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.

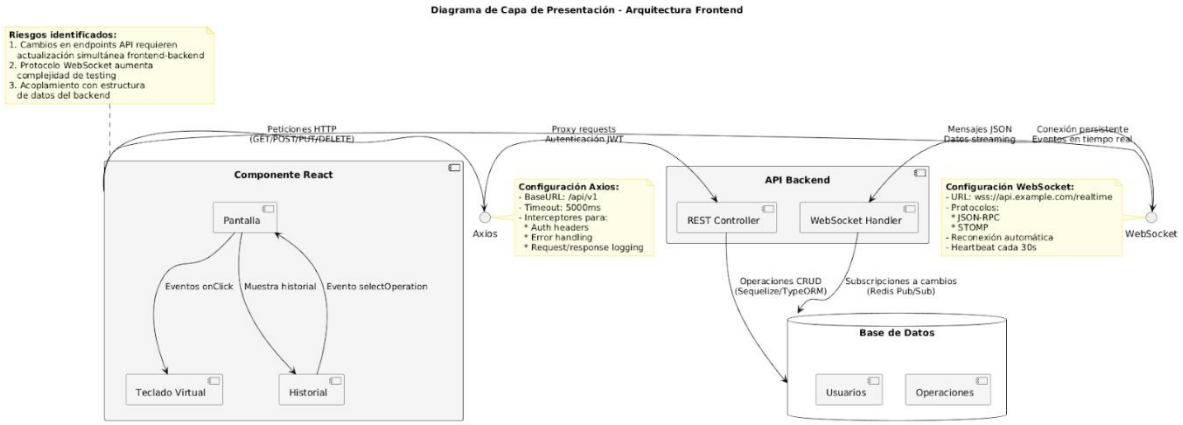
Capa de Lógica (Backend)

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)

```
{
 "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.