1. ARQUITECTURA DEL SISTEMA

1.1 Tecnologías Elegidas

Backend:

Node.js + Express: Framework minimalista y rápido para APIs RESTful. Ideal para prototipos rápidos con amplio ecosistema.

TypeScript: Tipado estático que mejora mantenibilidad y reduce errores en tiempo de desarrollo.

Sequelize ORM: Simplifica la interacción con la base de datos relacional, manejo de migraciones y relaciones entre modelos.

SQLite: Base de datos embebida, ideal para desarrollo local y pruebas. No requiere configuración de servidor externo.

CORS: Middleware para habilitar comunicación entre frontend y backend en diferentes puertos.

Frontend (Propuesto):

Angular 21: Framework completo para SPAs con arquitectura modular, inyección de dependencias y herramientas CLI robustas.

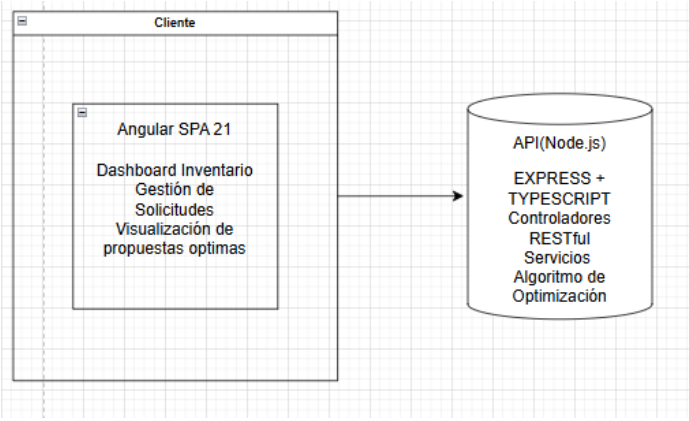
TypeScript: Consistencia con el backend y tipado fuerte para mejor desarrollo.

RxJS: Manejo reactivo de datos, integrado naturalmente con Angular.

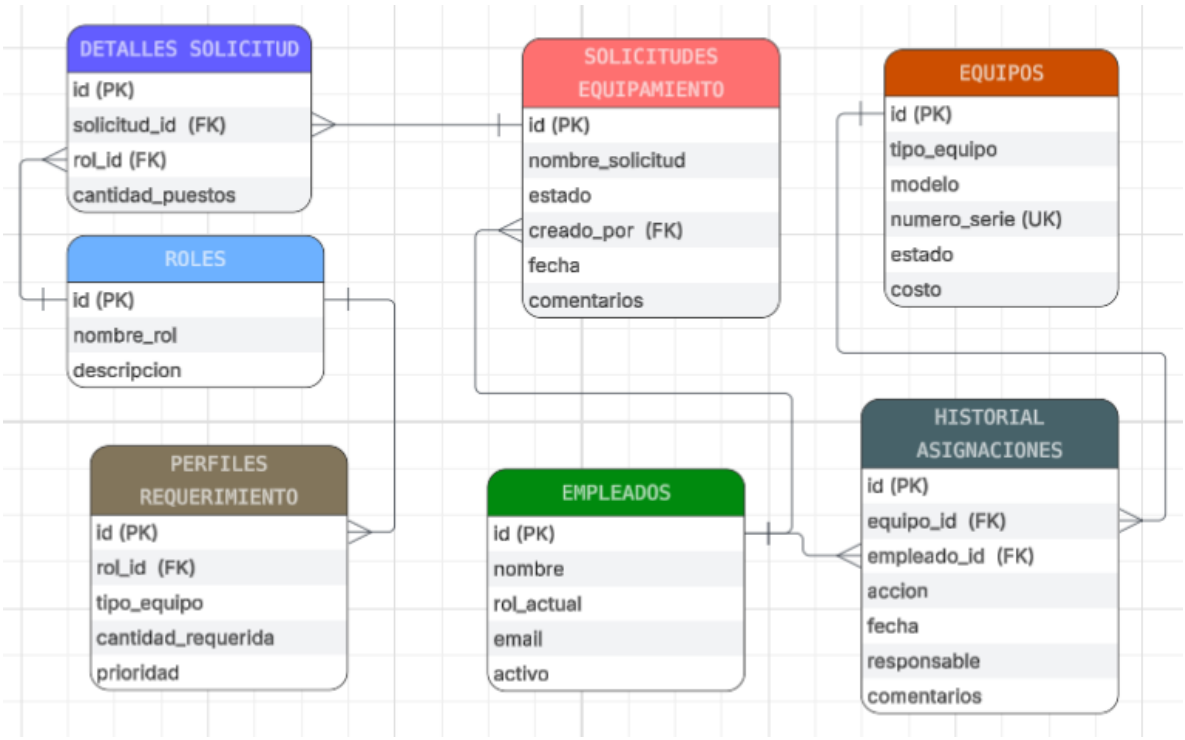
Tailwind: Framework CSS con multiples clases para generar componentes responsivos.

Justificación: Este stack proporciona desarrollo rápido (Node/Express), seguridad de tipos (TypeScript en ambas partes), y una base sólida para escalar (arquitectura modular de Angular).

1.2 Diagrama de Arquitectura



1.3 Diagrama del Esquema de Base de Datos



2. ALGORITMO DE OPTIMIZACIÓN

2.1 Criterio de "Optimalidad" Elegido

Se agregaron 2 datos, uno siendo la prioridad que es para determinar cuan que tan importante es x equipo para un empleado, también se agrego el rendimiento al equipo, para así relacionarlo con la prioridad, si el equipo es muy importante para el rol, el rendimiento del equipo debe ser mejor.

Se implementó un algoritmo híbrido multi-criterio que optimiza:

Maximización de satisfacción de requerimientos: Prioridad × Rendimiento

Minimización de costos relativos: Penalización controlada por parámetro λ

Balance calidad-precio: Equipos con mejor relación rendimiento/costo

Fórmula de Score:

Score = Satisfacción - Penalización por Costo

Satisfacción = (Prioridad/100) × (Rendimiento/100)

Penalización = λ × (Costo/CostoPromedio) × (1 - Prioridad/100)

Donde:

λ = 0.3: Factor que controla el balance entre calidad y economía

Prioridad: 0-100 (de PerfilRequerimiento)

Rendimiento: 0-100 (de Equipo)

CostoPromedio: Calculado dinámicamente por tipo de equipo

2.2 Explicación Paso a Paso del Algoritmo

Fase 1: Preparación de Datos

1. Validar existencia de solicitud

2. Obtener detalles (roles y cantidades)

3. Consultar inventario disponible

4. Calcular costos promedios por tipo

Fase 2: Procesamiento Principal - Triple Bucle Anidado

Para cada ROL en la solicitud (n roles)

Para cada PUESTO del rol (m puestos/rol)

Para cada REQUERIMIENTO del perfil (k requerimientos/rol)

1. Filtrar equipos disponibles del tipo requerido
2. Calcular score para cada equipo candidato
3. Ordenar por score descendente
4. Seleccionar los N mejores (N = cantidad_requerida)
5. Asignar al puesto y actualizar inventario
6. Registrar faltantes si no hay suficientes

Fase 3: Consolidación de Resultados

1. Calcular costo total estimado

2. Determinar mensaje según resultado

3. Retornar propuesta estructurada

2.3 Complejidad Computacional

Análisis Big O:

Preparación de datos: O(E) + O(R) donde E = equipos, R = requerimientos

Procesamiento principal:

$O(n \times m \times k \times E\_filtrado \times \log(E\_filtrado))$

Donde:

n: Número de roles en la solicitud

m: Puestos por rol (promedio)

k: Requerimientos por rol

E_filtrado: Equipos del tipo específico

Caso promedio: $O(N \log N)$ donde N es el número total de equipos procesados

Caso peor: $O(n \times m \times k \times E) \approx O(N^2)$ en el peor escenario

2.4 Posibles Mejoras

Implementación actual (greedy con score)

Posible mejora: Algoritmo genético

Un algoritmo genético podría mejorar la propuesta optima debido a que podría evaluar sobre todos los posibles escenarios ya que re-evalua y ajusta iterativamente. Además, que el orden de los roles en el algoritmo actual, influye en la toma de decisiones. Con algoritmo genético tendríamos una mejor toma de decisiones para asignar equipos

Otra mejora basada en el algoritmo greedy es:

Cache de resultados: Para solicitudes repetidas con mismo inventario

Aprendizaje automático: Ajustar λ según historial de decisiones

Colas de procesamiento: Para solicitudes masivas

Cache distribuido: Redis para inventario frecuentemente consultado

Cabe aclarar que el valor de λ nos dará diferentes resultados, si el valor se aproxima a 1 obtendremos resultados que prioricen el costo, si se acerca más al 0 daremos prioridad al rendimiento. Actualmente dicho valor esta en 0.3 siendo algo balanceado aunque se decanta más por la calidad que por el costo.

3. DECISIONES DE DISEÑO

3.1 Validaciones y Seguridad

Validaciones Implementadas:

1. Nivel Schema (Sequelize)

2. Nivel Servicio

3. Nivel Ruta

Decisiones de Seguridad:

SQL Injection: Prevenida mediante Sequelize (parametrización automática)

XSS: Sanitización de entradas en frontend

CORS: Configuración específica por entorno

Logging: Registro de operaciones críticas (asignaciones, altas/bajas)

3.2 Algoritmo

El algoritmo se eligió en base a los atributos extras esto para obtener un panorama más claro de cual sería “la mejor respuesta”, ya que sería muy sencillo elegir simplemente el de menor costo y asumir que es la mejor opción. Por ello el agregar una prioridad y rendimiento del equipo nos ayuda a entender mejor la necesitad de la solicitud y así generar mejores respuestas basadas en estos puntos. Por otro lado, posterior a agregar dichos datos aumentan las opciones de algoritmos a elegir. En este caso se eligio el greedy debido a que la idea es que podamos tener una relación entre el rendimiento y la prioridad y a su vez, analizar los costos para generar una mejor opción.

3.3 Manejo de errores

Para el manejo de errores tenemos múltiples funciones que notificar algún problema que ocurra en el procesamiento de alguna función, lo cual permite tener una idea de que es lo que esta pasando, porque el sistema no funciona como debería. Cabe aclarar que una herramienta que podría ayudar es usar un sistema de alertas como sweet alerts, las cuales nos indiquen en todo momento el estado del sistema, por cuestiones de tiempo, no se agrego sweet alerts, pero podría ajustarse mediante un componente en angular para recibir solo el título, el tipo de alerta y el texto.