En este documento justificaré las elecciones de **Arquitectura**, **Algoritmo de optimización** y **Decisiones de diseño**.

Empezando con la **Arquitectura**, usamos **React** para el front-end y **ASP.NET** para el back-end. Esta separación permite distribuir responsabilidades dentro del funcionamiento de la aplicación: podemos mejorar la interfaz de usuario sin tocar la API, o mejorar la API sin modificar la UI.

En cuanto a **escalabilidad**, en React se pueden crear **SPA** que cargan rápidamente y actualizan solo los componentes necesarios, mientras que ASP.NET permite manejar muchas peticiones concurrentes, integrar bases de datos robustas y conectarse a servicios externos. El rendimiento es alto gracias a **Kestrel** y la optimización de .NET, y React solo actualiza lo que necesita la UI usando el **DOM virtual**.

En términos de **seguridad**, ASP.NET permite implementar **JWT, OAuth, roles y políticas**, manejando autenticación y autorización de forma centralizada. React se encarga de consumir los datos y puede almacenar tokens de manera segura.

Diagrama de la Arquitectura

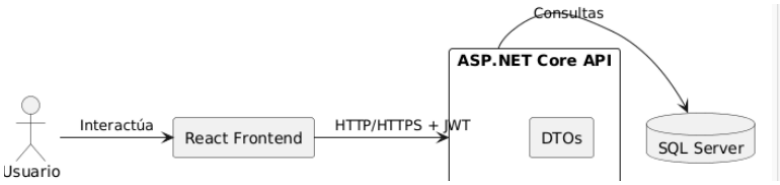
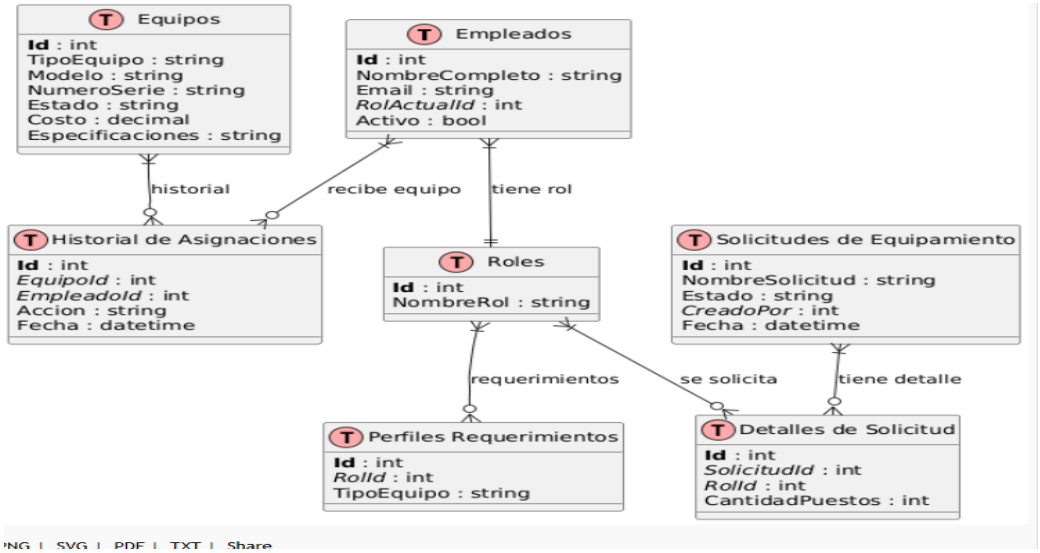


Diagrama del esquema de la base de datos



Algoritmo de Optimización de Asignación de Equipos

Objetivo: Asignar los equipos más adecuados a cada rol y puesto, cumpliendo requerimientos y minimizando costo.

Pasos:

1. **Obtener solicitud** con los roles y cantidad de puestos.
2. **Filtrar equipos disponibles** (`Estado = "disponible"`).
3. **Por cada rol y cada puesto:**
    a. Revisar los requerimientos de tipo de equipo.
    b. Filtrar los equipos candidatos que cumplan el tipo requerido.
    c. Calcular un **score** para cada candidato según CPU, RAM, GPU, almacenamiento y costo.
    d. Seleccionar el equipo con mayor score.
    e. Marcar equipo como asignado (no reutilizable).
    f. Si no hay candidatos, registrar como faltante.
4. **Registrar asignaciones** y calcular el costo total estimado.
5. **Devolver resultado** con: asignaciones, faltantes y costo total.

Criterio de optimalidad: Maximizar score por rol y puesto, priorizando desempeño y costo.

Complejidad aproximada:

$O(R \cdot P \cdot Q \cdot E \log E)$O(R·P·Q·ElogE)

- R = roles
- P = puestos por rol
- Q = requerimientos por rol
- E = equipos disponibles

Posibles mejoras:

El algoritmo podría adaptarse a la relación **costo-beneficio**, ya que muchas empresas buscan minimizar la inversión sin afectar el desempeño. Esto implicaría conocer detalladamente las capacidades de hardware que requiere cada rol dentro de la empresa. Además, con el tiempo se podría construir un historial de

casos de buen funcionamiento de los equipos y basarse en esos datos para optimizar mejor los requisitos por perfil.

Decisiones de diseño

- **Validaciones:** Se validan campos obligatorios al crear equipos o solicitudes (tipo, modelo, número de serie, estado, etc.).
- **Manejo de errores:** Se devuelven mensajes claros con `BadRequest`, `NotFound` o `Unauthorized` según el caso; errores de token o datos nulos se controlan explícitamente, actualmente si no estamos logueados no se cargan los datos o nos redirige al login..
- **Autenticación:** Se usa JWT con `[Authorize]` para proteger rutas de la API.
- **Manejo de roles:** Algunas rutas permiten acceso restringido mediante `[Authorize(Roles = "Admin")]`.
- **Asignación de equipos:** Evita duplicación marcando equipos como asignados y devuelve faltantes si no hay candidatos.