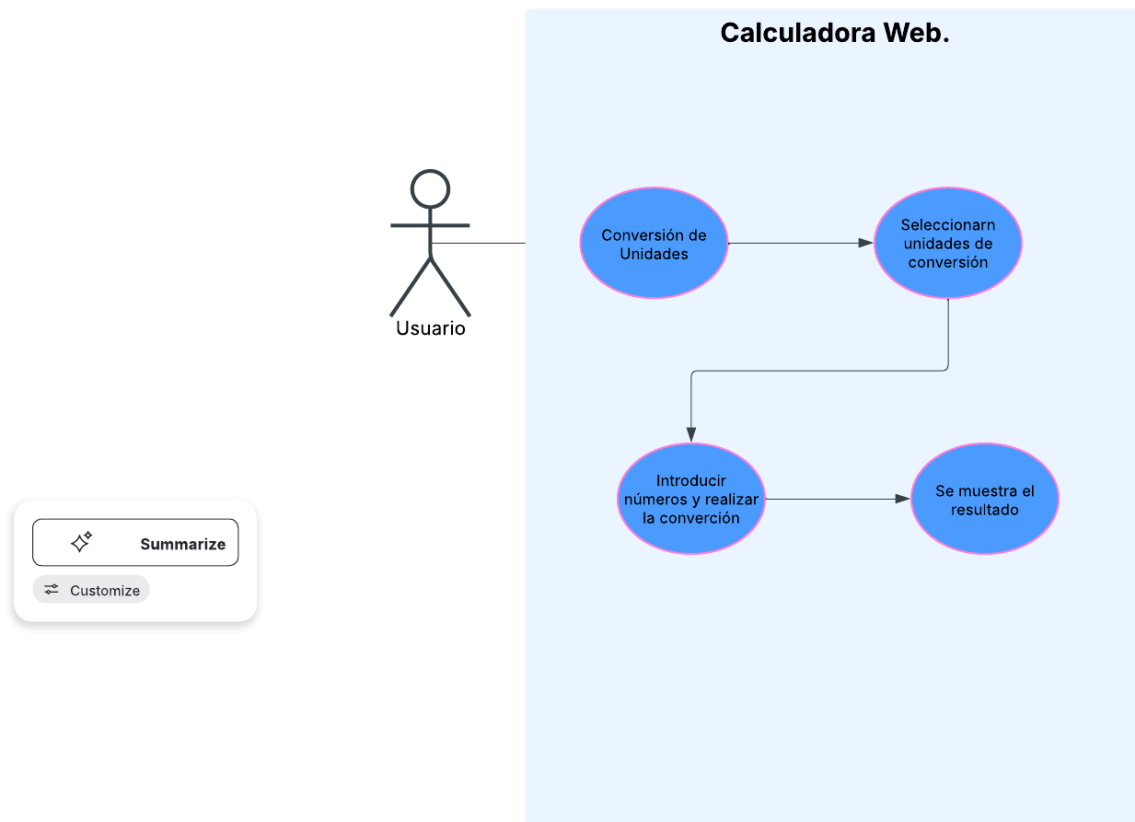
Explicación detallada del caso de uso:

Conversión de números.

Nombre: "Conversión"

Actor: Usuario

Diagrama detallado:



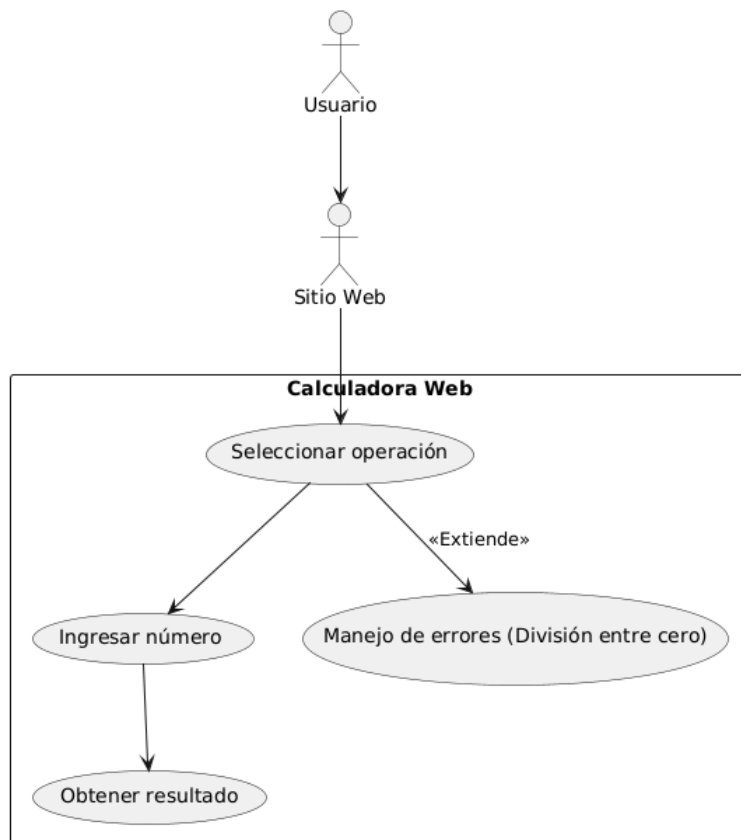
Descripción: Se planea que el usuario pueda realizar conversiones de números decimales a hexadecimales, binarios, octales y viceversa, además de poder hacer cualquier otro tipo de conversión sobre las opciones antes mencionadas.

Precondiciones: Que el usuario tenga un equipo compatible ya sea celular o computadora, cuente con conexión a internet (preferiblemente).

Flujo de eventos normales: Se realice correctamente la conversión solicitada, además de que sea rápido y se tenga un historial de las conversiones para una posible consulta de la convención.

Flujo de eventos alternativos: No se pudo realizar la convención por que el usuario no introdujo valores aceptables o permitidos, así como posibles imprecisiones al usar números con punto(flotantes) al hacer conversiones porque podrían llegar a redondear las cifras.

Operaciones básicas de la calculadora



1. Actores involucrados:

Usuario: Es la persona que interactúa con la calculadora. En el diagrama, es el actor que inicia las acciones al ingresar números, seleccionar una operación y obtener el resultado.

2. Sistema:

Calculadora Web: Es el sistema en el que se realizan las operaciones y que gestiona las interacciones del usuario.

Casos de uso (Usuario):

1. Ingresar números :

El Usuario inicia el proceso al ingresar dos números en los campos proporcionados en la interfaz de la calculadora. La calculadora puede pedir un número o varios dependiendo de la operación seleccionada. Este es el primer paso que permite al sistema realizar la operación matemática solicitada.

Ejemplo de interacciones:

El usuario ingresa un número (ejemplo: 5) en el campo "Número 1".

El usuario ingresa otro número (ejemplo: 3) en el campo "Número 2".

2. Seleccionar operación:

Después de ingresar los números, el Usuario selecciona la operación matemática que desea realizar, como Suma, Resta, Multiplicación o División. El sistema ofrece las opciones de operaciones, y el usuario selecciona la correspondiente, normalmente mediante botones (por ejemplo, un botón para sumar, restar, etc.). Dependiendo de la operación seleccionada, el sistema procederá a realizar los cálculos adecuados.

3. Obtener resultado:

Después de que el Usuario haya seleccionado la operación, el sistema calcula el resultado de la operación. El sistema muestra el resultado en la interfaz de usuario, ya sea como un número o un mensaje que indica el resultado de la operación.

Ejemplo de interacción:

El usuario selecciona la operación de suma ($5 + 3$) y la calculadora muestra el resultado como "8".

4. Manejo de errores:

Si el Usuario selecciona una operación que podría causar un error (como la división por cero), el sistema extiende el caso de uso para gestionar este tipo de errores. El sistema debe ser capaz de identificar que la operación es inválida (por ejemplo, $5 \div 0$) y mostrar un mensaje de error (ejemplo: "Error: División entre cero no permitida"). Este caso de uso es una extensión del caso de uso principal "Seleccionar operación", lo que significa que es una condición especial que se verifica si ocurre un intento de división por cero.

Relación entre casos de uso:

Flujo básico:

1. El Usuario ingresa los números.
2. El Usuario selecciona la operación.
3. El sistema realiza el cálculo y muestra el resultado.

Flujo alternativo (Error):

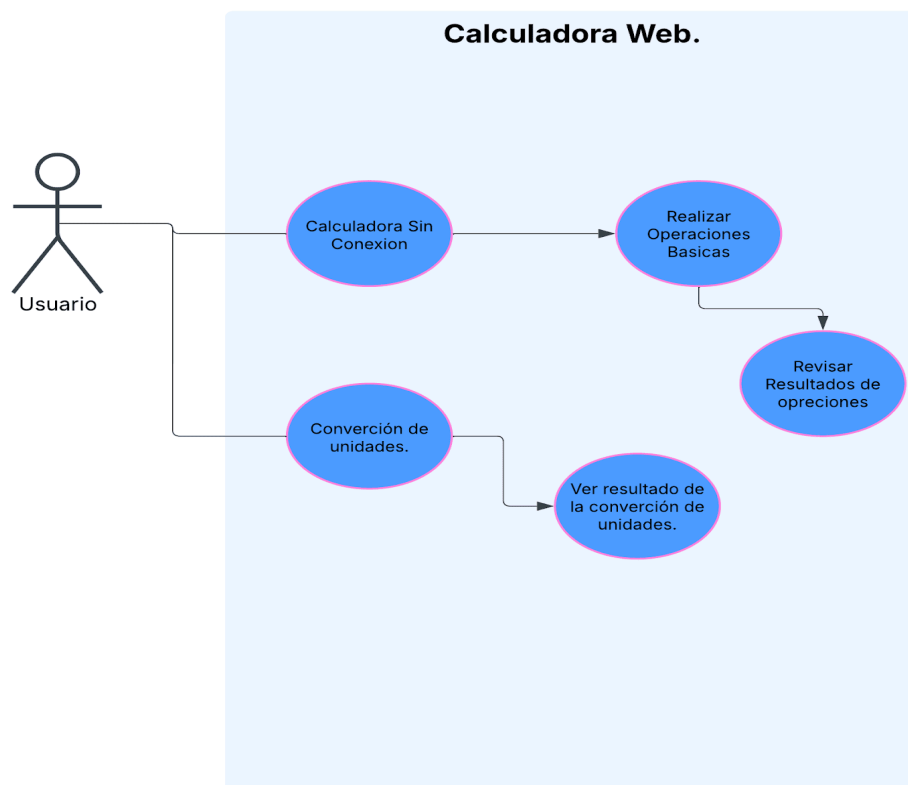
4. El Usuario ingresa los números.
5. El Usuario selecciona una operación que puede causar un error (por ejemplo, la división por cero).
6. El sistema detecta el error y muestra un mensaje adecuado al Usuario (por ejemplo, "Error: No se puede dividir entre cero").

Calculadora sin conexión.

Nombre: "Sin Conexión"

Actor: Usuario.

Diagrama detallado:



Descripción: Que el usuario tenga accesos ciertas funciones sin estar conectado a internet, siempre dejando en claro las limitaciones de hacer uso de esta función, así como poder consultar los datos previamente cargados cuando se estaba conectado a un red de internet.

Precondiciones: Contar con un dispositivo compatible, sea computadora o celular.

Flujos de eventos normales: Que pueda hacer uso de las operaciones básicas y consultar datos previamente cargados pero sin poder utilizar las funciones que requieran conexión a internet.

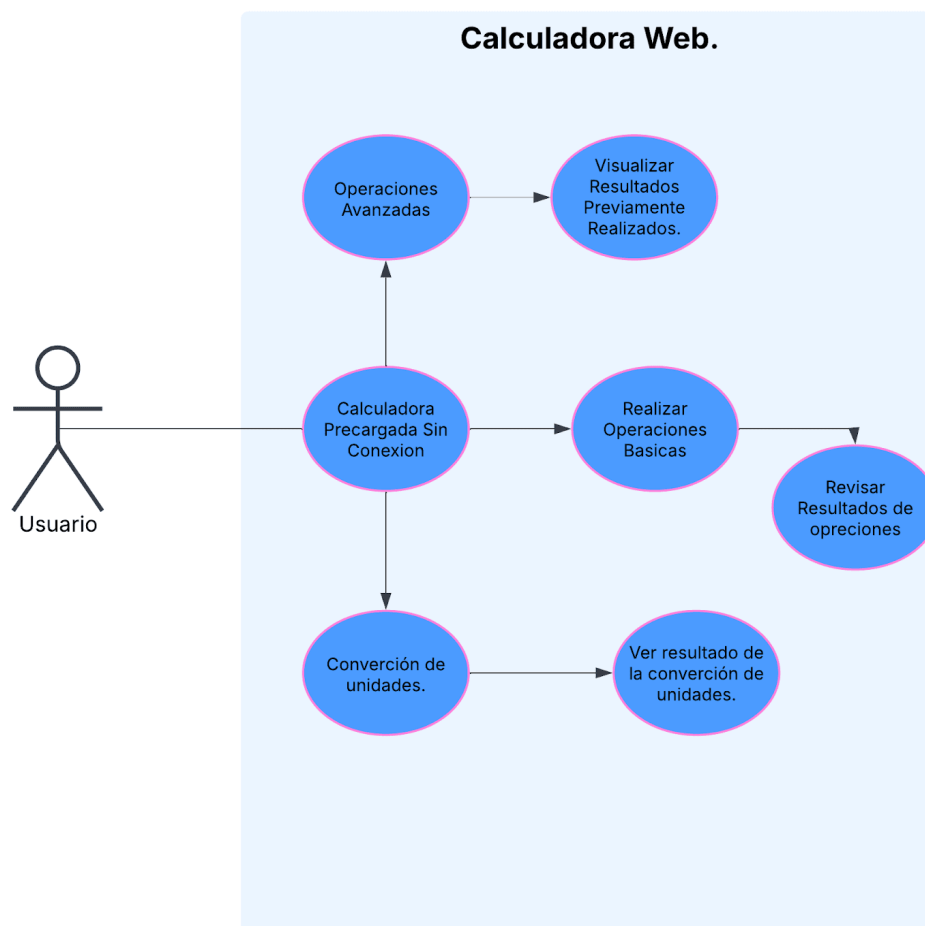
Flujos de eventos alternativos: Que no se puedan ocupar las funciones sin conexión o poder consultar los datos previamente cargados, que la aplicación no funcione o cargue correctamente.

Calculadora previamente cargada.

Nombre: "Previamente cargado"

Actor: Usuario

Diagrama detallado:



Descripción: EL usuario podrá consultar operaciones o resultados previamente realizadas o cargadas mientras se tenía conexión con el servidor, poder cargar la interfaz pero sin poder realizar las operaciones hasta que se conecte a internet, poder usar las operaciones básicas sin dificultades.

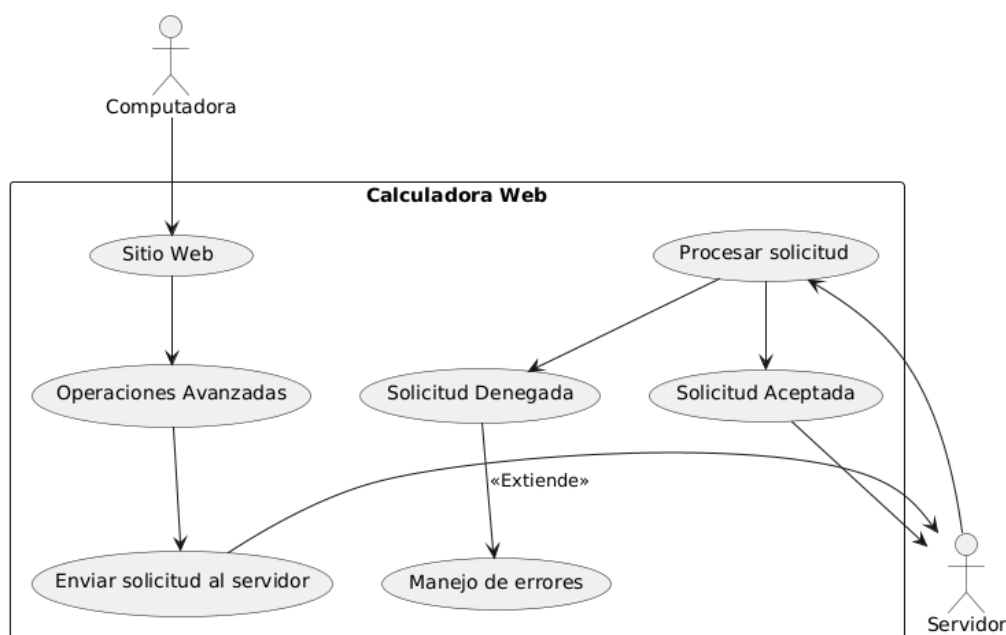
Precondiciones: Tener un dispositivo compatible con la aplicación, además de tener conexión a internet para precargar los datos.

Flujos de eventos normales: El usuario puede consultar resultados de operaciones avanzadas y conversiones realizadas cuando se tenía conexión, además de poder utilizar las operaciones básicas sin problema alguno y a su vez ver el historial de las acciones realizadas.

Flujos de eventos alternativos: El usuario no podrá hacer visualizaciones de los resultados realizados en inversiones u operaciones avanzadas cuando se hicieron durante la conexión a internet o en su defecto que no pueda hacer uso de ninguna función de la calculadora.

5.2.2 Elementos de diseño (computadora a servidor “aaron” y servidor a computadora “erick” casos de uso con y sin internet)

Análisis Detallado del Diagrama de Casos de Uso



1. Actores

Computadora: Representa el dispositivo del usuario que accede a la calculadora web.

Servidor: Representa el backend que procesa las solicitudes enviadas desde la calculadora web.

2. Casos de Uso

Sitio Web

Descripción: El usuario accede al sitio web de la calculadora desde su computadora.

Relaciones:

La Computadora inicia la interacción con el sistema accediendo al Sitio Web.
El Sitio Web redirige al usuario hacia la funcionalidad de Operaciones Avanzadas.

Operaciones Avanzadas

Descripción: El usuario selecciona una operación matemática avanzada (ejemplo: funciones trigonométricas, logaritmos, exponentes).

Relaciones:

Extiende el flujo de interacción con la calculadora web al permitir realizar operaciones más complejas. Una vez seleccionada la operación, el sistema debe enviar la solicitud al servidor para su procesamiento.

Enviar Solicitud al Servidor

Descripción: La calculadora envía la solicitud de operación avanzada al servidor.

Relaciones:

Depende del paso anterior (Operaciones Avanzadas) porque la solicitud solo puede enviarse después de que el usuario seleccione una operación. El Servidor recibe esta solicitud y la pasa al módulo de procesamiento.

Procesar Solicitud

Descripción: El servidor procesa la solicitud enviada por la calculadora web.

Relaciones:

Es activado por el caso de uso Enviar Solicitud al Servidor. Aquí se validan los datos, se calculan los valores y se genera la respuesta que se enviará de vuelta al usuario. Puede terminar en dos caminos: Solicitud Aceptada o Solicitud Denegada.

Solicitud Aceptada

Descripción: La solicitud se procesa correctamente y se genera un resultado válido.

Relaciones:

Está conectada con el caso de uso Procesar Solicitud, lo que significa que ocurre si la operación matemática es válida.

Una vez aceptada, el resultado se envía de vuelta al servidor para su transmisión a la computadora del usuario.

Solicitud Denegada

Descripción: El servidor detecta un problema en la operación (como una división entre cero) y rechaza la solicitud.

Relaciones:

Extiende el caso de uso Procesar Solicitud, ya que solo se activa si hay un error en la operación. Conduce al manejo de errores para notificar al usuario.

Manejo de Errores

Descripción: Se activa si la solicitud es denegada debido a una operación inválida.

Relaciones:

Es activado por Solicitud Denegada cuando ocurre un error en el cálculo. Envía un mensaje a la computadora informando al usuario del problema (por ejemplo, "Error: No se puede dividir entre cero").

Resumen del Flujo

1. Computadora accede al Sitio Web de la calculadora.
2. Usuario selecciona una operación avanzada.
3. El sistema envía la solicitud al Servidor.
4. El Servidor procesa la solicitud.

Si la solicitud es válida, se acepta y se genera un resultado.

Si hay un error, la solicitud es denegada y se activa el manejo de errores.

5. El resultado (éxito o error) se envía de vuelta al usuario.

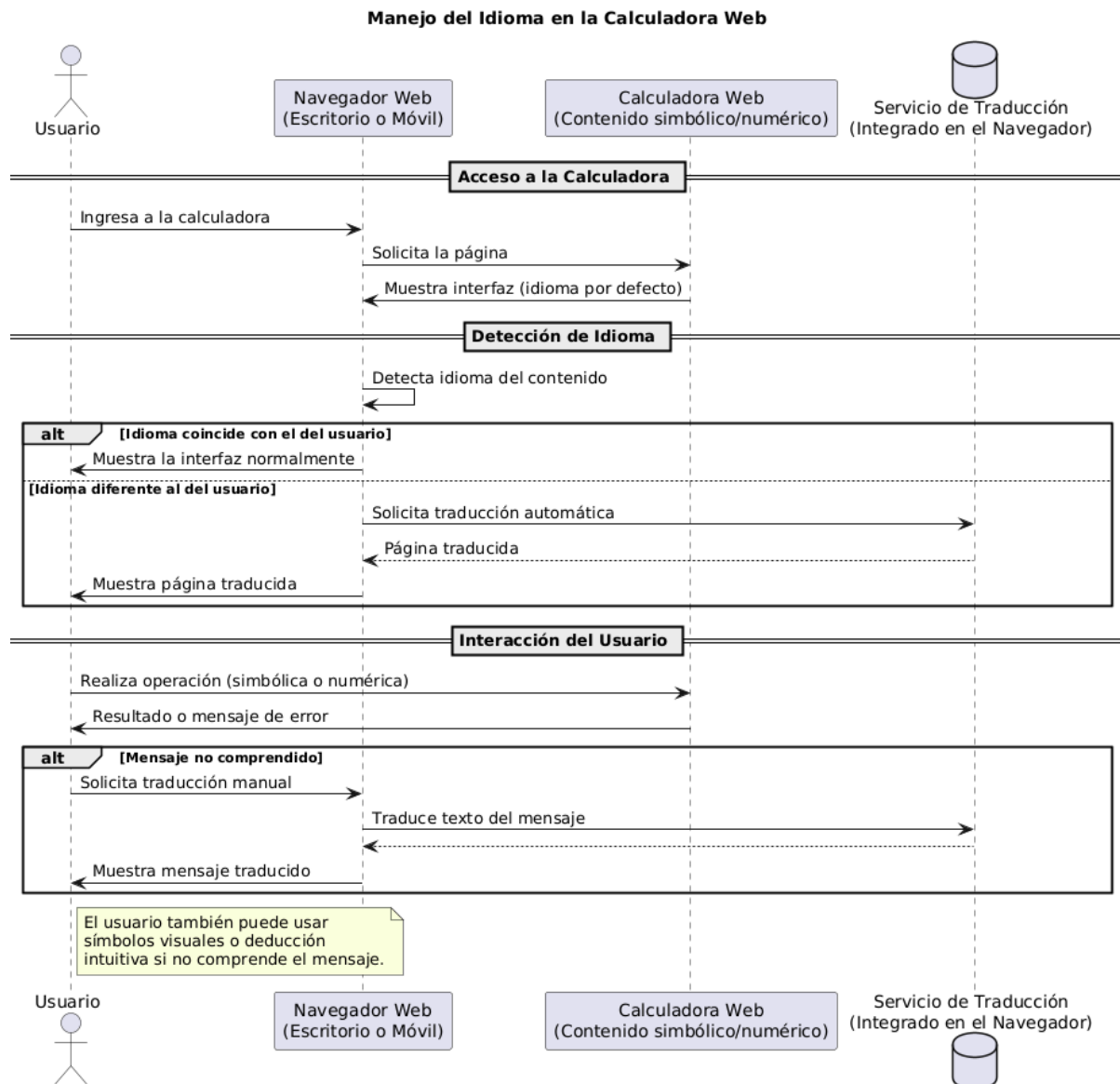
5.2.3 Ejemplos de idiomas

Selección del idioma

nombre: Idiomas

Actor: Usuario, idioma del navegador por defecto y dispositivo.

Diagrama detallado:



Descripción:

Actores involucrados

- Usuario: Al ser una calculadora no hay mucho vocabulario involucrado, es puro numérico y simbólico, en el caso de que no entienda el poco lenguaje que se llegue a ocupar en las ayudas o ventanas de error y correcciones se realizará la siguiente acción por parte del usuario.

Buscar una manera de traducir la página de ser necesario buscar en diferentes fuentes el cómo hacer una traducción de manera nativa desde su navegador móvil o de escritorio.

Reconsiderar si es necesario la traducción de la página, se resalta el hecho de que no se maneja mucho el lenguaje todo será mediante números y símbolos.

En caso de que sea necesario para el usuario se realizará lo siguiente:

- El usuario ingresa a la página mediante una computadora, al realizar una operación avanzada o básica el usuario le marca un error o busca realizar una operación en específico el idioma en el que se encuentra el navegador no lo entiende, puede guiarse mediante el sistema de símbolos para realizar la operación que necesita, en caso de error revisar intuitivamente su error.
- Puede buscar una traducción nativa del navegador, en muchos casos el mismo navegador hace la traducción automática, el idioma será seleccionado automáticamente por el navegador en este caso.
- Navegador: En este caso se propone el navegador por defecto detecta el idioma diferente y hace una traducción automática presentando al usuario el idioma de la región del usuario.
- Dispositivo: EL idioma del navegador está ligado por defecto al idioma nativo seleccionado por el usuario detectando la región en la que se encuentra el usuario esto también puede cambiar si el usuario configuró en un idioma diferente su dispositivo .

Precondiciones: Las condiciones más aptas para en cambio de idioma se recomiendan las acciones de saber cómo opera el navegador y hacer una traducción nativa del navegador para realizar esto se necesita una conexión estable con su red, todo eso se aplica de ser necesario esto será decidido por el usuario.

Flujo de eventos normales: El cambio de idioma se realiza de una forma eficiente por medio de la red del navegador, el idioma se detecta directamente del navegador y el usuario no necesita hacer una intervención para el entendimiento de esta página.

5.3 Propósito del Punto de Vista de Composición

Este punto de vista descompone el sistema en módulos, componentes y subsistemas, definiendo sus responsabilidades y relaciones. Su objetivo es:

- Facilitar y permitir el desarrollo paralelo (frontend/backend).
- Identificar oportunidades de reutilización (ej: librerías compartidas).
- Establecer contratos de interfaz entre componentes.
- Facilitar el mantenimiento y comprensión del sistema

Audiencia principal:

Arquitectos de software: Para evaluar cohesión y acoplamiento. Esto ayudará para futuras mejoras en el producto ya que tendrían en esencia el comportamiento de todo el producto. (manteneabilidad, testeabilidad y escalabilidad).

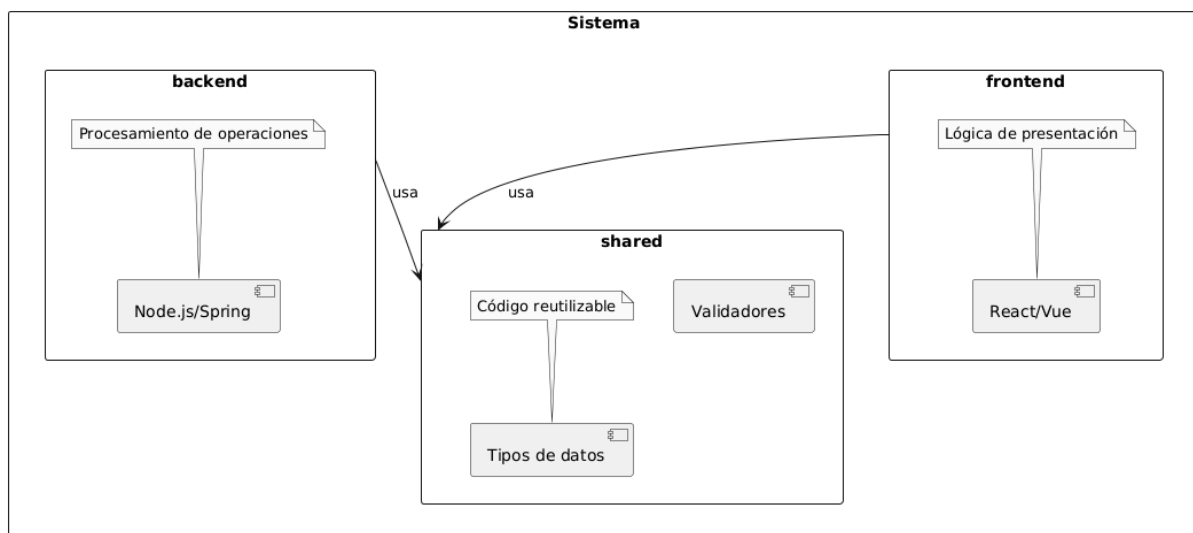
Líderes de equipo: Para asignar módulos a desarrolladores: para distribuir tareas y responsabilidades. Así poner un cronograma y partes correspondientes a cada equipo de ser necesario por el alcance del proyecto de ser necesario contratar un tercero para partes necesarias del producto.

Desarrolladores: Esto para que ellos tengan presentes el alcance de cada módulo y así mejorando el entendimiento y límites de cada equipo aliviando cargas de trabajo correspondiente.

Equipo de testers: Poner a prueba los módulos correspondientes para así diseñar las pruebas correspondientes de cada módulo ya sea por sección o todo en conjunto

Estructura Modular del Sistema

Diagrama de Paquetes (UML)



Descripción:

frontend: Contiene lógica de presentación. La principal encargada de mostrar errores ante el usuario y la estructura interna del producto también mostrará las entradas correspondientes al usuario, el uso de tecnologías responsables serán HTML y JavaScript

backend: Procesamiento de operaciones. Estas serán las operaciones básicas y la conexión con el servidor para las operaciones avanzadas es la interacción directa con el servidor.

share: Código reutilizable (ej: validadores, tipos de datos) este utilizara y tendrá contacto tanto con el front como con el back.

Componentes Principales

Frontend

Interfaz de Usuario: Encargada de representar visualmente la calculadora y sus botones. Contiene toda la interacción entre el usuario y la computadora. Se encargará de mostrar guías o mensajes sobre los botones desconocidos para el usuario ya sea en cálculos avanzados o básicos.

Gestor de Eventos: Captura las interacciones del usuario ya sea registro de números con el click o el teclado y lanza las acciones correspondientes. Muestra tanto errores como opciones para que el usuario pueda comprender más fácil el entorno.

Almacenamiento Local: Guarda historial de operaciones cuando no hay conexión con un límite de guardado del historial. Esto será con un aprox de 25 operaciones realizadas, ya con una conexión a internet se tratará de ajustar el límite unas 50 operaciones aprox.

Servicio de comunicación API: Encapsular la lógica para realizar llamadas HTTP/S al backend (ej: enviar operaciones para cálculo, solicitar historial). Manejar la serialización/deserialización de datos.

Backend

API REST: Expone endpoints (cualquier dispositivo que se conecta a una red para acceder a información o recursos) para enviar y recibir datos de operaciones esto conlleva la validación de datos ya sean de datos o escrito.

Servicio de Cálculo: Procesa operaciones avanzadas utilizando librerías matemáticas. Ya sea para cálculos avanzados o básicos. Esto conlleva a utilizar y aplicar la razón y lógica de las operaciones con esto hace el llamado, adonde a las librerías necesarias para cumplir con lo solicitado.

Middleware de Seguridad: Asegura la validación de solicitudes, como autenticación para historial o las solicitudes de computadora servidor evitando así ataques comunes hacia el producto o datos del servidor.

Capa de persistencia: interactúa con la base de datos para el historial ya sea de manera local o con el servidor esto también guarda sugerencias de operaciones anteriores

Frontend

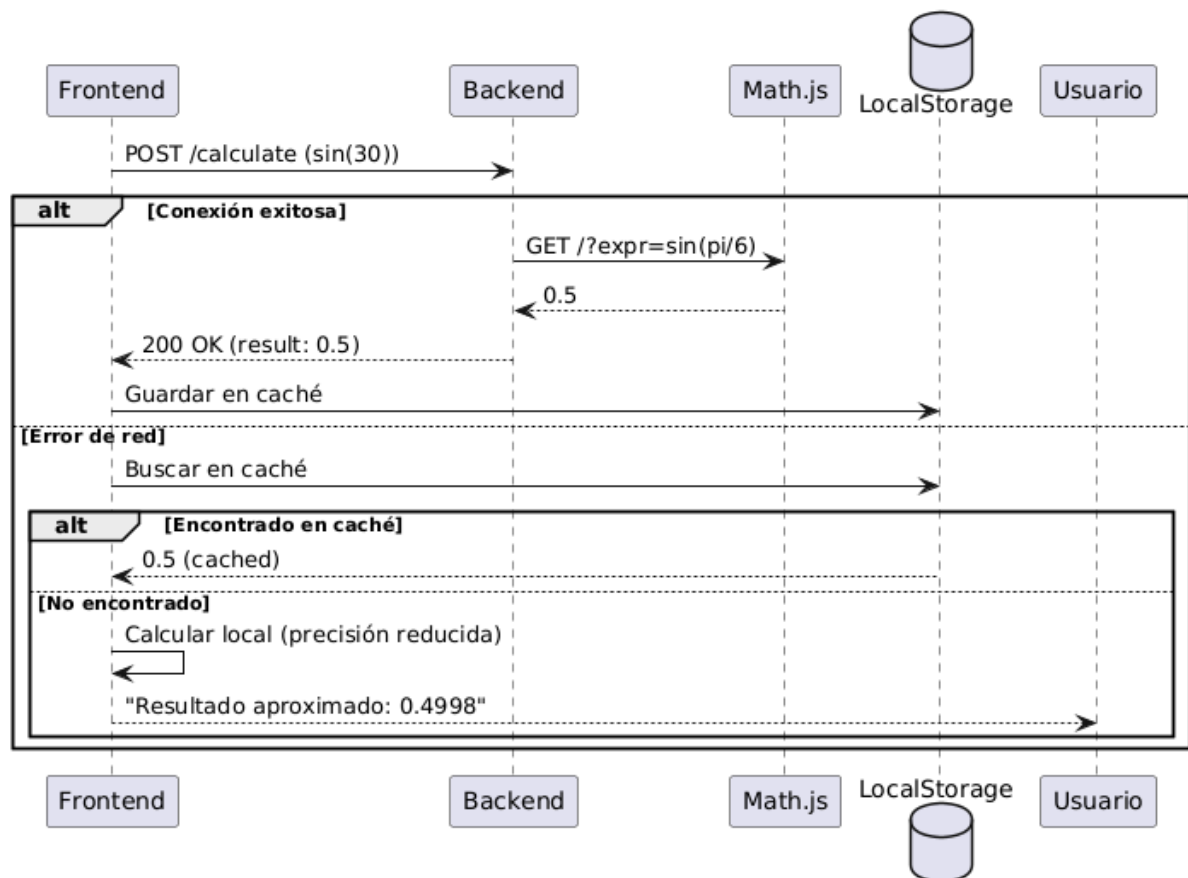
Componente	Responsabilidad	Tecnología	Dependencias
Calculadora UI	Renderizar botones/pantalla.	React	Gestor Eventos

Gestor Eventos	Manejar clics/teclas → disparar cálculos.	JavaScript	Servicio API (backend)
LocalStorage	Cachear resultados offline.	Index	-

Backend

Componente	Responsabilidad	Protocolo
API REST	Exponer endpoints (POST /calculate).	Express.js
ServicioCalculo	Delegar operaciones a Math.js.	HTTP/HTTPS
AuthMiddleware	Validar tokens JWT para historial.	-

Diagrama



Reutilización de Componentes

El reuso de componentes para una mejor optimización de procesos y una mejora para el usuario, referente a que ayuda a la mantenibilidad, ayudando también al desarrollador en futuras entregas o correcto de errores:

Librerías matemáticas (Math.js): Utilizada tanto en el frontend (para validaciones rápidas esto en cuestión de operaciones básicas) como en el backend (esto para operaciones avanzadas esto como unico metodo de lógica verdadera para los resultados) para mantener la consistencia en los resultados. Al ya estar previamente cargados su utilidad es recursiva ayuda a la optimización de procesos.

Funciones de validación: Aplicadas para asegurar que los inputs del usuario sean válidos antes de ejecutar operaciones esto delimitará los errores o los reduce en un gran porcentaje. El uso de estas es inmediato en el front y en el back como una segunda capa de seguridad y validación antes de su procesamiento. Esta reutilización mejora la mantenibilidad y reduce la duplicación de lógica.

Modelos de Datos/Tipos (del paquete share):

Ejemplos: `OperationPayload { operand1: number, operand2: number, operator: string },`
`HistoryEntry { expression: string, result: number, timestamp: Date }.`

Usados por front y back para asegurar consistencia en la estructura de datos intercambiada.

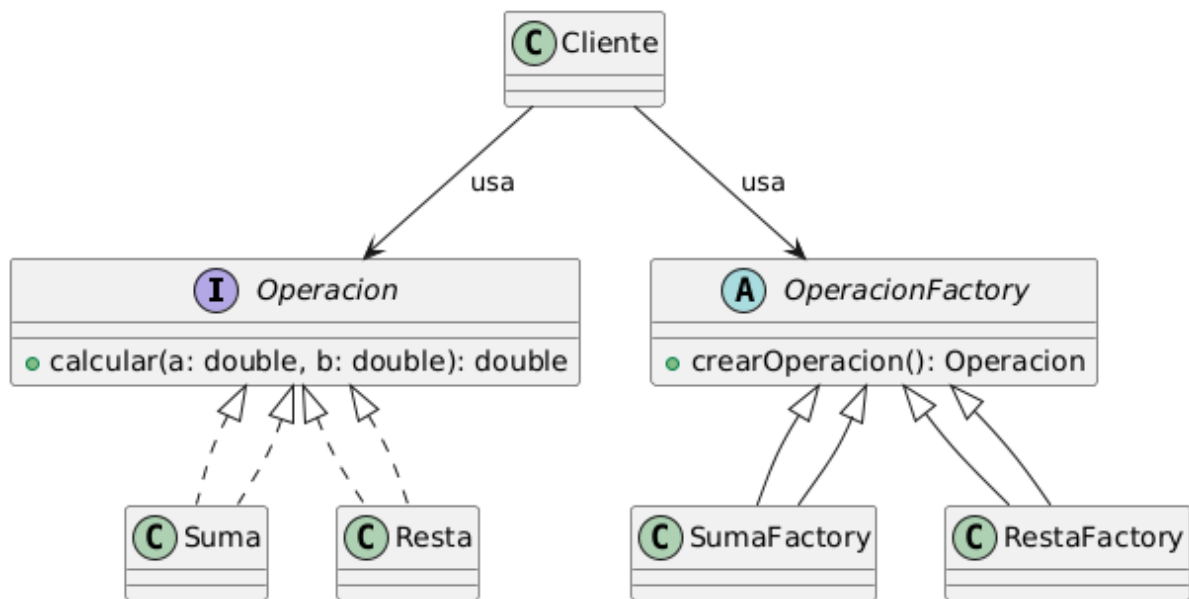
Librerías Compartidas

- ❖ Math.js: Usada en frontend (operaciones básicas y validaciones) y backend (avanzadas y como aseveración lógica). Configuración y utilidad a través de javascript y HTML
- ❖ De ser necesario añadir para futuras entregas más librerías encargadas de diferentes herramientas un ejemplo si el producto a lo largo de su realización necesita la hora o como futura referencia de problemas relacionados con el horario se añadirá la librería necesaria.

Patrones de Diseño Aplicados

Factory Method: Para instanciar operaciones (patrón de diseño creacional que define una interfaz para crear objetos, pero permite a las subclasses decidir qué clase concretas se crearán.)

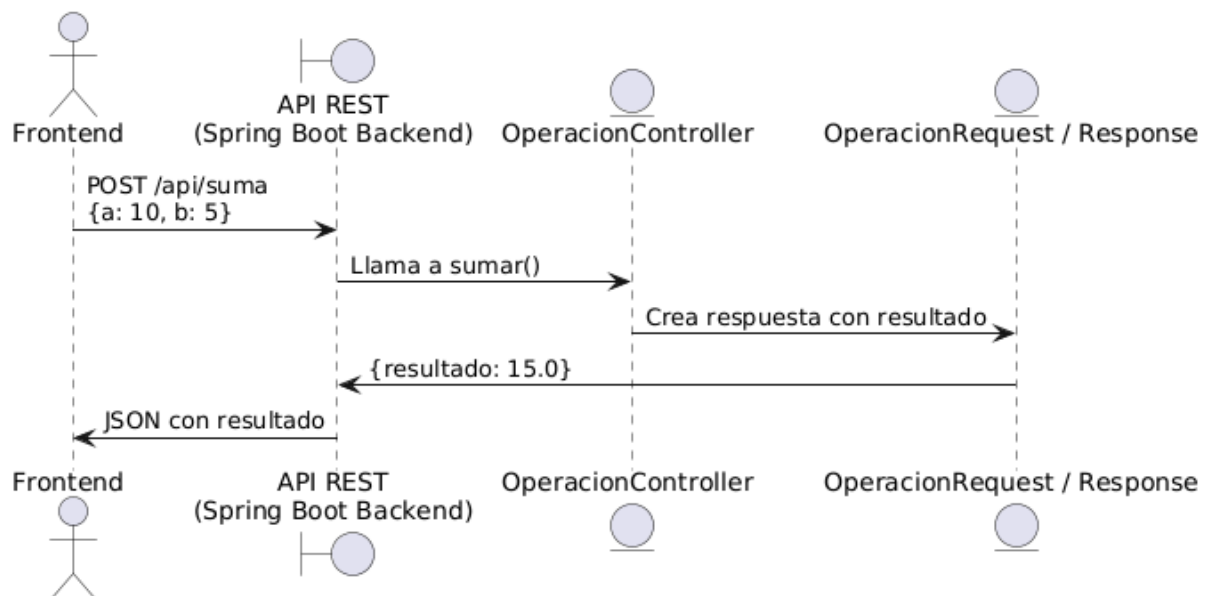
java



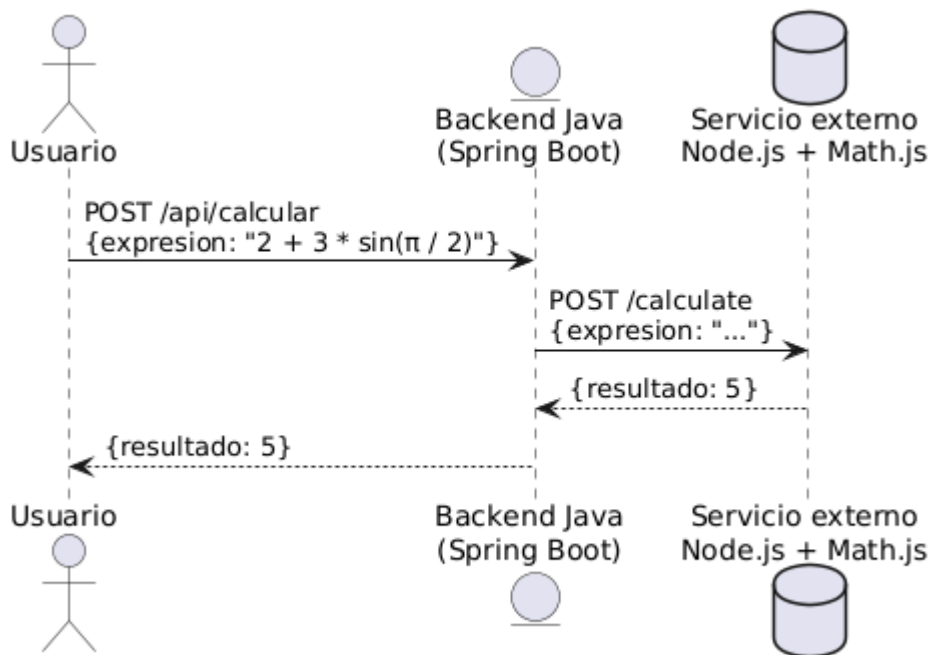
Interfaces Críticas

Las interfaces clave que conectan los componentes son:

Frontend ↔ Backend: A través de una API REST (Es una forma común de que los sistemas informáticos se comuniquen entre sí a través de internet, especialmente en arquitecturas cliente-servidor y de microservicios.), usando JSON. Se define una conexión clara para cada operación matemática.



Backend ↔ Librería Math.js o servicios externos: Comunicación directa para ejecutar cálculos complejos con precisión. Reutilizada junto con otras librerías en el servidor.



Frontend ↔ Backend

Contrato API REST:

Método	Endpoint	Cuerpo de solicitud	Resultado escrito
POST	/api/calcutate	{"op": "sqrt", "val": 16}	{"result": 4}
GET	/api/history	Headers: Authorization: Bearer <JWT>	["sqrt(16)", "sin(30)"]

Manejo de error de acuerdo al error y a los códigos de HTML se proporcionarán diferentes códigos para saber mejor el error y su correspondientes factores y forma de solución tanto para el usuario como para el desarrollador.

Despliegue Físico

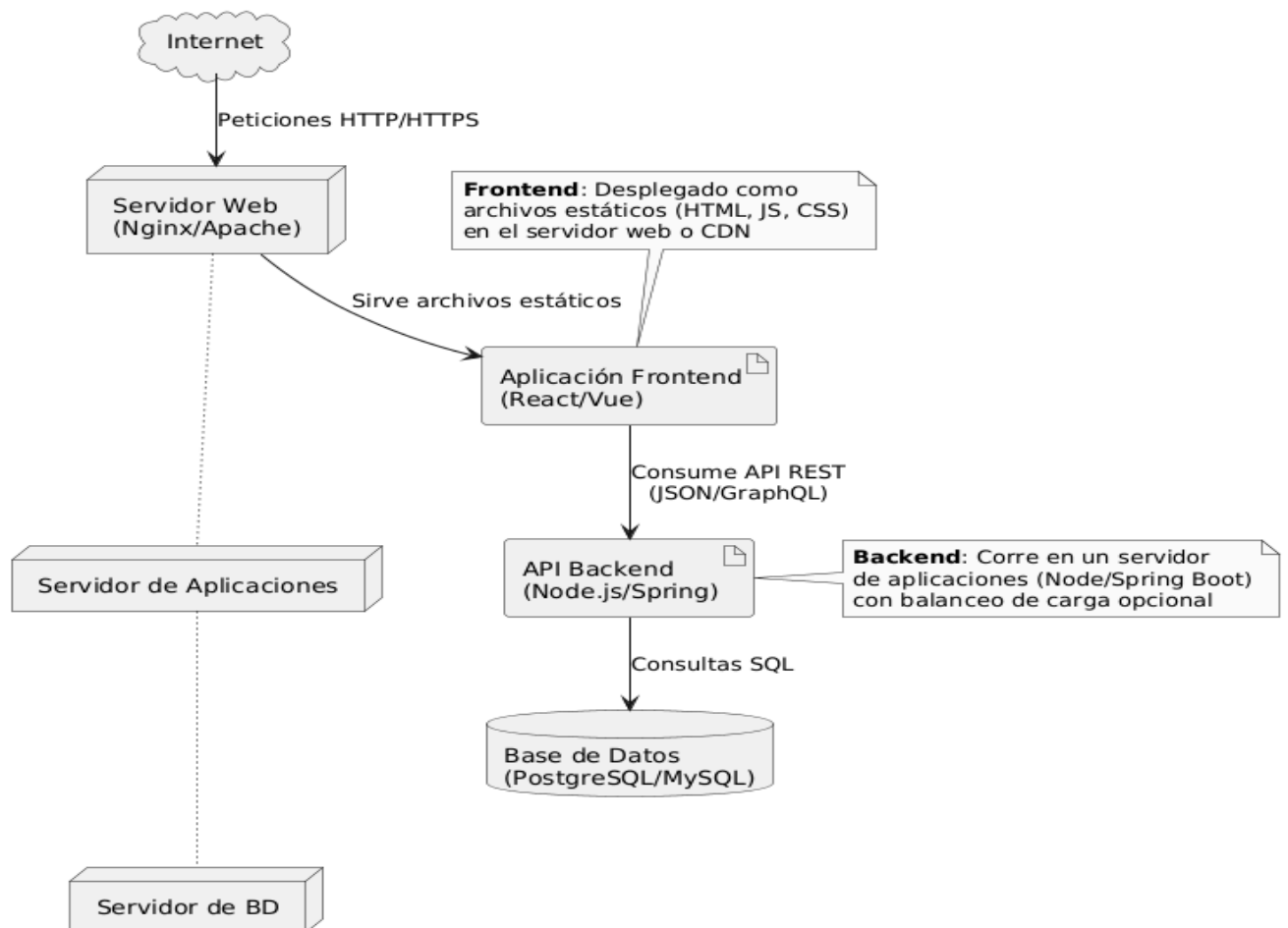
Requisitos por componente:

Frontend: Ligero, debe correr en cualquier navegador moderno. Al igual que en cualquier versión disponible que sea compatible ya sea en dispositivo móvil o de escritorio y que soporte la infraestructura del producto ya sea servicios de JavaScript o HTML

Backend: Debe poder escalar horizontalmente si aumenta el número de solicitudes. No tener errores de lógica o de conexión con el servidor evitando falsas respuestas y desconexiones de respuesta del servidor, interacción fluida con el servidor para evitar saturación de múltiples solicitudes de los usuarios.

Persistencia: Aplica almacenamiento local para modo offline y base de datos externa para modo conectado. En cuestión de almacenamiento local sin línea de red solo se almacenarán 25 operaciones.

Diagrama de Despliegue



Requisitos de Infraestructura

Componente	CPU	Memoria	Almacenamiento
Frontend	1 vCPU	512 MB	1 GB (CDN)
Backend	2 vCPUs	4 GB	10 GB (Docker)

Si el usuario va a ser una variedad pesada de operaciones se recomienda tener una red de banda ancha para la velocidad de procesamiento sin sufrir una desconexión del servidor por respuesta tardía.

Estrategia de Pruebas por Componente

Cada componente será probado de forma independiente:

Frontend: Pruebas visuales y de interacción (UI). Al igual sus dimensiones para que todo sea visible para el usuario sin necesidad de modificar la vista desde el zoom del navegador así proporcionando una vista general de todas las funciones principales.

De ser necesario realizar las pruebas necesarias para la lógica y las bibliotecas, retroceso y acciones del usuario.

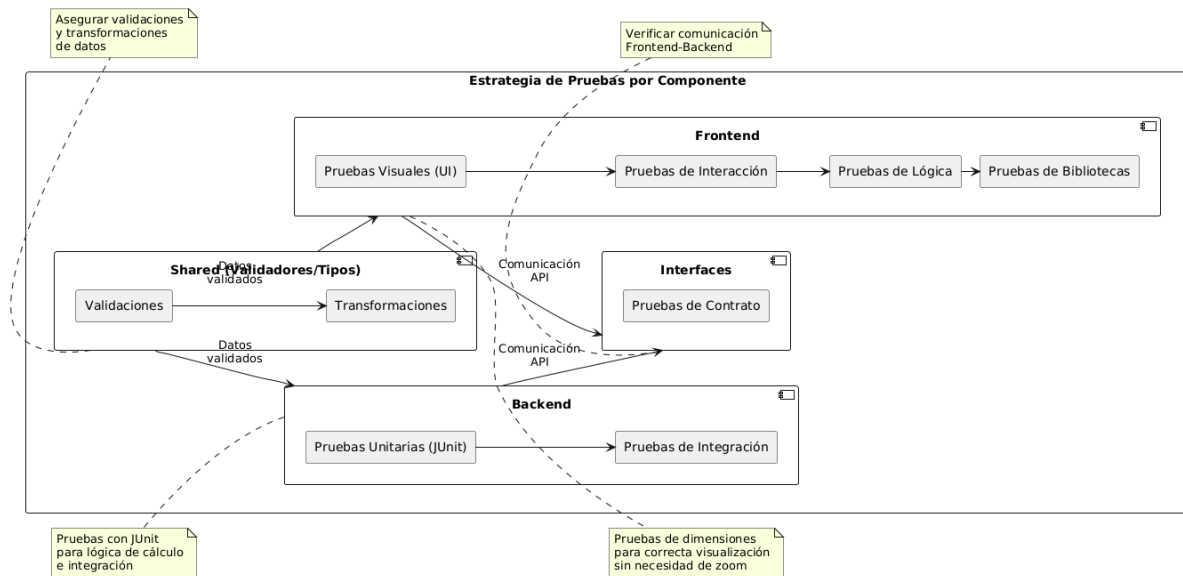
Backend: Pruebas unitarias e integración para lógica de cálculo. Las pruebas se realizarán con la integración de otros métodos y controladores pruebas realizadas con JUnit.

Share Validadores/Tipos: Pruebas unitarias para asegurar que las validaciones y transformaciones de datos funcionan como se espera.

Interfaces: Pruebas de contrato para asegurar que el frontend y backend se comuniquen correctamente.

Componente	Tipo de Prueba	Herramienta	Criterio de Éxito
CalculadoraUI	Pruebas de snapshot	JUnit	Renderizado coherente con diseño.
ServicioCalculo	Pruebas de integración	Mocha/Chai/ JUnit	100% cobertura de operaciones.

Diagrama Pruebas Por Componentes



Evolución Arquitectónica

Las versiones que contienen después del primer lanzamiento del producto se darán para dar paso a un mejoramiento, esto se tomará de la siguiente forma.

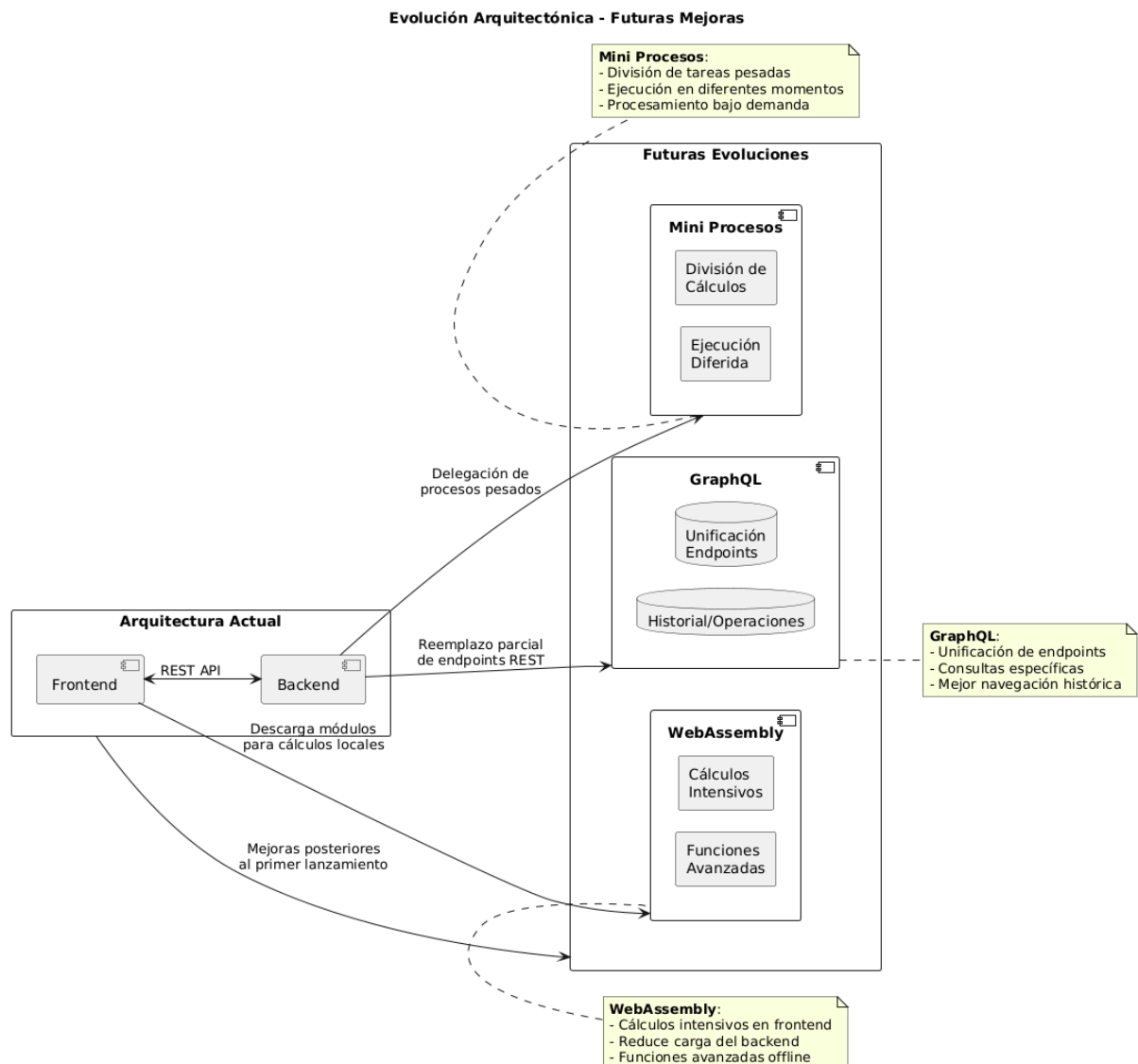
Futuro posible:

WebAssembly: Para cálculos intensivos en frontend. Esto ayudará a reducir la carga del back en el procesamiento de los datos posiblemente se agregaran una parte de las funciones avanzadas sin necesidad del servidor.

GraphQL: Unificar endpoints de historial/operaciones. Mejora la navegación en el historial del usuario así dándole datos más específicos de ser necesarios modificando el procedimiento o mirarlo directamente para consultarlo.

Mini procesos: Si la tarea del cálculo es muy pesada podrían dividirse los cálculos o hacerlos en diferentes momentos a petición del usuario para que el proceso sea el deseado.

Diagrama Evaluacion de Arquitectura



5.4 Punto de Vista Lógico

Propósito del Punto de Vista Lógico

Este punto de vista se enfoca en las abstracciones clave del sistema, representando las entidades fundamentales, sus atributos, operaciones y relaciones. Su objetivo es:

Definir la estructura de clases y objetos que componen la calculadora.

Establecer jerarquías de herencia para promover la reutilización de código.

Documentar los principios de diseño (SOLID, DRY) aplicados.

Mejorar en la mantenibilidad y escalabilidad del sistema.

Audiencia principal:

Desarrolladores: Para implementar las clases y métodos con precisión.

Equipo de pruebas: Para diseñar casos de prueba unitarios basados en contratos de métodos.

Diagrama de Clases Principal

El sistema se basa en una jerarquía de clases que representa las operaciones, el historial y la calculadora como orquestadora.

Entidades clave:

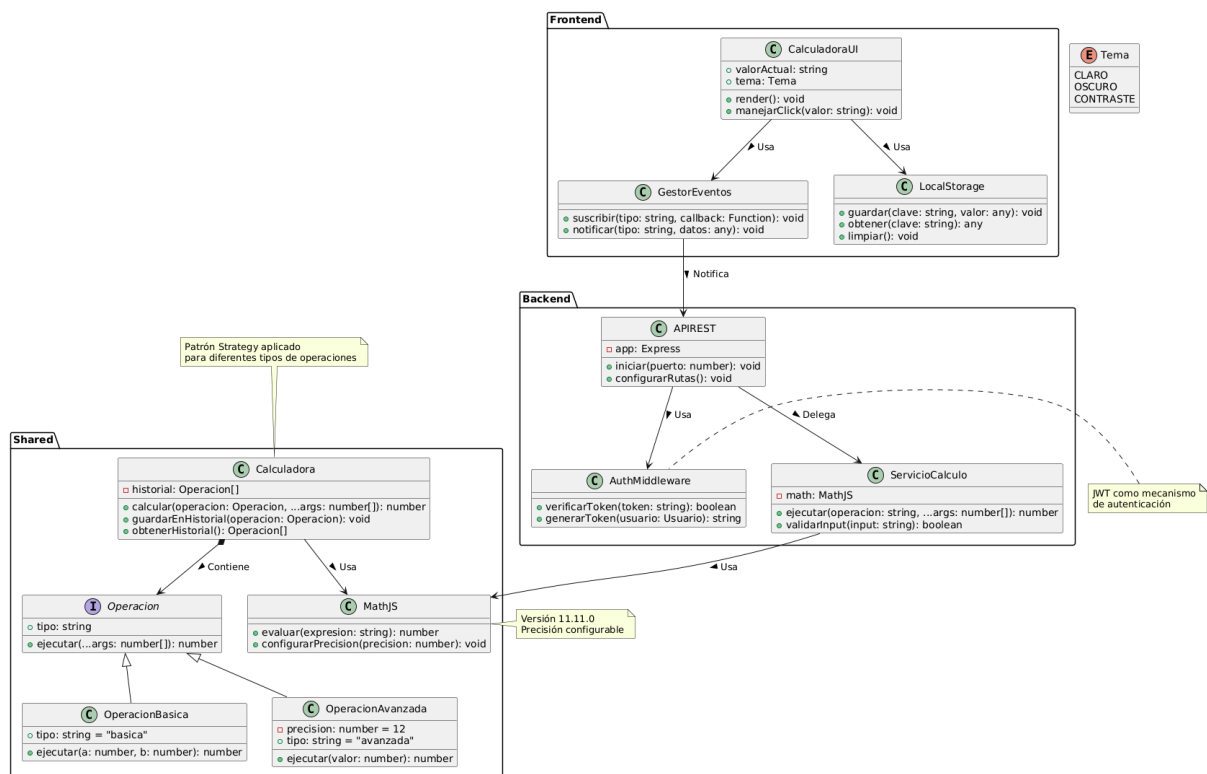
Operación: Clase abstracta base que define la estructura común de toda operación matemática.

Operación Básica: Hereda de Operación; implementa operaciones como suma, resta, multiplicación y división.

Operación Avanzada: Hereda de Operación; maneja cálculos más complejos como trigonometría, logaritmos y exponenciales.

Calculadora: Administra la ejecución de operaciones y el almacenamiento del historial.

Cada clase cuenta con sus propios atributos y métodos específicos para cumplir con su responsabilidad.



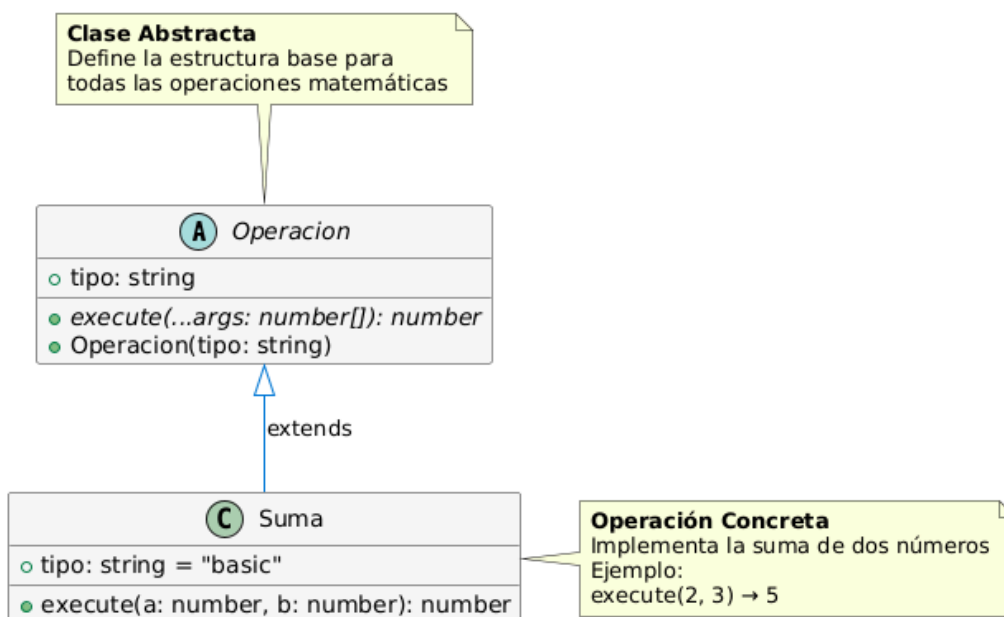
Descripción de Clases

Clase	Responsabilidad	Atributos/Métodos Clave
Operación	Clase abstracta para todas las operaciones.	tipo: String (basic, trig), execute()
Operación Básica	Implementa suma/resta/multiplicación/división	execute(a, b) → a + b
Operación Avanzada	Maneja trigonometría, logaritmos, etc.	precision: Number, execute(valor) → sin(valor)
Calculadora	Orquesta operaciones y gestiona historial.	calcular(op, valor), guardar en Historial()

Jerarquía de Herencia Detallada

La jerarquía de herencia permite organizar las operaciones según su complejidad y reutilizar código común. Se establece de la siguiente manera:

Operación es una clase abstracta general. Operación Básica y Operación Avanzada son clases hijas que implementan el método de ejecución de forma especializada. Nuevas operaciones (como raíces, potencias, factoriales, etc.) pueden añadirse fácilmente extendiendo esta jerarquía sin modificar clases existentes.



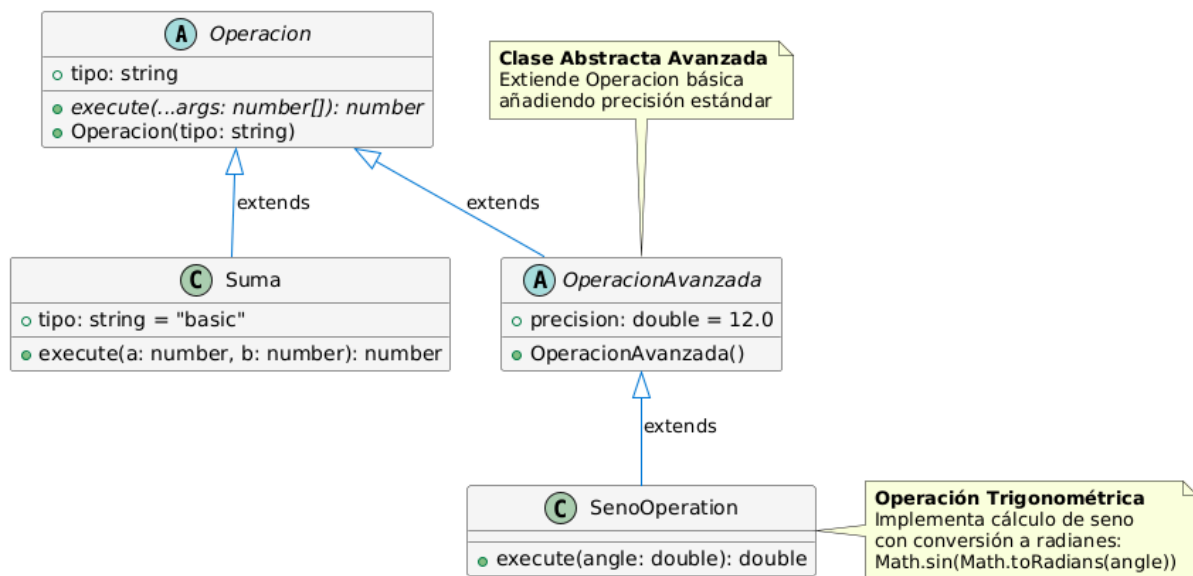
Operaciones Avanzadas

// Ejemplo en Java

```
public abstract class OperacionAvanzada extends Operacion {  
    protected double precision = 12.0;
```

```
    public OperacionAvanzada() {  
        super("advanced");  
    }  
}
```

```
public class SenoOperation extends OperacionAvanzada {  
    @Override  
    public double execute(double angle) {  
        return Math.sin(Math.toRadians(angle));  
    }  
}
```



Patrones de Diseño Aplicados

Se aplican los siguientes patrones para mejorar la flexibilidad y robustez del diseño:

Strategy: Permite cambiar dinámicamente la lógica de cálculo según el tipo de operación seleccionada por el usuario.

Factory Method: Facilita la creación de instancias de operaciones a través de fábricas especializadas.

Singleton: Aplicado al historial para garantizar una única instancia compartida por todo el sistema.

Estos patrones aseguran bajo acoplamiento y alta cohesión entre clases.

Strategy Pattern

Contexto: Seleccionar algoritmos de cálculo en runtime.

Implementación:

python

Ejemplo en Python

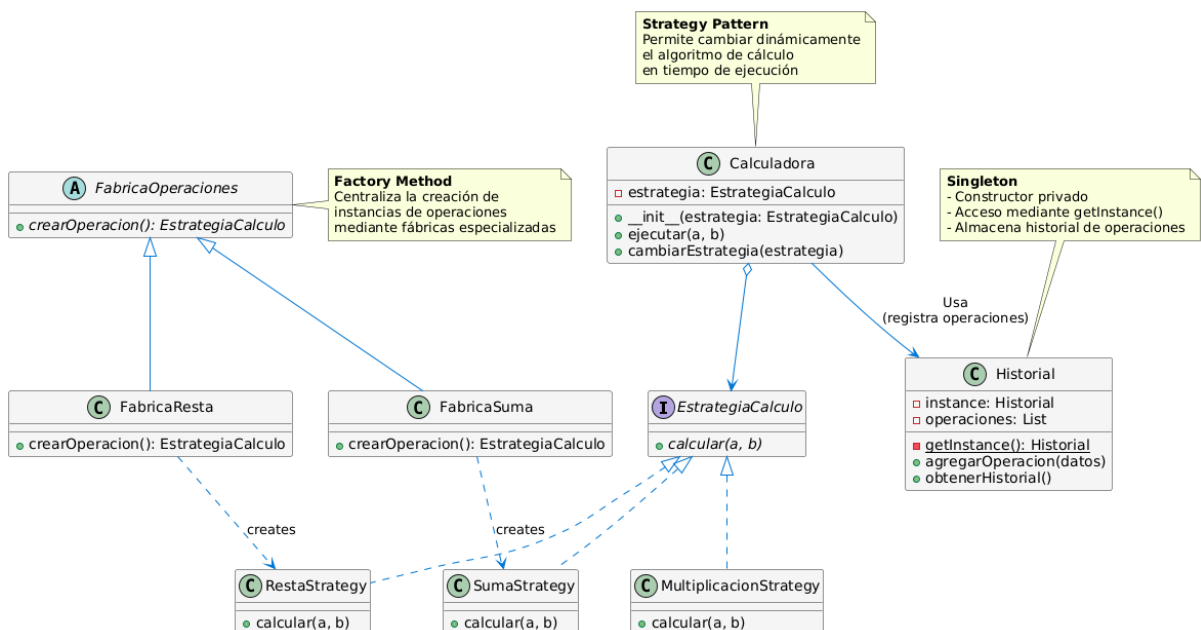
```
class EstrategiaCalculo:
    def calcular(self, a, b):
        pass
```

```
class SumaStrategy(EstrategiaCalculo):
    def calcular(self, a, b):
        return a + b
```

```
class Calculadora:
    def __init__(self, estrategia: EstrategiaCalculo):
        self.estrategia = estrategia

    def ejecutar(self, a, b):
        return self.estrategia.calcular(a, b)
```

Diagrama



Singleton (Para el Historial)

javascript

// Ejemplo en JavaScript

```
class Historial {
    static instancia;
```

```

constructor() {
    if (!Historial.instancia) {
        this.operaciones = [];
        Historial.instancia = this;
    }
    return Historial.instancia;
}

agregar(op) {
    this.operaciones.push(op);
}
}

```

Contratos de Métodos Críticos

Para asegurar la correcta ejecución de las operaciones, se establecen contratos con:

Parámetros esperados.

Restricciones previas (precondiciones) como no dividir entre cero.

Resultados esperados (postcondiciones) con ejemplos concretos.

Esto ayuda a validar el comportamiento esperado y prevenir errores comunes.

Operaciones Matemáticas

Método	Parámetros	Retorno	Precondiciones	Postcondiciones
execute(a: Number)	a: Ángulo en grados.	Numero	$-360 \leq a \leq 360$	$-1 \leq \text{retorno} \leq 1$ (para seno/coseno)
execute(a: Number, b: Number)	a, b: Operandos.	Numero	$b \neq 0$ (si es división)	$\text{retorno} == a + b$ (ejemplo)

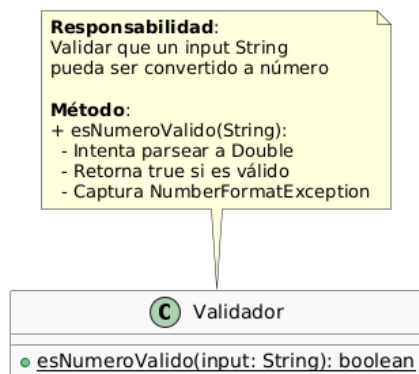
Validación de Input

```

java
public class Validador {
    public static boolean esNumeroValido(String input) {
        try {
            Double.parseDouble(input);
            return true;
        } catch (NumberFormatException e) {
            return false;
        }
    }
}

```

}



Modelado de Estados (Para Operaciones Complejas)

El sistema tiene estados definidos para reflejar la interacción del usuario con la calculadora:

Inactiva: No se ha ingresado ninguna expresión.

Validando: El sistema analiza la entrada.

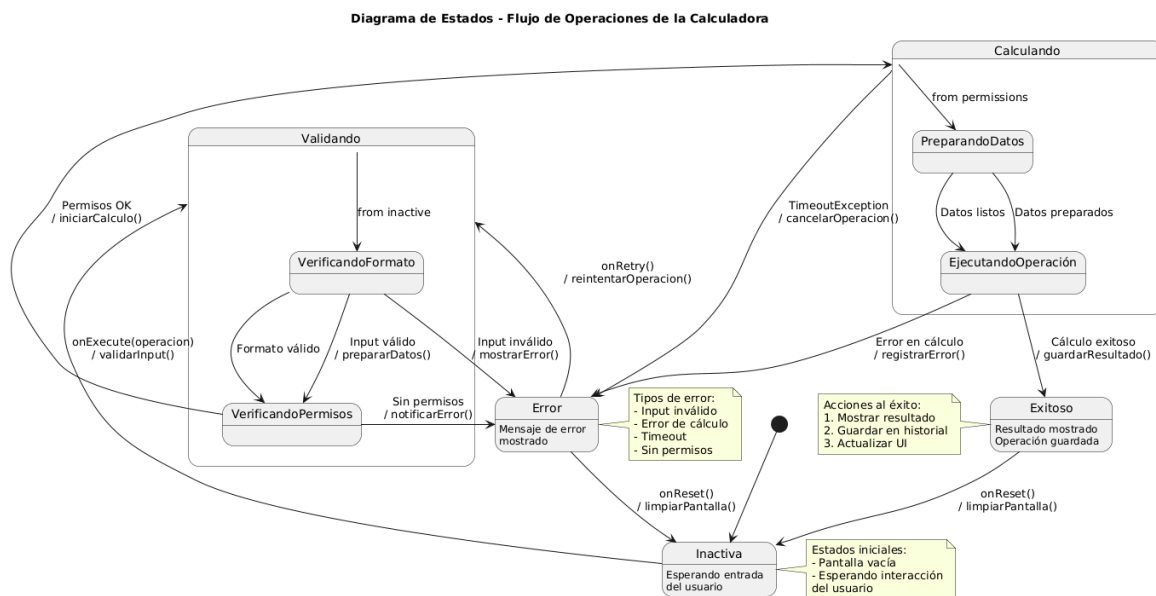
Calculando: Se realiza la operación.

Exitoso: Resultado válido mostrado.

Error: Entrada inválida o problema en el cálculo.

Cada transición entre estados está gobernada por eventos disparados desde la interfaz o el backen.

Diagrama de Estados (UML)



Transiciones de Estado

Estado	Evento	Acción	Nuevo Estado
Inactiva	onExecute(op)	Validar input.	Activa
Activa	onSuccess(result)	Guardar resultado.	Exitoso
Activa	onError(exception)	Mostrar mensaje de error.	Error

Ejemplos de Código por Lenguaje

El diseño lógico es agnóstico al lenguaje, pero la implementación concreta usa:

TypeScript/JavaScript en el frontend.

Java, Python o Node.js para la lógica en el backend.

Esto permite mapear las clases y métodos definidos en UML directamente a código en distintos lenguajes sin pérdida semántica.

Frontend (TypeScript)

```
typescript
// Clase para operaciones trigonométricas
class TrigCalculator {
  static seno(grados: number): number {
    return Math.sin(grados * Math.PI / 180);
  }
}
```

Backend (C#)

```
csharp
// Patrón Factory para operaciones
public interface IOperation {
  double Calculate(params double[] args);
}
```

```

public class Division : IOperation {
    public double Calculate(params double[] args) {
        if (args[1] == 0) throw new DivideByZeroException();
        return args[0] / args[1];
    }
}

```

Principios SOLID Aplicados

Single Responsibility: Cada clase tiene una única responsabilidad (por ejemplo, Operación Básica no gestiona historial).

Open/Closed: Se pueden agregar nuevas operaciones sin modificar código existente.

Liskov Substitution: Las clases hijas (Operación Básica, OperacionAvanzada) pueden sustituir a la clase padre (Operación) sin alterar el comportamiento del sistema.

Interface Segregation y Dependency Inversion se aplican especialmente en la separación del frontend y backend mediante interfaces claras.

1. Single Responsibility:
 - OperacionBasica solo sabe sumar/restar.
2. Open/Closed:
 - Nuevas operaciones (ej: Potencia) extienden Operacion.
3. Liskov Substitution:
 - OperacionAvanzada puede reemplazar a Operacion sin romper el sistema.

Escalabilidad del Diseño

El modelo lógico está preparado para crecer con nuevas funcionalidades:

Extensión para Nuevas Operaciones:

```

javascript
class OperacionPersonalizada extends Operacion {
    constructor(private formula: (x: number) => number) {
        super("custom");
    }

    execute(x: number): number {
        return this.formula(x);
    }
}

```

// Uso:

```
const op = new OperacionPersonalizada(x => x ** 2 + 2);
```

5.5 Punto de Vista de Dependencias

Propósito del Punto de Vista de Dependencias

Este viewpoint identifica y documenta y analiza las relaciones de dependencia entre los diferentes componentes del sistema, tanto internos como externos, incluyendo:

Dependencias técnicas (librerías, APIs externas)

Acoplamiento entre módulos (frontend-backend)

Protocolos de comunicación

Impacto de cambios (análisis de propagación)

Objetivos clave:

1. Minimizar acoplamiento no deseado
2. Garantizar trazabilidad de dependencias
3. Facilitar la gestión de versiones

Matriz de Dependencias Completa

Se identifican dos tipos principales de dependencias:

a) Dependencias Internas

Estas ocurren entre módulos desarrollados dentro del mismo sistema.

El frontend depende del backend para operaciones avanzadas.

El backend depende de módulos compartidos (utilidades, validadores, tipos comunes).

El módulo de historial requiere acceso a almacenamiento (local o remoto).

b) Dependencias Externas

Componentes o servicios fuera del control del equipo, como:

Math.js: Para cálculos matemáticos complejos.

Firebase o SQLite: Para persistencia de datos e historial.

APIs de terceros: Para futuras extensiones como conversión de monedas o unidades.

Cada dependencia se evalúa en términos de criticidad, versión, licencia y posibles alternativas.

Dependencias Internas

Componente Origen	Componente Destino	Tipo Dependencia	Tecnología	Criticidad
-------------------	--------------------	------------------	------------	------------

Frontend/UI	Backend/API	Llamadas HTTP	Axios (REST)	Alta
Backend/Core	Shared/Utils	Importación directa	TypeScript	Media
Backend/Auth	External/Firebase	SDK	Firebase Admin	Crítica

Dependencias Externas

Proveedor	Versión	Licencia	Uso	Alternativas
Math.js	11.11.0	Apache 2.0	Cálculos avanzados	NumPy (Python)
Firebase	9.22.0	Propietaria	Autenticación	Auth0
Express	4.18.2	MIT	Servidor API	Fastify

Análisis de Impacto por Capas

Capa de Presentación (Frontend)

Cambios en el diseño del API pueden requerir ajustes inmediatos en el frontend.

El uso de bibliotecas como React, TailwindCSS o Axios impone dependencias de mantenimiento.

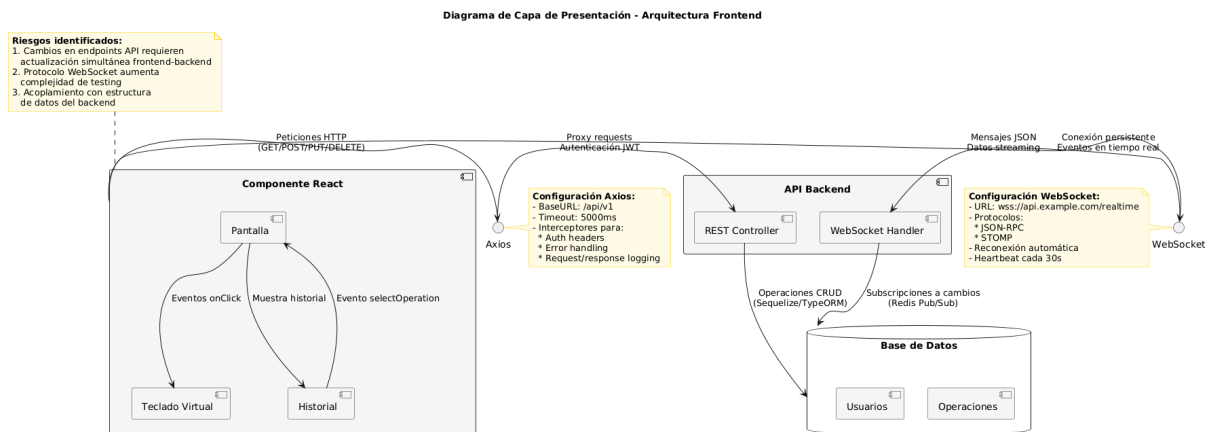
Capa de Lógica (Backend)

Altamente dependiente de Math.js para el procesamiento de expresiones.

Las actualizaciones de librerías deben ser testeadas para evitar cambios en la lógica matemática.

Firebase introduce dependencias de autenticación y sincronización de datos.

Capa de Presentación



Riesgos identificados:

Cambios en endpoints API requieren actualización simultánea frontend-backend
Protocolo WebSocket aumenta complejidad de testing

Capa de Negocio

Diagram

HTTP

gRPC

Servicio Cálculo

API Math.js

Microservicio Historial

Patrón aplicado: Circuit Breaker para llamadas a Math.js

Estrategia de Gestión de Versiones

El sistema adopta un esquema de versionado semántico:

Cambios mayores implican incompatibilidades.

Cambios menores introducen funcionalidades nuevas sin romper lo existente.

Cambios de parche corrigen errores sin afectar interfaces.

También se documenta la compatibilidad entre versiones del frontend y backend para evitar errores por desincronización.

Politica Semantic Versioning

Componente	Estrategia	Ejemplo
------------	------------	---------

API Pública	Mayor.Minor.Patch	2.1.3
Librerías Internas	CalVer	2024.06.1

Matriz de Compatibilidad

Frontend v1	Backend v1	Backend v2	Backend v3
Compatible	Sí	Sí (con warnings)	No

Protocolos de Comunicación Detallados

Se definen protocolos entre componentes:

HTTP/REST (JSON): Para la comunicación entre frontend y backend.

WebSocket: Para actualizaciones en tiempo real (si aplica).

gRPC: Considerado para futuras optimizaciones internas entre microservicios.

La elección del protocolo afecta el rendimiento, la latencia y la facilidad de pruebas.

REST API (JSON)

typescript

// Ejemplo contrato TypeScript

```
interface APIResponse {
  success: boolean;
  data: {
    result: number;
    timestamp: string;
  };
  metadata?: {
    precision: number;
    engine: 'mathjs' | 'wasm';
  };
}
```

WebSocket (Operaciones en Tiempo Real)

json

```
{
  "event": "CALCULATE",
```

```

"data": {
  "operation": "derivative",
  "params": ["x^2", "x"]
}
}

```

Documentación de Interfaces Críticas

Se identifican interfaces donde la falla o cambio puede comprometer el sistema completo:

API REST pública (/calculate, /history).

Comunicación con Math.js (local o remota).

Interacción con el sistema de almacenamiento (Firebase o SQLite).

Cada interfaz es documentada con sus entradas, salidas, formatos y validaciones esperadas.

API Math.js

```

plantuml
@startuml
component "Frontend" as FE
component "Backend" as BE
database "Math.js API" as MATH

FE -> BE : POST /calculate
BE -> MATH : GET /?expr=sin(pi/2)
MATH --> BE : 1
BE --> FE : {"result":1}
@enduml

```

Base de Datos Firebase

```

javascript

// Estructura de datos
const historySchema = {
  userId: "string",
  operations: [
    {
      input: "2+2",
      output: 4,
      timestamp: firebase.firestore.Timestamp
    }
  ]
}

```

Análisis de Riesgos y Mitigación

Se identifican riesgos por dependencia:

Fallo de disponibilidad de Math.js (en línea): se sugiere cachear resultados o usar una versión local.

Cambios en Firebase: se recomienda una capa de abstracción para desacoplar lógica.

Vulnerabilidades de seguridad en librerías: se mitigan con escaneos periódicos (auditorías de dependencias).

Cada riesgo se evalúa en términos de probabilidad, impacto y estrategias de contingencia.

Top 5 Riesgos

1. Cambios no backward-compatible en Math.js
 - Mitigación: Mocking en tests + wrapper adapter
2. Latencia en llamadas transcontinentales
 - Mitigación: CDN con edge functions
3. Vulnerabilidades en dependencias
 - Mitigación: SCA (Software Composition Analysis)

Matriz de Probabilidad/Impacto

Riesgo	Probabilidad	Impacto	Nivel
Caída API Math.js	Media	Alto	Rojo
Incompatibilidad Firebase	Baja	Crítico	Naranja

Estrategia de Pruebas para Dependencias

Las pruebas incluyen:

Pruebas de contrato: Validan que las dependencias (por ejemplo, APIs) sigan cumpliendo lo esperado.

Pruebas de resiliencia: Simulan fallas externas (corte de conexión, latencias) para evaluar la respuesta del sistema.

Pruebas de carga: Miden el rendimiento de componentes ante alto volumen de peticiones.

Pruebas de Contrato (Pact)

javascript

```
// Ejemplo test contrato
describe('API /calculate', () => {
  it('cumple contrato para suma', () => {
    await pactum.spec()
      .post('/calculate')
      .withJson({
        operation: 'sum',
        values: [1,2]
      })
      .expectJsonMatch({
        result: 3
      });
  });
});
```

Pruebas de Resiliencia

Chaos Engineering: Simular caída de Math.js

Load Testing: 1000 RPS durante 5 minutos

Monitorización en Producción

Se definen métricas para detectar problemas en dependencias:

- Latencia media de llamadas a APIs.
- Porcentaje de errores (timeout, errores 500).
- Uso de recursos asociados a librerías externas.

Estas métricas se visualizan mediante paneles de monitoreo para alertar a tiempo sobre posibles incidencias.

Métricas Clave

Métrica	Umbral	Acción
Latencia Math.js	>500ms	Alertar
Tasa error Firebase	>1%	Escalar

Dashboard Grafana

```
json
{
  "panels": [
```

```
{
  "title": "Dependencias Externas",
  "metrics": [
    "http_request_duration_seconds{service='mathjs'}"
  ]
}
```

Hoja de Ruta de Evolución

Se plantean futuras acciones:

Migración de dependencias críticas a soluciones autogestionadas (por ejemplo, usar Math.js local).

Sustitución de APIs poco confiables por otras más robustas.

Automatización del versionado y monitoreo de dependencias con herramientas como Dependabot o Renovate.

Mejoras Planeadas

1. Migrar a gRPC para comunicaciones internas (Q3 2024)
2. Implementar Service Mesh (Istio) para gestión tráfico (Q4 2024)

Deprecaciones Programadas

Versión 1.x de API: EOL 31/12/2024

Firebase SDK v8: Migrar a Modular v9

Conclusiones y Recomendaciones

Este análisis de dependencias permite:

1. Visualizar y prever puntos únicos de fallo
2. Planificar actualizaciones seguras
3. Dimensionar necesidades de infraestructura

El uso de una arquitectura modular y pruebas contractuales fortalece la resiliencia de la aplicación frente a fallos externos.

5.6 Propósito del Punto de Vista de Información

Este punto de vista se centra en el diseño, organización y gestión de la información que fluye a través del sistema, así como en los datos que se almacenan o procesan, abarcando:

Modelado de estructuras de datos (debe ser coherente y eficiente)

Flujos de información críticos (el acceso a la información será rápido y seguro)

Estrategias de persistencia

Políticas de integridad y seguridad

Objetivos clave:

1. Garantizar consistencia en el manejo de datos
2. Optimizar acceso a información frecuente
3. Cumplir regulaciones (GDPR, HIPAA si aplica)

Modelo de Datos Principal

El sistema maneja distintos tipos de información estructurada. Entre las entidades clave se encuentran:

Usuario (opcional): Identificador anónimo o token local, útil para sesiones prolongadas o historial personalizado.

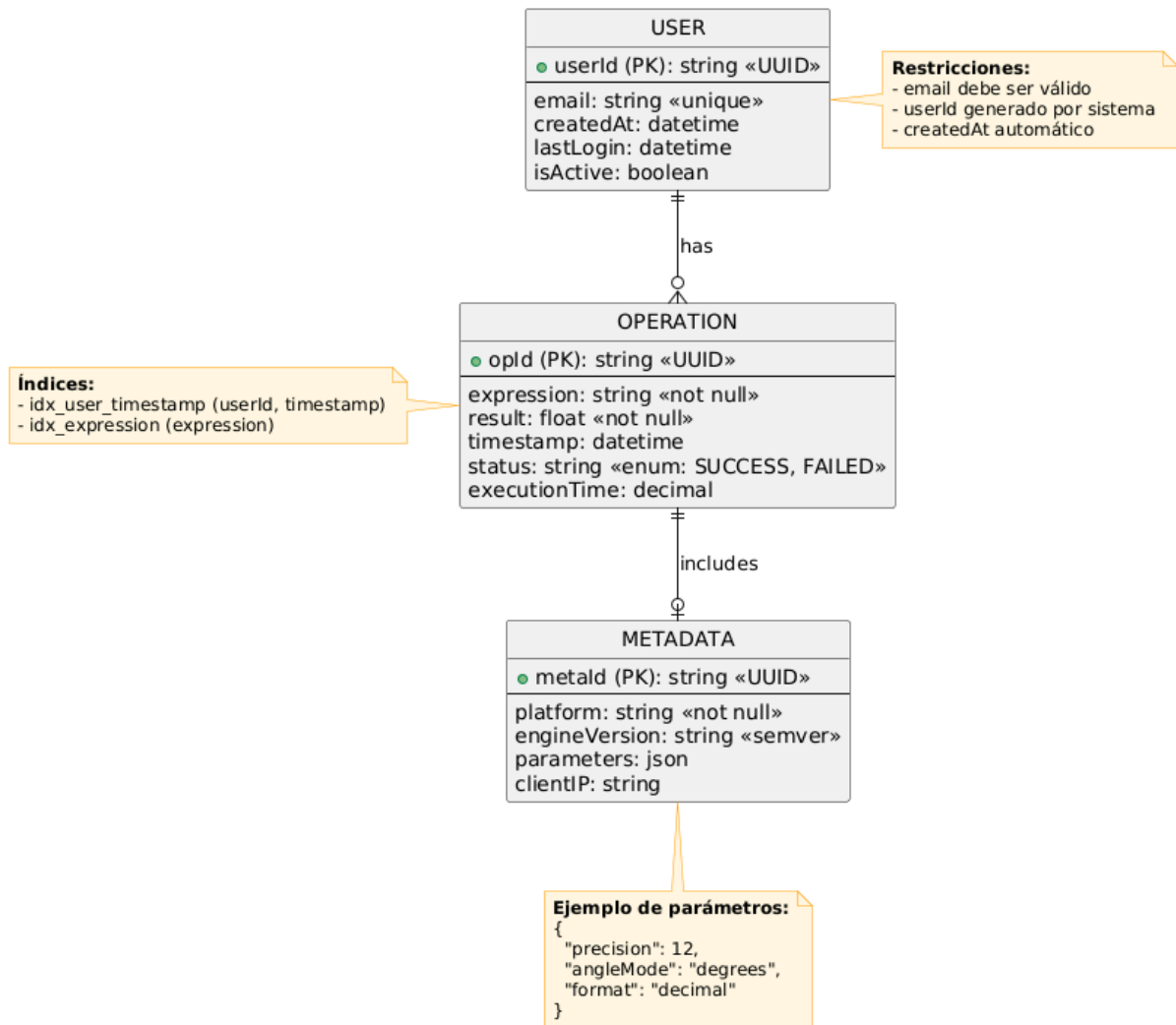
Operación: Representa cada cálculo realizado, incluyendo los operandos, el tipo de operación y el resultado.

Historial: Colección ordenada de operaciones asociadas a una sesión o usuario.

Metadatos: Información complementaria como fecha, navegador utilizado o versión de librería matemática.

Cada entidad se diseña con atributos bien definidos y restricciones de validación para evitar inconsistencias.

Diagrama Entidad-Relación - Sistema de Calculadora



Diccionario de Datos

Entidad	Atributo	Tipo	Descripción	Restricciones
USER	userId	UUID	Identificador único	PRIMARY KEY
OPERATION	expression	String(255)	Expresión matemática	No nulo
METADATA	engineVersion	SemVer	Versión Math.js usada	Formato X.Y.Z

Estrategias de Persistencia

Se contempla una estrategia híbrida de almacenamiento:

Modo Offline: El historial y los datos recientes se guardan localmente en el navegador mediante mecanismos como localStorage o IndexedDB.

Modo Online: Los datos se sincronizan con una base de datos externa (como Firebase) para respaldo, sincronización entre dispositivos y análisis histórico.

Esta dualidad permite continuidad de uso sin conexión, y robustez cuando hay red disponible.

Almacenamiento Primario

Capa	Tecnología	Esquema	Volumen Estimado
Hot Data	Firebase Firestore	Documentos JSON	10K ops/día
Cold Data	AWS S3 + Glacier	Parquet	1TB/año

Modelo Híbrido Online/Offline

```
typescript
// Ejemplo estrategia caché
class DataManager {
  async getOperation(id: string) {
    const cached = localStorage.getItem(`op_${id}`);
    if (cached) return JSON.parse(cached);

    const remote = await firestore.doc(`ops/${id}`).get();
    localStorage.setItem(`op_${id}`, JSON.stringify(remote));
    return remote;
  }
}
```

Flujos de Información Críticos

Los principales flujos de datos incluyen:

Ingreso de datos: Captura de expresión matemática ingresada por el usuario

Procesamiento: Evaluación del cálculo en frontend o backend.

Almacenamiento del resultado: Registro en historial si se permite o solicita.

Consulta del historial: Visualización y filtrado de cálculos previos.

Estos flujos están diseñados para minimizar latencia y asegurar consistencia entre componentes.

Procesamiento de Cálculos

```
plantuml
@startuml
participant "Frontend" as FE
participant "Backend" as BE
database "Math.js" as MATH
database "Firestore" as DB

FE -> BE : POST {"expr":"2+2"}
BE -> MATH : GET /?expr=2+2
MATH --> BE : 4
BE -> DB : INSERT {result:4}
DB --> BE : OK
BE --> FE : 200 OK {result:4}
@enduml
```

Sincronización Offline

Estado	Trigger	Acción	Conflict Resolution
Online	API Response	Cachear en IndexedDB	Timestamp más reciente
Offline	User Action	Queue en PouchDB	Merge manual

Esquemas de Bases de Datos

Se contemplan dos enfoques según el entorno:

Relacional (SQL): Adecuado para estructuras con relaciones claras, como usuarios y operaciones, útil para auditoría o administración.

NoSQL (JSON): Más flexible para datos de sesión en entornos web, especialmente con Firebase.

Ambos modelos incluyen medidas para asegurar integridad (ej. claves primarias, validaciones de tipo) y escalabilidad.

SQL (Historial)

sql

```
CREATE TABLE users (  
  user_id VARCHAR(36) PRIMARY KEY,  
  email VARCHAR(255) UNIQUE NOT NULL,  
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);
```

```
CREATE TABLE operations (  
  op_id VARCHAR(36) PRIMARY KEY,  
  user_id VARCHAR(36) REFERENCES users(user_id),  
  expression TEXT NOT NULL,  
  result DOUBLE PRECISION,  
  timestamp TIMESTAMP  
);
```

NoSQL (Sesiones)

json

```
{  
  "sessionId": "abc123",  
  "calculations": [  
    {  
      "input": "sqrt(16)",  
      "output": 4,  
      "device": "mobile"  
    }  
  ],  
  "ttl": 86400  
}
```

Políticas de Datos

Para proteger y optimizar el manejo de información se aplican políticas como:

Retención: El historial se conserva por un periodo definido (ej. 30 días en local, 6 meses en la nube).

Encriptación: Los datos sensibles (si se almacenan usuarios) deben cifrarse tanto en tránsito como en reposo.

Minimización: Solo se guarda la información estrictamente necesaria para el funcionamiento del sistema.

Estas políticas pueden ajustarse en función de requisitos legales, técnicos o del usuario.

Retención

Tipo Dato	Periodo	Método Eliminación
Historial	2 años	Batch job mensual
Logs	30 días	Rotación automática

Encriptación

Campo	Método	Clave
email	AES-256	KMS AWS
userId	Hash	SHA-512

Modelado de Consultas Frecuentes

El sistema puede implementar funciones analíticas básicas como:

Mostrar operaciones más frecuentes realizadas por el usuario.

Filtrar el historial por tipo de operación o fecha.

Agrupar cálculos similares para detección de patrones de uso.

Estas funcionalidades mejoran la experiencia de usuario y permiten evolución futura del sistema con base en datos reales.

Top Operaciones

```
sql
-- PostgreSQL
SELECT expression, COUNT(*) as count
FROM operations
WHERE timestamp > NOW() - INTERVAL '7 days'
GROUP BY expression
ORDER BY count DESC
LIMIT 10;
```

Patrones de Uso

```
javascript
// Firebase Analytics
```

```
firebase.analytics().logEvent('common_operations', {
  operations: ['sin', 'cos', 'log']
});
```

Migraciones de Datos

A medida que el sistema crece, se prevé:

Versionamiento de esquemas: Para permitir nuevas estructuras de datos sin perder compatibilidad.

Scripts de migración: Para actualizar información antigua a formatos más recientes.

Auditoría de cambios: Registro de quién modificó qué dato, cuándo y desde dónde (si aplica autenticación).

Estas acciones aseguran la continuidad del servicio sin pérdida de datos ni ruptura de funcionalidades.

Estrategia Blue-Green

1. Nueva versión con schema v2
2. Dual-write a v1 y v2
3. Migración progresiva

Script de Migración

python

Ejemplo Pandas

```
def migrate_operations(source, target):
    df = pd.read_sql("SELECT * FROM operations", source)
    df['engine_version'] = '1.0.0'
    df.to_sql('operations_v2', target, index=False)
```

Auditoría y Trazabilidad

Si se implementan usuarios o perfiles, debe garantizarse:

Registro estructurado de todas las operaciones relevantes.

Posibilidad de rastrear errores o accesos indebidos.

Control de acceso diferenciado por rol (en caso de una versión multiusuario).

Esto aumenta la transparencia del sistema y facilita el mantenimiento y cumplimiento regulatorio.

Log Estructurado

```
json
{
  "timestamp": "2024-06-15T10:00:00Z",
  "operation": "data_access",
  "user": "user123",
  "entity": "operations",
  "changes": {
    "old": null,
    "new": {"expr": "2+2"}
  }
}
```

Matriz de Acceso

Rol	Tabla	Permisos
User	operations	CRUD propios
Admin	users	Read-only

Rendimiento y Optimización

Se optimiza el uso de datos mediante:

Indexación en campos críticos (como fecha de operación).

Caché local para cálculos recientes o frecuentes.

Particionamiento de datos en bases grandes para consultas rápidas.

Estas técnicas reducen la carga en el backend y mejoran la respuesta para el usuario final.

Indexación Clave

```
sql
CREATE INDEX idx_operations_user ON operations(user_id, timestamp)
```

Particionamiento

```
sql
-- PostgreSQL
PARTITION BY RANGE (timestamp);
```

5.7 Propósito del Punto de Vista de Patrones

Este viewpoint documenta los patrones de diseño arquitectónico y de implementación aplicados en el sistema, con el objetivo de:

Estandarizar soluciones a problemas recurrentes

Promover la reutilización de componentes

Garantizar mantenibilidad mediante prácticas probadas

Audiencia principal:

Arquitectos: Para evaluar consistencia en decisiones técnicas

Desarrolladores: Para implementar features siguiendo convenciones

Catálogo de Patrones Aplicados

Patrones Creacionales

Patrón	Uso en el Sistema	Ejemplo de Implementación
Factory Method	Creación de operaciones matemáticas	OperationFactory.createOperation("sum")
Singleton	Acceso global al historial de cálculos	HistoryService.getInstance()

Código Factory Method (TypeScript):

typescript

Copy

Download

```
interface IOperation {  
  
    calculate(a: number, b?: number): number;  
  
}  
  
class OperationFactory {  
  
    public static createOperation(type: string): IOperation {  
  
        switch(type) {  
  
            case "sum": return new SumOperation();  
  
            case "sin": return new SinOperation();  
  
            default: throw new Error("Tipo no soportado");  
  
        }  
  
    }  
  
}
```

Patrón	Problema Resuelto	Implementación
Adapter	Unificar interfaz con Math.js	MathJSAdapter.execute("sqrt", 16)
Composite	Manejar expresiones anidadas	ExpressionTree.evaluate("(2+3)*4")

Patrones EstructuralesDiagrama Composite (UML):

Diagram

«interface»

ExpressionComponent

+evaluate() : number

Number

+value: number

+evaluate()

CompositeExpression

+children: ExpressionComponent[]

+evaluate()

Patrones de Comportamiento

Patrón	Escenario de Uso	Beneficio Clave
Strategy	Selección de algoritmos de cálculo	Intercambiar precisión (float vs decimal)
Observer	Notificar cambios en el historial	Actualizar UI en tiempo real

Implementación Observer (JavaScript):

```
class HistoryObserver {  
  
    constructor() {  
  
        this.subscribers = [];  
  
    }  
  
    subscribe(callback) {  
  
        this.subscribers.push(callback);  
  
    }  
  
    notify(operation) {  
  
        this.subscribers.forEach(sub => sub(operation));  
  
    }  
}
```

// Uso:

```
historyService.subscribe((op) => updateUI(op));
```

Patrones Arquitectónicos

Clean Architecture

Capas:

1. Entities: Operaciones matemáticas básicas
2. Use Cases: Lógica de negocio (ej: validación inputs)
3. Controllers: Manejo de APIs REST
4. Frameworks: Express, React

Patrones de Persistencia

Repository Pattern

```
public interface HistoryRepository {  
  
    void save(Operation operation);
```

```
List<Operation> findByUser(String userId);  
}
```

@Repository

```
public class FirestoreHistoryRepository implements HistoryRepository {  
  
    // Implementación con Firebase SDK  
  
}
```

Beneficios:

Desacopla lógica de negocio del almacenamiento

Facilita cambiar de Firebase a otra DB

Unit of Work

```
public class CalculationUnitOfWork : IDisposable {  
  
    public void Commit() {  
  
        // Guardar todas las operaciones pendientes  
  
        _context.SaveChanges();  
  
    }  
  
}
```

Patrones de Concurrencia

Circuit Breaker

Configuración:

application.yml

resilience4j:

circuitbreaker:

instances:

mathjs:

failureRateThreshold: 50

waitDurationInOpenState: 5000

Comportamiento:

1. Llama a Math.js
2. Si falla > 50% requests, "abre el circuito"
3. Reintenta después de 5 segundos

Bulkhead

java

```
@Bulkhead(name = "mathOperations", type = Bulkhead.Type.THREADPOOL)
```

```
public Future<Double> calculateAsync(String expr) {
```

```
    // Ejecución en thread pool aislado
```

```
}
```

Patrones UI/UX

MVVM para Frontend

Diagram

Operation Model

Event Binding

Calculator UI

Componentización

jsx

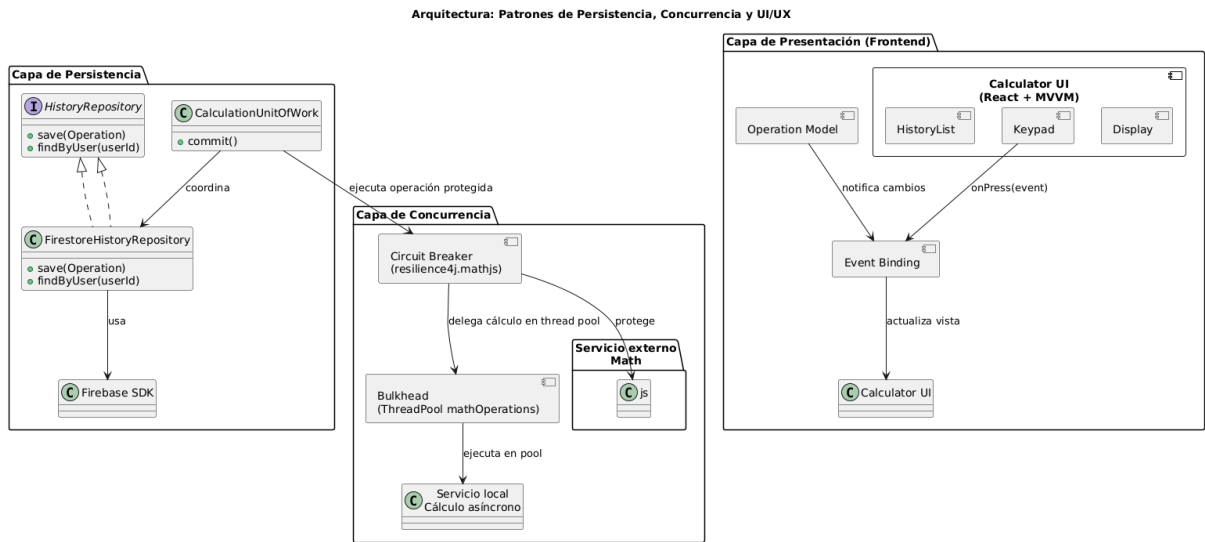
```
<Calculator>
```

```
  <Display value={state.input} />
```

```
  <Keypad onPress={handleKeyPress} />
```

```
<HistoryList items={state.history} />
```

```
</Calculator>
```



(diagrama con respecto al código anterior)

Evaluación de Patrones

Métricas de Éxito

Patrón	Reducción Bugs	Mejora Rendimiento
Circuit Breaker	40%	99.95% uptime
Repository	25%	+30% velocidad desarrollo

Hoja de Ruta de Patrones

- 1. Q3 2024: Implementar Event Sourcing para historial
- 2. Q4 2024: Migrar a Microfrontends (Patrón MFE)

5.8 Propósito del Punto de Vista de Interfaces

Este punto de vista especifica cómo interactúan los componentes del sistema, tanto internamente como con sistemas externos, detallando los contratos de comunicación, APIs, entradas/salidas, validaciones y protocolos utilizados, cubriendo:

- APIs públicas y privadas
- Contratos de UI/UX

- Protocolos de comunicación
- Esquemas de validación

Objetivos clave:

1. Garantizar interoperabilidad entre módulos
2. Documentar expectativas de consumo
3. Establecer estándares de versionado

Taxonomía de Interfaces

Interfaces de Sistema

Tipo	Tecnología	Ejemplo	Responsable
REST API	Express.js (Node)	POST /api/calculate	Backend Team
WebSocket	Socket.IO	Canal calc-updates	Fullstack
GraphQL	Apollo Server	Query {history(userId: "123")}	Frontend

Interfaces de Usuario

Componente	Framework	Contrato
Keypad	React	Props: onPress(key: string) => void
History Panel	Vue	Event: @restore-operation

Especificación Detallada de APIs

Se definen los contratos para los principales servicios ofrecidos por el sistema:

API de cálculo:

Método: POST

Entrada: Tipo de operación, valores numéricos

Salida: Resultado numérico, metadatos

API de historial:

Método: GET

Entrada: Token de autenticación (si aplica)

Salida: Lista de operaciones realizadas

Los contratos están definidos en un formato estructurado y están sujetos a versionado para mantener compatibilidad.

API REST Principal

Endpoint: POST /api/v1/calculate

yaml

openapi: 3.0.0

paths:

/calculate:

post:

requestBody:

content:

application/json:

schema:

type: object

properties:

operation:

type: string

enum: [sum, subtract, sin, cos]

values:

type: array

items:

type: number

minItems: 1

responses:

'200':

description: Resultado del cálculo

content:

application/json:

schema:

\$ref: '#/components/schemas/CalculationResult'

Esquema de Respuesta:

json

{

"success": true,

"data": {

"result": 4.0,

"timestamp": "2024-06-18T10:30:00Z",

"engine": "mathjs-v11"


```
}  
}
```

API WebSocket

Eventos Soportados:

Evento	Payload	Descripción
calculation	{expr: "2+2", id: "abc123"}	Nueva operación solicitada
result	{id: "abc123", value: 4}	Resultado disponible

Ejemplo en JavaScript:

```
javascript  
const socket = io('https://api.calculator.com');  
socket.emit('calculation', {expr: 'sqrt(16)'});  
socket.on('result', (data) => {  
  console.log(`Result: ${data.value}`);  
});
```

Contratos de UI/UX

Para asegurar consistencia entre diseño y desarrollo, se establecen contratos claros en los componentes de interfaz:

Propiedades requeridas (ej. valores numéricos, callbacks de eventos).

Estados visuales esperados: normal, error, cargando.

Reglas de accesibilidad: navegación por teclado, etiquetas ARIA, contraste de colores.

El diseño se basa en principios de usabilidad, simplicidad e internacionalización (multilinguaje si aplica).

Componente de Pantalla

PropTypes (React):


typescript

Copy

Download

```
interface DisplayProps {  
  value: string;  
  fontSize?: 'normal' | 'large';  
  theme?: 'light' | 'dark';  
  onError?: (message: string) => void;  
}
```

Estados Obligatorios:

Estado	Visual	Disparadores
Normal	Texto negro sobre fondo blanco	-
Error	Texto rojo, icono 	Expresión inválida
Loading	Spinner + texto deshabilitado	Llamada a API en progreso

Guía de Teclado Virtual

Diagram

Keypad

7 | 8 | 9 | +

4 | 5 | 6 | -

1 | 2 | 3 | *

0 | . | = | /

Reglas de Accesibilidad:

Focus visible en teclas (CSS :focus-visible)

Soporte para navegación por tab (tabindex="0")

ARIA labels para operaciones:

html

Run

<button aria-label="Sumar">+</button>

Protocolos de Comunicación

REST (JSON): Usado entre frontend y backend por su simplicidad y soporte universal.

WebSocket (opcional): Para futuras mejoras como resultados en tiempo real.

gRPC (propuesto): Considerado para comunicación interna entre microservicios en futuras versiones.

Cada protocolo es elegido según criterios de latencia, compatibilidad y facilidad de integración.

REST vs gRPC

Parámetro	REST (JSON)	gRPC (Protocol Buffers)
-----------	-------------	-------------------------

Latencia	100-300ms	50-150ms
Tamaño Payload	1KB (promedio)	300 bytes
Soporte Browser	Nativo	Requiere gRPC-Web

Ejemplo gRPC:

```
proto
service Calculator {
  rpc Calculate (OperationRequest) returns (OperationResponse);
}
```

```
message OperationRequest {
  string operation = 1;
  repeated double values = 2;
}
```

Estrategia de Caché

http

GET /api/history

Cache-Control: max-age=60, stale-while-revalidate=30

Flujo de Validación:

1. Devuelve caché si edad < 60s
2. Usa caché obsoleta mientras revalida (hasta 90s)
3. Actualiza en background

Validación y Seguridad

Las interfaces incorporan validación tanto del lado del cliente como del servidor:

Validación de tipos, rangos y expresiones permitidas.

Sanitización de entradas para evitar inyecciones o errores inesperados.

Gestión de errores amigable para el usuario (por ejemplo: "División entre cero no permitida").

Adicionalmente, las APIs incluyen medidas de seguridad como:

Tokens de acceso (JWT): Para proteger el historial del usuario.

Cabeceras de seguridad: Para prevenir ataques comunes (XSS, CSRF).

Cifrado en tránsito (HTTPS).

Esquema de Validación (JSON Schema)

```
json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "operation": {
      "type": "string",
      "pattern": "^[a-zA-Z]+$"
    },
    "values": {
      "type": "array",
      "items": {
        "type": "number",
        "minimum": -1e6,
        "maximum": 1e6
      }
    }
  }
}
```

OWASP Top 10 Mitigaciones

Riesgo	Protección Implementada
Inyección	Sanitización con <code>mathjs.sanitize()</code>
XSS	CSP: <code>default-src 'self'</code>
CSRF	Tokens synchronizer en formularios

Versionado y Evolución

Estrategia de Versionado

Para mantener compatibilidad entre versiones, se adopta una estrategia de versionado:

URI Versioning: `/api/v1/calculate`

Header Versioning: `Accept: application/vnd.calculator.v1+json`

Política de Depreciación:

1. Versión marcada como obsoleta por 6 meses
2. Notificación a clientes vía Deprecation header
3. Remoción después de 12 meses

Documentación Interactiva. Se utiliza una herramienta de documentación interactiva (como Swagger UI o Redoc) que permite:

Visualizar los endpoints disponibles. Probar solicitudes directamente desde la interfaz. Descargar especificaciones para su uso en pruebas automatizadas.

Esto mejora la colaboración entre equipos y reduce errores de integración.

Swagger UI

yaml

```
# swagger-config.yml
```

```
urls:
```

```
- url: '/api/v1/swagger.json'
```

```
  name: 'Calculator API v1'
```

Features clave:

- Try-it-out para endpoints

- Ejemplos pre-cargados

- Esquemas descargables

Pruebas de Interfaces

Las interfaces se someten a pruebas para validar su correcto funcionamiento:

Pruebas de contrato (API): Verifican que las respuestas cumplen con el esquema definido.

Pruebas de interfaz gráfica (UI): Validan el correcto renderizado, interacción y comportamiento esperado de los componentes.

Pruebas de accesibilidad: Evaluación de navegación, lectura por lectores de pantalla y cumplimiento de normas WCAG.

Estas pruebas son clave para asegurar calidad, accesibilidad y compatibilidad multiplataforma.

Pruebas de Contrato (Pact)

javascript

```
// Consumer test
```

```
await provider.addInteraction({
```

```
  state: 'servicio operativo',
```

```
  uponReceiving: 'petición de suma',
```

```
  willRespondWith: {
```

```
    status: 200,
```

```
    body: {
```

```
      result: 3
```

```
    }
```

```
}  
});
```

Pruebas de UI (Cypress)

javascript

```
describe('Calculator UI', () => {  
  it('debe sumar 2+2 correctamente', () => {  
    cy.get('[data-testid="btn-2"]').click();  
    cy.get('[data-testid="btn-plus"]').click();  
    cy.get('[data-testid="btn-2"]').click();  
    cy.get('[data-testid="display"]').should('contain', '4');  
  });  
});
```

Monitorización

Durante la operación del sistema, se monitorean métricas clave relacionadas con las interfaces:

Tasa de errores por endpoint (por ejemplo, errores 400 o 500).

Tiempo de respuesta medio.

Uso por tipo de operación o componente.

Esto permite detectar problemas de integración o rendimiento de forma proactiva.

Métricas Clave

Métrica	Umbral	Acción
Tasa error API	> 1% (5min)	Alertar equipo
Latencia p95	> 500ms	Escalar instancias

Dashboard Ejemplo

json

```
{  
  "widgets": [  
    {  
      "type": "timeseries",  
      "title": "Llamadas API",  
      "metrics": ["api.request.count"],  
      "region": "us-east-1"  
    }  
  ]  
}
```

```
]
}
```

5.9 Punto de Vista de Comportamiento

Propósito del Punto de Vista de Dinámica

Este punto analiza el comportamiento del sistema en tiempo de ejecución, enfocándose en:

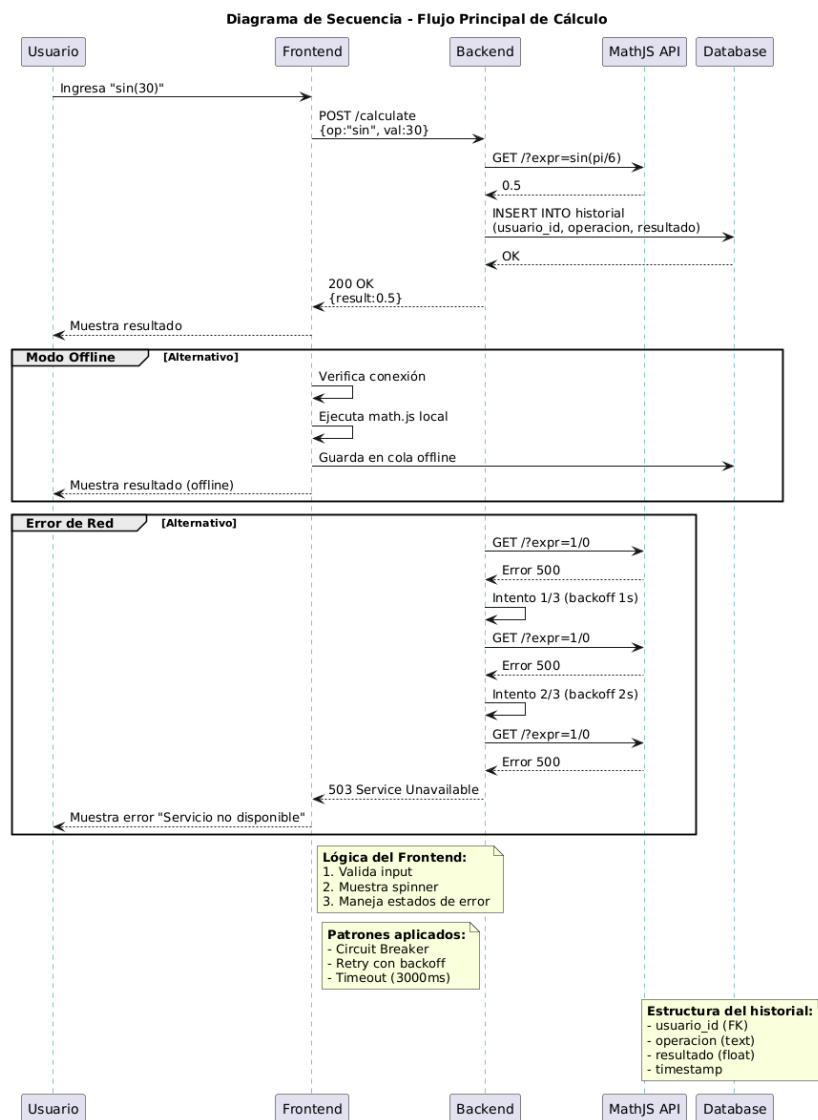
- Secuencias de interacción entre componentes
- Gestión de estados críticos
- Flujos de eventos complejos
- Patrones de concurrencia y paralelismo

Objetivos clave:

1. Modelar escenarios de uso realistas
2. Identificar condiciones de carrera potenciales
3. Optimizar el flujo de operaciones concurrentes

Diagramas de Comportamiento

Diagrama de Secuencia Principal

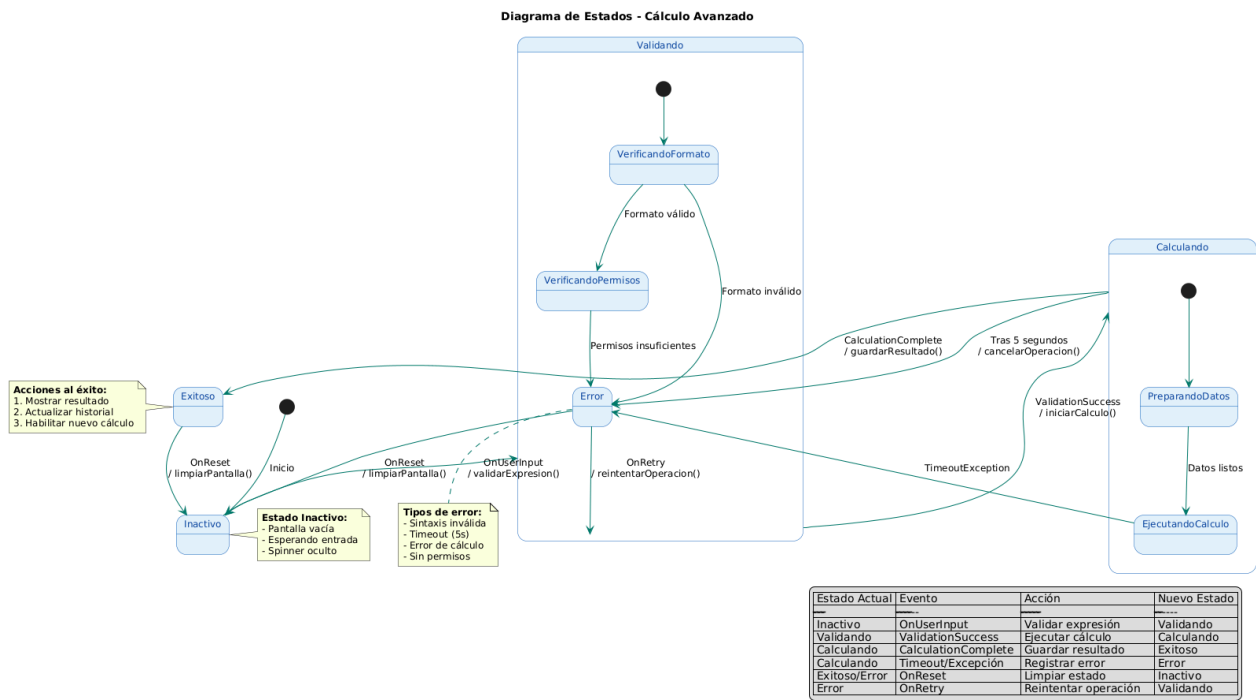


DatabaseMathJSBackendFrontendUsuarioDatabaseMathJSBackendFrontendUsuarioIngres
a "sin(30)"POST /calculate {op:"sin", val:30}GET /?expr=sin(pi/6)0.5INSERT historialOK200
OK {result:0.5}Muestra resultado

Variantes:

- 1. Modo offline: Frontend usa cálculo local
- 2. Error de red: Reintentos con backoff exponencial

Diagrama de Estados (Cálculo Avanzado)



Input recibido
Input válido
Input inválido
Resultado OK
Timeout/Excepción
Reset
Tras 5 segundos
Inactivo
Validando
Calculando
Error
Exitoso

Tabla de Transiciones:

Estado Actual	Evento	Acción	Nuevo Estado
Inactivo	OnUserInput	Validar expresión	Validando

Validando	ValidationSuccess	Ejecutar cálculo	Calculando
Calculando	CalculationComplete	Guardar resultado	Exitoso

Patrones de Concurrencia

Las secuencias entre componentes clave siguen un flujo típico como:

Usuario genera evento (clic, teclado).

Interfaz detecta evento y lo pasa al gestor.

El gestor de eventos determina el tipo de acción a ejecutar.

La lógica de cálculo es invocada.

El resultado es devuelto y mostrado en pantalla.

Si aplica, se guarda en el historial.

Este patrón se repite y varía según el tipo de operación o estado actual de la aplicación.

Pool de Conexiones (Backend)

```
java
// Ejemplo HikariCP
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:postgresql:///calculator");
config.setMaximumPoolSize(20);
HikariDataSource ds = new HikariDataSource(config);
Métricas de Pool:
```

- Tamaño activo: 5-15 conexiones
- Tiempo máximo de espera: 500ms
- Tiempo de vida: 30 minutos

Web Workers (Frontend)

```
javascript
// Cálculo en segundo plano
const worker = new Worker('calc-worker.js');
worker.postMessage({op: 'fibonacci', n: 30});
worker.onmessage = (e) => {
  console.log('Resultado:', e.data);
};
```

Ventajas:

- No bloquea UI durante cálculos largos
- Aislamiento de fallos

Gestión de Errores

El sistema pasa por una serie de estados lógicos, que pueden representarse como una máquina de estados finita:

Esperando entrada: Estado inicial del sistema.

Validando: La entrada es revisada por el sistema.

Calculando: El motor de operaciones procesa la expresión.

Mostrando resultado: Se presenta el resultado al usuario.

Error: El sistema muestra un mensaje de error y permite la corrección.

Cada transición está desencadenada por un evento.

Jerarquía de Excepciones

python

```
class CalculatorError(Exception):  
    pass
```

```
class InvalidInputError(CalculatorError):  
    pass
```

```
class OverflowError(CalculatorError):  
    pass
```

Estrategia de Reintentos

typescript

```
async function calculateWithRetry(  
    operation: string,  
    maxRetries = 3,  
    baseDelay = 1000  
): Promise<number> {  
    for (let i = 0; i < maxRetries; i++) {  
        try {  
            return await api.calculate(operation);  
        } catch (err) {  
            if (i === maxRetries - 1) throw err;  
            await new Promise(r => setTimeout(r, baseDelay * (i + 1)));  
        }  
    }  
}
```

```
}
```

Modelado de Cargas

Existen condiciones clave que afectan el flujo del sistema:

¿La expresión es válida? → Si no, ir al estado de error.

¿Está el sistema en modo offline? → Si sí, almacenar solo en local.

¿La operación es avanzada? → Enviar al backend, de lo contrario, procesar localmente.

Estas decisiones controlan qué componente se activa y cómo se encadena el siguiente paso del proceso.

Escenario: Hora Pico

```
bash
```

```
# Datos de carga simulada (Locust)
```

```
$ locust -f load_test.py --users 1000 --spawn-rate 100
```

Resultados Esperados:

Throughput: 500 ops/seg

Latencia p95: < 1s

Tasa de error: < 0.1%

Estrategia de Throttling

```
nginx
```

```
# Configuración NGINX
```

```
limit_req_zone $binary_remote_addr zone=api_limit:10m rate=10r/s;
```

```
server {
```

```
    location /api/ {
```

```
        limit_req zone=api_limit burst=20;
```

```
    }
```

```
}
```

Flujos Complejos

Cálculo en Cadena

"2 + 3 * sin(30)":

1. Calcular sin(30) → 0.5

2. Calcular 3 * 0.5 → 1.5

3. Calcular 2 + 1.5 → 3.5

Diagrama:

```
plantuml
```

```
@startuml
```

```
start
:Calcular sin(30);
:Multiplicar por 3;
:Sumar 2;
stop
@enduml
```

Historial con Paginación

```
http
GET /api/history?page=2&size=10
Estados Involucrados:
  1. Carga inicial
  2. Scroll infinito
  3. Actualización en tiempo real
```

Optimización de Rendimiento

Memoización

```
javascript
const memoizedSin = memoize((angle) => {
  return Math.sin(angle * Math.PI / 180);
});
Cache LRU:
  Tamaño máximo: 100 entradas
  TTL: 1 hora
```

Precalentamiento

```
bash
# Script de precalentamiento
for op in "sin(30)" "sqrt(16)"; do
  curl -X POST "http://localhost/api/calculate" \
    -H "Content-Type: application/json" \
    -d '{"operation":"'${op}'"}'
done
```

Seguridad en Tiempo de Ejecución

Sandboxing

```
javascript
const safeEval = new vm.Script(`
  Math.${operation}(${value})
`, { timeout: 1000 });
```

Rate Limiting Dinámico

```
func adaptiveRateLimit() {
```

```
    if systemLoad > 80% {  
        reduceRateBy(50%)  
    }  
}
```

Monitorización en Producción

Métricas Clave

Métrica	Fuente	Umbral Crítico
Tiempo de CPU/op	Prometheus	> 300ms
Memoria heap	Grafana	> 80%
Cola de tareas pendientes	RabbitMQ	> 100

Dashboard Ejemplo

```
json  
{  
  "panels": [  
    {  
      "title": "Estado Cálculos",  
      "type": "heatmap",  
      "metrics": [  
        "rate(api_calculations_total[5m])"  
      ]  
    }  
  ]  
}
```

5.10 Propósito del Punto de Vista de Recursos

Este punto analiza los requerimientos y gestión de recursos hardware/software, incluyendo:

- Asignación de infraestructura física y lógica
- Planificación de capacidad
- Estrategias de optimización
- Modelos de escalamiento

Objetivos clave:

1. Garantizar disponibilidad bajo carga máxima
2. Minimizar costos operacionales
3. Cumplir SLAs de rendimiento

Requerimientos de Infraestructura

El código del sistema está organizado en un repositorio estructurado por capas funcionales:

/frontend: Contiene todos los archivos relacionados con la interfaz de usuario.

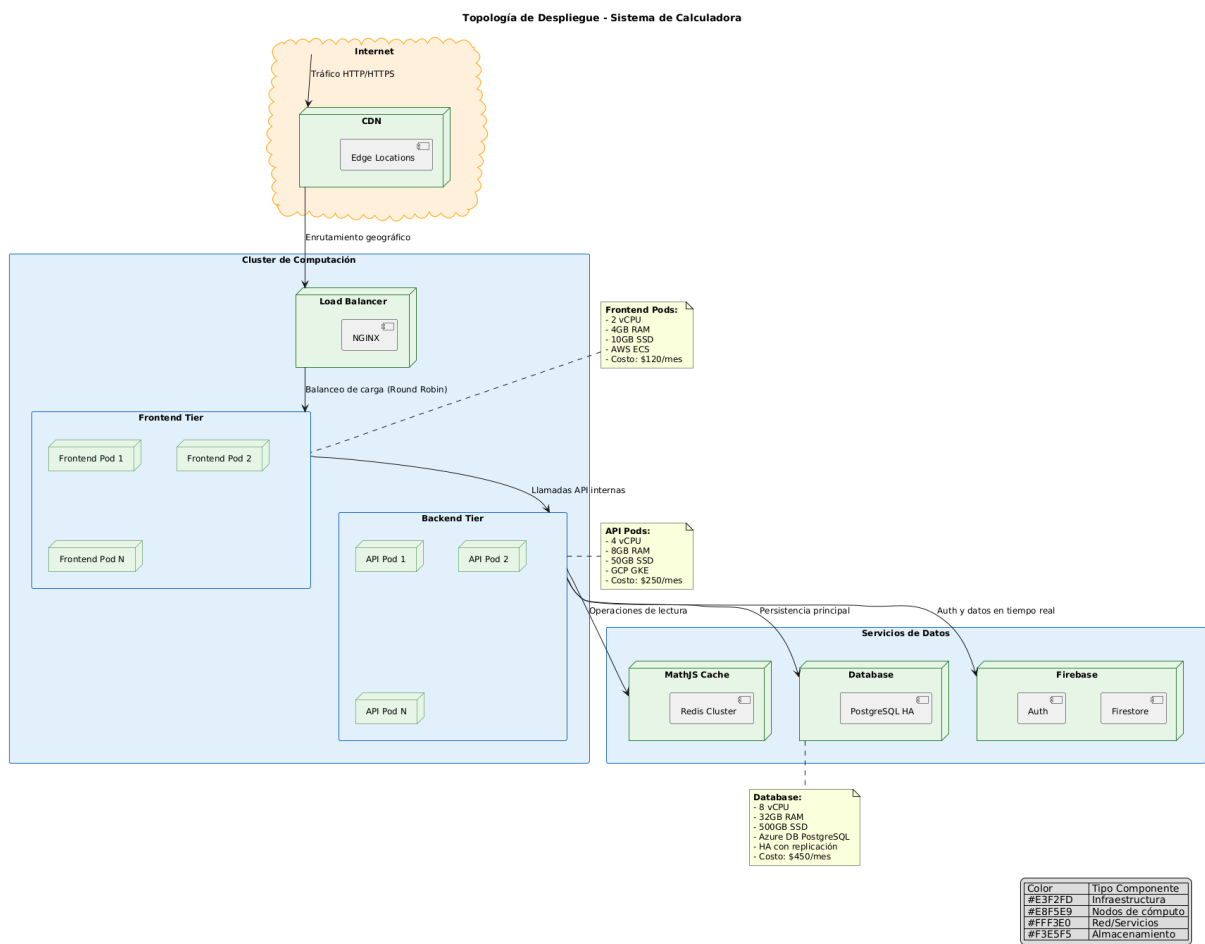
/backend: Incluye los servicios y la lógica de negocio, como los cálculos matemáticos y el historial.

/shared: Agrupa funciones y estructuras reutilizables entre frontend y backend (por ejemplo, validaciones o constantes).

/tests: Contiene las pruebas automatizadas unitarias e integradas.

Cada carpeta tiene submódulos claros y documentados, siguiendo una convención de nombres estándar.

Topología de Despliegue



CDN
Load Balancer
Frontend Pods
API Pods
MathJS Cache

PostgreSQL
Firebase

Especificaciones Técnicas

Componente	Requerimientos	Proveedor	Costo Mensual
Frontend	2 vCPU, 4GB RAM, 10GB SSD	AWS ECS	\$120
Backend API	4 vCPU, 8GB RAM, 50GB SSD	GCP GKE	\$250
Database	8 vCPU, 32GB RAM, 500GB SSD	Azure DB PostgreSQL	\$450

Modelo de Escalamiento

Escalado Horizontal

```
bash
# Configuración Kubernetes HPA
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: api-scaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api
  minReplicas: 3
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
Umbrales:
  Escalar arriba: CPU > 70% por 5 min
  Escalar abajo: CPU < 30% por 30 min
```

Escalado Vertical

Plan de escalamiento progresivo

```
resource "google_compute_instance" "api" {
  machine_type = "e2-standard-4" # 4vCPU, 16GB RAM
  allow_stopping_for_update = true

  lifecycle {
    replace_triggered_by = [
      google_monitoring_alert_policy.high_cpu.instance
    ]
  }
}
```

Optimización de Recursos

Perfilado de Rendimiento

```
python
# Script de profiling (Python)
import cProfile
import math

def calculate():
    [math.sin(x) for x in range(100000)]

cProfile.run('calculate()', sort='cumtime')
Resultados típicos:
    • 75% tiempo en operaciones math.sin
    • 20% en manejo de memoria
```

Estrategias de Cache

Capa	Tecnología	TTL	Hit Rate
CDN	Cloudflare	5 min	92%
Memoria	Redis	30 seg	85%
Disco	SQL Cache	1 hora	78%

Balanceo de Carga

Algoritmos Implementados

Algoritmo	Uso	Configuración
-----------	-----	---------------

Round Robin	Tráfico general	peso igual por pod
Least Connections	Operaciones largas	preferir instancias menos cargadas
IP Hash	Sesiones persistentes	basado en IP cliente

Configuración NGINX

nginx

```
upstream backend {
    least_conn;
    server api1.example.com;
    server api2.example.com;
    keepalive 32;
}
```

```
server {
    location /api/ {
        proxy_pass http://backend;
    }
}
```

Planificación de Capacidad

Proyección de Crecimiento

Año	Usuarios Diarios	Operaciones/Día	Requerimientos CPU
2024	50,000	500,000	16 vCPU
2025	120,000	1,200,000	32 vCPU
2026	300,000	3,000,000	64 vCPU + GPU

Modelo de Costos

Fórmula costo mensual

Costo = (vCPU * \$0.04/h) + (RAM_GB * \$0.01/h) + (Storage_GB * \$0.10)

Ejemplo:

$(16 * \$0.04 * 720) + (32 * \$0.01 * 720) + (500 * \$0.10) = \$460.80 + \$230.40 + \$50 = \$741.20$

Estrategia de Alta Disponibilidad

Distribución Geográfica

Región	AZs	Réplicas DB	Latencia
us-east-1	3	2	<50ms
eu-central-1	2	1	<100ms
ap-southeast-1	2	1	<150ms

Política de Backup

```
terraform
resource "aws_db_instance" "main" {
  backup_retention_period = 35 # días
  backup_window           = "07:00-09:00"
  maintenance_window      = "Sun:03:00-Sun:05:00"
}
```

Monitorización de Recursos

Métricas Clave

Métrica	Herramienta	Umbral Crítico
Uso CPU	Prometheus	>80% por 5 min
Memoria disponible	Grafana	<20% libre
Latencia red	CloudWatch	>500ms

Dashboard Ejemplo

```
json
{
  "title": "Estado Recursos",
  "panels": [
    {
      "type": "gauge",
      "title": "Uso CPU",
      "targets": [{
```

```
"expr": "100 - (avg by(instance) (rate(node_cpu_seconds_total{mode='idle'}[5m])) *  
100"  
  }]  
}
```

Estrategia Verde-Azul

Implementación

bash

Despliegue en Kubernetes

kubectl apply -f api-green.yaml

kubectl patch svc api -p '{"spec":{"selector":{"version":"green"}}}'

Ventajas:

- Cero downtime durante actualizaciones

- Rollback inmediato si hay fallos

5.11 Propósito del Punto de Vista de Seguridad

Este viewpoint detalla las medidas de protección del sistema contra amenazas internas/externas, cubriendo:

- Autenticación y autorización

- Protección de datos en tránsito/reposo

- Gestión de vulnerabilidades

- Cumplimiento normativo

Objetivos clave:

1. Garantizar confidencialidad, integridad y disponibilidad (CID)
2. Mitigar riesgos OWASP Top 10
3. Cumplir con GDPR, HIPAA (si aplica)

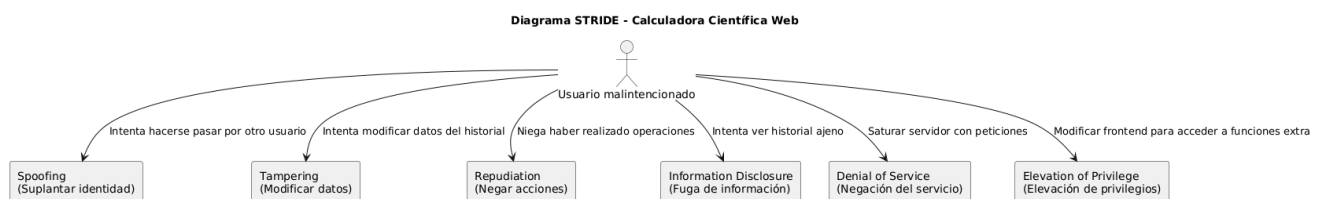
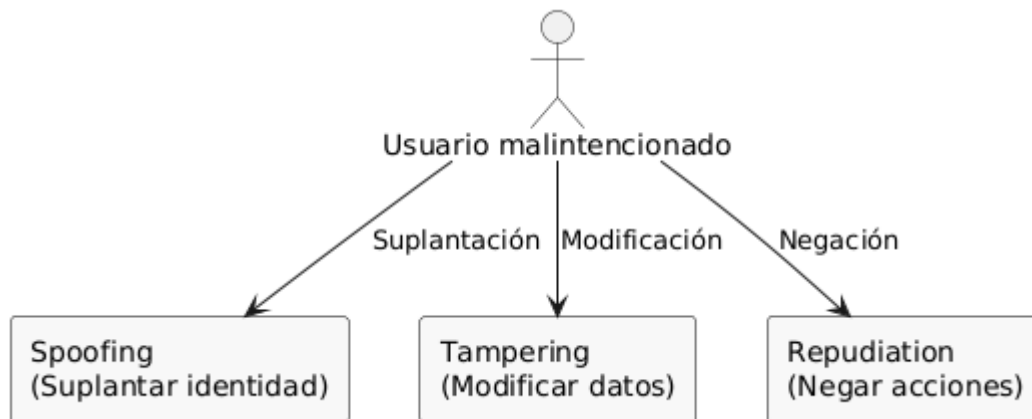
Modelo de Amenazas

Top 5 Riesgos Identificados

Amenaza	Impacto	Probabilidad	Mitigación
Inyección SQL	Alto	Media	Prepared Statements + ORM
XSS	Medio	Alta	Sanitización con DOMPurify
Exposición de APIs	Crítico	Media	Rate Limiting + OAuth 2.0

Diagrama STRIDE

Letra	Amenaza	Descripción
S	Spoofing	Suplantación de identidad
T	Tampering	Alteración o modificación de datos o código
R	Repudiation	Negación de acciones por parte del usuario
I	Information Disclosure	Exposición de datos sensibles
D	Denial of Service	Interrupción del servicio
E	Elevation of Privilege	Escalada de privilegios para acceder a funcionalidades no autorizadas



Spoofing

Tampering

Repudiation

Usuario malintencionado

Suplantar identidad

Modificar datos

Negar acciones

Arquitectura Segura

Capas de Defensa

1. Perímetro: WAF (Cloudflare)

- 2. Aplicación: Validación entrada/salida
- 3. Datos: Encriptación AES-256

Segmentación de Red

```
bash
# Reglas firewall (ejemplo)
ufw allow 443/tcp # HTTPS
ufw deny 22/tcp  # SSH expuesto
```

Control de Accesos

Modelo RBAC

Rol	Permisos
Usuario	Leer/escribir sus operaciones
Admin	CRUD usuarios + ver logs

JWT Implementation

```
javascript
// Ejemplo Node.js
const token = jwt.sign(
  { userId: 123 },
  process.env.JWT_SECRET,
  { expiresIn: '1h' }
);
```

Protección de Datos

Encriptación

Tipo	Tecnología	Uso
Tránsito	TLS 1.3	Todas las comunicaciones
Reposo	AWS KMS	Datos sensibles

Máscara de Datos

```
sql
-- PostgreSQL
CREATE VIEW masked_users AS
```

```
SELECT id,  
       regexp_replace(email, '(.)*(. @)', '\1***\2') AS email  
FROM users;
```

Gestión de Vulnerabilidades

Proceso de Parcheo

1. Escaneo semanal (Trivy + Snyk)
2. Priorización CVSS > 7.0
3. Parches en 72 horas (críticos)

Cumplimiento

Checklist GDPR

Consentimiento explícito
Derecho al olvido
DPO asignado

Logs de Seguridad

```
json  
{  
  "timestamp": "2024-06-20T12:00:00Z",  
  "event": "failed_login",  
  "ip": "192.168.1.100",  
  "userAgent": "Chrome/114"  
}
```

Hardening

Configuración Segura

```
docker  
# Dockerfile  
USER node # No root  
RUN apt-get update && apt-get upgrade -y
```

Headers HTTP

```
nginx  
Copy  
Download  
add_header X-Content-Type-Options "nosniff";  
add_header X-Frame-Options "DENY";
```

Respuesta a Incidentes

Playbook Ejemplo

1. Contención: Aislar sistemas afectados
2. Erradicación: Eliminar malware
3. Recuperación: Restaurar backups

Contactos Clave

Rol	Teléfono
CSIRT	+34 900 000 000

con esto garantizamos:

Protección proactiva contra amenazas. Detección temprana de anomalías.
Cumplimiento regulatorio