# 🏗️ Manual del Desarrollador - Arquitectura del Sistema

**Proyecto:** PepsiCo Fleet Management System

**Versión:** 1.0.0

**Fecha:** Octubre 15, 2024

**Equipo:** Joaquín Marín & Benjamin Vilches

---

## 📋 Índice

1. [Arquitectura General](#arquitectura-general)
2. [Stack Tecnológico](#stack-tecnológico)
3. [Lenguajes de Programación](#lenguajes-de-programación)
4. [Librerías y Dependencias](#librerías-y-dependencias)
5. [Patrones de Diseño](#patrones-de-diseño)
6. [Estructura del Proyecto](#estructura-del-proyecto)
7. [Flujo de Datos](#flujo-de-datos)
8. [Convenciones de Código](#convenciones-de-código)

---

## 1. Arquitectura General

### 1.1 Visión de Alto Nivel

┌─────────────────────────────────────────────────────────────┐  
│ ARQUITECTURA DEL SISTEMA │  
│ │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ FRONTEND │ │ BACKEND │ │  
│ │ (React) │◄──►│ (Node.js) │ │  
│ │ │ │ │ │  
│ │ • TypeScript │ │ • Express.js │ │  
│ │ • TailwindCSS │ │ • Prisma ORM │ │  
│ │ • Zustand │ │ • JWT Auth │ │  
│ │ • TanStack Query│ │ • RBAC │ │  
│ └─────────────────┘ └─────────────────┘ │  
│ │ │ │  
│ │ │ │  
│ ▼ ▼ │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ BROWSER │ │ DATABASE │ │  
│ │ │ │ │ │  
│ │ • Local Storage │ │ • PostgreSQL │ │  
│ │ • Session Mgmt │ │ • 20 Tables │ │  
│ │ • Cache │ │ • Indexes │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────┘

### 1.2 Patrón Arquitectónico

**Arquitectura en Capas (Layered Architecture)**

┌──────────────────────────────────────┐  
│ Presentation Layer (Frontend) │ ← React Components, Pages, UI  
├──────────────────────────────────────┤  
│ API Layer (Routes) │ ← Express Routes, Middlewares  
├──────────────────────────────────────┤  
│ Business Logic Layer (Services) │ ← Lógica de negocio, validaciones  
├──────────────────────────────────────┤  
│ Data Access Layer (Prisma ORM) │ ← Queries, Transacciones  
├──────────────────────────────────────┤  
│ Database Layer (PostgreSQL) │ ← Almacenamiento persistente  
└──────────────────────────────────────┘

### 1.3 Principios de Diseño

* \*\*Separación de Responsabilidades\*\*: Cada capa tiene una responsabilidad específica
* \*\*Inversión de Dependencias\*\*: Las capas superiores dependen de abstracciones
* \*\*Principio DRY\*\*: No repetir código
* \*\*Principio SOLID\*\*: Código mantenible y extensible
* \*\*Type Safety\*\*: TypeScript en todo el stack

---

## 2. Stack Tecnológico

### 2.1 Backend Stack

Runtime:  
 - Node.js: 20 LTS  
 - TypeScript: 5.5  
  
Framework & Libraries:  
 - Express.js: 4.19 (Web framework)  
 - Prisma: 5.20 (ORM)  
 - JWT: 9.0 (Autenticación)  
 - bcrypt: 5.1 (Hash de passwords)  
 - Winston: 3.11 (Logging)  
 - Helmet: 7.1 (Security headers)  
 - CORS: 2.8 (Cross-origin)  
 - express-rate-limit: 7.1 (Rate limiting)  
   
Development:  
 - tsx: 4.7 (TypeScript execution)  
 - nodemon: 3.0 (Hot reload)  
 - ESLint: 8.57 (Linter)  
 - Prettier: 3.2 (Formatter)

### 2.2 Frontend Stack

Framework:  
 - React: 18.3  
 - TypeScript: 5.5  
 - Vite: 5.4 (Build tool)  
  
UI & Styling:  
 - TailwindCSS: 3.4  
 - PostCSS: 8.4  
 - Autoprefixer: 10.4  
  
State Management:  
 - Zustand: 4.5 (Global state)  
 - TanStack Query: 5.56 (Server state)  
 - zustand/middleware: persist  
  
Routing & Forms:  
 - React Router: 6.26  
 - React Hook Form: 7.53  
 - Zod: 3.22 (Validation)  
  
HTTP & Utils:  
 - Axios: 1.7  
 - date-fns: 3.0 (Date utils)

### 2.3 Database Stack

DBMS: PostgreSQL 15  
ORM: Prisma 5.20  
Hosting: Neon / Supabase / Self-hosted  
Backup: pg\_dump / Neon automatic backups

### 2.4 DevOps Stack

Version Control: Git + GitHub  
CI/CD: GitHub Actions  
Backend Deployment: Railway / Render  
Frontend Deployment: Vercel / Netlify  
Monitoring: UptimeRobot, Sentry (opcional)

---

## 3. Lenguajes de Programación

### 3.1 TypeScript (Principal)

**Uso:** Backend y Frontend

**Versión:** 5.5

**Configuración:** Strict mode habilitado

// Configuración estricta  
{  
 "compilerOptions": {  
 "strict": true,  
 "noImplicitAny": true,  
 "strictNullChecks": true,  
 "noUnusedLocals": true,  
 "noUnusedParameters": true,  
 "noImplicitReturns": true,  
 "noFallthroughCasesInSwitch": true  
 }  
}

**Ventajas:**

* Type safety en tiempo de compilación
* Mejor IntelliSense y autocompletado
* Refactoring más seguro
* Detección temprana de errores

### 3.2 JavaScript (Runtime)

**Uso:** Node.js runtime

**Versión:** ES2022+

**Características:** Async/await, modules, destructuring

### 3.3 SQL (Base de Datos)

**Uso:** Queries de base de datos

**Dialecto:** PostgreSQL

**ORM:** Prisma (genera SQL optimizado)

---

## 4. Librerías y Dependencias

### 4.1 Backend Dependencies

#### Core Dependencies

{  
 "express": "^4.19.2", // Web framework  
 "prisma": "^5.20.0", // ORM  
 "jsonwebtoken": "^9.0.2", // JWT authentication  
 "bcryptjs": "^2.4.3", // Password hashing  
 "winston": "^3.11.0", // Logging  
 "helmet": "^7.1.0", // Security headers  
 "cors": "^2.8.5", // CORS handling  
 "express-rate-limit": "^7.1.5" // Rate limiting  
}

#### Development Dependencies

{  
 "typescript": "^5.5.0", // TypeScript compiler  
 "tsx": "^4.7.0", // TypeScript execution  
 "nodemon": "^3.0.2", // Hot reload  
 "@types/node": "^20.10.0", // Node.js types  
 "@types/express": "^4.17.21", // Express types  
 "@types/bcryptjs": "^2.4.6", // bcrypt types  
 "@types/jsonwebtoken": "^9.0.5", // JWT types  
 "eslint": "^8.57.0", // Linter  
 "prettier": "^3.2.0" // Formatter  
}

### 4.2 Frontend Dependencies

#### Core Dependencies

{  
 "react": "^18.3.0", // UI framework  
 "react-dom": "^18.3.0", // React DOM  
 "react-router-dom": "^6.26.0", // Routing  
 "zustand": "^4.5.0", // State management  
 "@tanstack/react-query": "^5.56.0", // Server state  
 "axios": "^1.7.0", // HTTP client  
 "react-hook-form": "^7.53.0", // Form handling  
 "zod": "^3.22.0", // Validation  
 "date-fns": "^3.0.0" // Date utilities  
}

#### UI Dependencies

{  
 "tailwindcss": "^3.4.0", // CSS framework  
 "postcss": "^8.4.0", // CSS processing  
 "autoprefixer": "^10.4.0" // CSS prefixes  
}

#### Development Dependencies

{  
 "typescript": "^5.5.0", // TypeScript  
 "vite": "^5.4.0", // Build tool  
 "@vitejs/plugin-react": "^4.3.0", // React plugin  
 "@types/react": "^18.3.0", // React types  
 "@types/react-dom": "^18.3.0", // React DOM types  
 "eslint": "^8.57.0", // Linter  
 "prettier": "^3.2.0" // Formatter  
}

### 4.3 Shared Dependencies

{  
 "typescript": "^5.5.0", // TypeScript  
 "zod": "^3.22.0" // Validation schemas  
}

---

## 5. Patrones de Diseño

### 5.1 Patrón MVC (Model-View-Controller)

┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐  
│ VIEW │ │ CONTROLLER │ │ MODEL │  
│ │ │ │ │ │  
│ • React Pages │◄──►│ • Express │◄──►│ • Prisma Models │  
│ • Components │ │ Controllers │ │ • Database │  
│ • UI Logic │ │ • Route Handlers│ │ • Business Logic│  
└─────────────────┘ └─────────────────┘ └─────────────────┘

### 5.2 Patrón Repository

// Service Layer (Repository Pattern)  
export class UserService {  
 async getAll(): Promise<User[]> {  
 return await prisma.user.findMany()  
 }  
   
 async getById(id: string): Promise<User> {  
 return await prisma.user.findUnique({ where: { id } })  
 }  
   
 async create(data: CreateUserData): Promise<User> {  
 return await prisma.user.create({ data })  
 }  
}

### 5.3 Patrón Singleton

// Database connection singleton  
const prisma = new PrismaClient()  
export default prisma

### 5.4 Patrón Factory

// Code generators  
export function generateEntryCode(): string {  
 const date = new Date()  
 const random = Math.floor(Math.random() \* 10000)  
 return `ING-${date.getFullYear()}${date.getMonth()+1}${date.getDate()}-${random.toString().padStart(4, '0')}`  
}

### 5.5 Patrón Observer (React Hooks)

// Custom hook for state management  
function useVehicles() {  
 const [vehicles, setVehicles] = useState<Vehicle[]>([])  
 const [loading, setLoading] = useState(false)  
   
 const loadVehicles = async () => {  
 setLoading(true)  
 const data = await vehicleService.getAll()  
 setVehicles(data)  
 setLoading(false)  
 }  
   
 return { vehicles, loading, loadVehicles }  
}

---

## 6. Estructura del Proyecto

### 6.1 Estructura General

Capstone\_github/  
├── backend/ # API Backend  
│ ├── src/  
│ │ ├── controllers/ # Controladores (10 archivos)  
│ │ ├── services/ # Lógica de negocio (9 archivos)  
│ │ ├── routes/ # Definición de rutas (11 archivos)  
│ │ ├── middlewares/ # Middlewares (5 archivos)  
│ │ ├── utils/ # Utilidades (4 archivos)  
│ │ └── config/ # Configuración (2 archivos)  
│ ├── prisma/  
│ │ ├── schema.prisma # Esquema de base de datos  
│ │ └── seed.ts # Datos de prueba  
│ └── package.json  
├── frontend/ # React Frontend  
│ ├── src/  
│ │ ├── components/ # Componentes reutilizables  
│ │ ├── pages/ # Páginas de la aplicación  
│ │ ├── services/ # Servicios de API  
│ │ ├── store/ # Estado global  
│ │ ├── hooks/ # Custom hooks  
│ │ └── types/ # Tipos TypeScript  
│ └── package.json  
├── shared/ # Código compartido  
│ └── types/  
│ └── index.ts # Tipos compartidos  
└── docs/ # Documentación  
 ├── api/  
 ├── arquitectura/  
 └── casos-uso/

### 6.2 Backend Structure

backend/src/  
├── controllers/ # Capa de Controladores  
│ ├── authController.ts # Autenticación  
│ ├── userController.ts # Gestión de usuarios  
│ ├── vehicleController.ts # Gestión de vehículos  
│ ├── workOrderController.ts # Órdenes de trabajo  
│ └── ...  
├── services/ # Capa de Servicios  
│ ├── authService.ts # Lógica de autenticación  
│ ├── userService.ts # Lógica de usuarios  
│ ├── vehicleService.ts # Lógica de vehículos  
│ └── ...  
├── routes/ # Definición de Rutas  
│ ├── authRoutes.ts # Rutas de autenticación  
│ ├── userRoutes.ts # Rutas de usuarios  
│ └── ...  
├── middlewares/ # Middlewares  
│ ├── auth.ts # Autenticación JWT  
│ ├── rbac.ts # Control de acceso  
│ ├── validation.ts # Validación de datos  
│ └── audit.ts # Auditoría  
├── utils/ # Utilidades  
│ ├── auth.ts # Helpers de autenticación  
│ ├── validation.ts # Validaciones  
│ └── response.ts # Respuestas estándar  
└── config/ # Configuración  
 ├── database.ts # Cliente Prisma  
 └── logger.ts # Configuración Winston

### 6.3 Frontend Structure

frontend/src/  
├── components/ # Componentes Reutilizables  
│ ├── Layout/  
│ │ └── MainLayout.tsx # Layout principal  
│ ├── shared/  
│ │ ├── Card.tsx # Componente Card  
│ │ ├── Button.tsx # Componente Button  
│ │ └── Badge.tsx # Componente Badge  
│ └── PrivateRoute.tsx # Guard de rutas  
├── pages/ # Páginas de la Aplicación  
│ ├── Login.tsx # Página de login  
│ ├── Dashboard.tsx # Router de dashboards  
│ └── dashboards/  
│ ├── AdminDashboard.tsx  
│ ├── GuardiaDashboard.tsx  
│ └── ...  
├── services/ # Servicios de API  
│ ├── api.ts # Cliente Axios  
│ ├── authService.ts # Servicio de autenticación  
│ └── vehicleService.ts # Servicio de vehículos  
├── store/ # Estado Global  
│ └── authStore.ts # Store de autenticación  
├── hooks/ # Custom Hooks  
├── types/ # Tipos TypeScript  
└── utils/ # Utilidades

---

## 7. Flujo de Datos

### 7.1 Flujo de Request HTTP

1. Usuario → Frontend (React)  
 ↓  
2. HTTP Request con JWT → Backend (Express)  
 ↓  
3. Middleware de Autenticación → Valida JWT  
 ↓  
4. Middleware de Autorización → Verifica permisos RBAC  
 ↓  
5. Middleware de Validación → Valida datos de entrada  
 ↓  
6. Controller → Recibe request  
 ↓  
7. Service → Ejecuta lógica de negocio  
 ↓  
8. Prisma ORM → Consulta/modifica base de datos  
 ↓  
9. Database (PostgreSQL) → Retorna datos  
 ↓  
10. Service → Procesa datos  
 ↓  
11. Controller → Formatea respuesta  
 ↓  
12. Middleware de Auditoría → Registra acción  
 ↓  
13. Response → Frontend  
 ↓  
14. Frontend → Actualiza UI

### 7.2 Flujo de Autenticación

1. Usuario ingresa credenciales  
 ↓  
2. Frontend envía POST /api/auth/login  
 ↓  
3. Backend valida credenciales  
 ↓  
4. Backend genera JWT (access + refresh)  
 ↓  
5. Backend retorna tokens  
 ↓  
6. Frontend guarda tokens en localStorage  
 ↓  
7. Frontend incluye token en requests  
 ↓  
8. Backend valida token en cada request  
 ↓  
9. Si token expira, Frontend usa refresh token  
 ↓  
10. Backend genera nuevo access token

### 7.3 Flujo de Estado (Frontend)

1. Usuario interactúa con UI  
 ↓  
2. Component llama a service  
 ↓  
3. Service hace request a API  
 ↓  
4. API retorna datos  
 ↓  
5. Service actualiza store (Zustand)  
 ↓  
6. Store notifica a componentes  
 ↓  
7. Componentes se re-renderizan  
 ↓  
8. UI se actualiza con nuevos datos

---

## 8. Convenciones de Código

### 8.1 Naming Conventions

#### TypeScript

// Clases: PascalCase  
class UserService { }  
class AuthController { }  
  
// Funciones: camelCase  
function getUserById() { }  
function validateEmail() { }  
  
// Constantes: UPPER\_SNAKE\_CASE  
const JWT\_SECRET = 'secret'  
const API\_URL = 'http://localhost:3000'  
  
// Interfaces: PascalCase  
interface User { }  
interface LoginRequest { }  
  
// Enums: PascalCase  
enum UserRole { }  
enum VehicleStatus { }

#### Base de Datos

-- Tablas: snake\_case  
users, work\_orders, vehicle\_entries  
  
-- Columnas: snake\_case  
first\_name, created\_at, updated\_at  
  
-- Índices: idx\_tabla\_columna  
idx\_users\_email, idx\_vehicles\_license\_plate

#### Archivos

// Components: PascalCase.tsx  
LoginPage.tsx, UserCard.tsx  
  
// Services: camelCase.ts  
authService.ts, vehicleService.ts  
  
// Utilities: camelCase.ts  
validation.ts, response.ts

### 8.2 Estructura de Archivos

#### Máximo 300 líneas por archivo

* Si excede, refactorizar en múltiples archivos
* Separar lógica compleja en funciones auxiliares

#### Un archivo por:

* Controller
* Service
* Route
* Component
* Page

### 8.3 Comentarios

#### JSDoc para funciones

/\*\*  
 \* Obtiene un usuario por ID  
 \* @param id - ID del usuario  
 \* @returns Usuario encontrado  
 \* @throws Error si usuario no existe  
 \*/  
async function getUserById(id: string): Promise<User> {  
 // Implementación  
}

#### Comentarios inline

// Validar RUT chileno  
if (!validateRUT(rut)) {  
 throw new Error('RUT inválido')  
}  
  
// TODO: Implementar cache de usuarios  
// FIXME: Corregir validación de email

### 8.4 Imports

#### Orden de imports

// 1. Node modules  
import express from 'express'  
import bcrypt from 'bcryptjs'  
  
// 2. Internal modules  
import prisma from '../config/database'  
import { validateRUT } from '../utils/validation'  
  
// 3. Types  
import type { Request, Response } from 'express'  
import type { User } from '../types'

---

## 9. Herramientas de Desarrollo

### 9.1 IDEs Recomendados

#### VS Code (Recomendado)

// Extensiones esenciales  
{  
 "recommendations": [  
 "ms-vscode.vscode-typescript-next",  
 "bradlc.vscode-tailwindcss",  
 "esbenp.prettier-vscode",  
 "ms-vscode.vscode-eslint",  
 "prisma.prisma",  
 "ms-vscode.vscode-json"  
 ]  
}

#### Configuración VS Code

{  
 "editor.formatOnSave": true,  
 "editor.codeActionsOnSave": {  
 "source.fixAll.eslint": true  
 },  
 "typescript.preferences.importModuleSpecifier": "relative"  
}

### 9.2 Herramientas de Testing

# Backend Testing  
npm install --save-dev jest @types/jest supertest @types/supertest  
  
# Frontend Testing   
npm install --save-dev vitest @testing-library/react @testing-library/jest-dom

### 9.3 Herramientas de Debugging

#### Backend Debug

// launch.json  
{  
 "type": "node",  
 "request": "launch",  
 "name": "Debug Backend",  
 "program": "${workspaceFolder}/backend/src/index.ts",  
 "env": {  
 "NODE\_ENV": "development"  
 }  
}

#### Frontend Debug

* React DevTools (extensión del navegador)
* Redux DevTools (para Zustand)
* Network tab para requests HTTP

---

## 10. Mejores Prácticas

### 10.1 Backend

#### ✅ DO

// Usar transacciones para operaciones múltiples  
await prisma.$transaction(async (tx) => {  
 await tx.user.create({ ... })  
 await tx.auditLog.create({ ... })  
})  
  
// Validar datos antes de guardar  
if (!validateRUT(rut)) {  
 throw new Error('RUT inválido')  
}  
  
// Usar try-catch en controladores  
async function myController(req: Request, res: Response) {  
 try {  
 const result = await myService.doSomething()  
 return sendSuccess(res, result)  
 } catch (error: any) {  
 return sendError(res, error.message)  
 }  
}

#### ❌ DON'T

// No hacer SELECT \* innecesariamente  
const users = await prisma.user.findMany() // ❌  
  
// Mejor: Select específico  
const users = await prisma.user.findMany({  
 select: { id: true, firstName: true, email: true }  
}) // ✅  
  
// No exponer errores detallados en producción  
return res.status(500).json({ error: error.stack }) // ❌  
  
// Mejor: Error genérico en producción  
const message = NODE\_ENV === 'development'   
 ? error.message   
 : 'Error interno del servidor' // ✅

### 10.2 Frontend

#### ✅ DO

// Componentes pequeños y reutilizables  
function UserCard({ user, onEdit }: UserCardProps) {  
 return (  
 <div className="card">  
 <h3>{user.name}</h3>  
 <button onClick={onEdit}>Editar</button>  
 </div>  
 )  
}  
  
// Manejar estados de loading y error  
if (loading) return <LoadingSpinner />  
if (error) return <ErrorMessage error={error} />  
return <DataDisplay data={data} />

#### ❌ DON'T

// No mutar state directamente  
state.items.push(newItem) // ❌  
  
// Mejor: Inmutabilidad  
setState({ ...state, items: [...state.items, newItem] }) // ✅  
  
// No hacer fetch en useEffect sin cleanup  
useEffect(() => {  
 fetch('/api/data').then(...) // ❌  
}, [])  
  
// Mejor: Con cleanup  
useEffect(() => {  
 let cancelled = false  
 fetch('/api/data').then(data => {  
 if (!cancelled) setData(data)  
 })  
 return () => { cancelled = true }  
}, []) // ✅

---

## 11. Troubleshooting Común

### 11.1 Problemas de Conexión

# Error: Cannot connect to database  
# Solución: Verificar DATABASE\_URL  
echo $DATABASE\_URL  
npx prisma db pull  
  
# Error: Port already in use  
# Solución: Matar proceso  
netstat -ano | findstr :3000  
taskkill /PID <pid> /F

### 11.2 Problemas de TypeScript

# Error: Module not found  
# Solución: Reinstalar dependencias  
rm -rf node\_modules package-lock.json  
npm install  
  
# Error: Type errors  
# Solución: Regenerar tipos  
npm run db:generate

### 11.3 Problemas de Build

# Error: Build failed  
# Solución: Limpiar y rebuild  
rm -rf dist  
npm run build  
  
# Error: Memory issues  
# Solución: Aumentar memoria Node  
node --max-old-space-size=4096 node\_modules/.bin/vite build

---

## 12. Recursos Adicionales

### 12.1 Documentación Oficial

* [Node.js](https://nodejs.org/docs)
* [Express.js](https://expressjs.com)
* [Prisma](https://prisma.io/docs)
* [React](https://react.dev)
* [TypeScript](https://www.typescriptlang.org/docs)
* [TailwindCSS](https://tailwindcss.com/docs)

### 12.2 Herramientas Recomendadas

* \*\*Postman/Insomnia\*\*: Testing de API
* \*\*TablePlus\*\*: GUI para PostgreSQL
* \*\*Prisma Studio\*\*: GUI incluida con Prisma
* \*\*Thunder Client\*\*: Extensión VS Code para API

### 12.3 Learning Resources

* [TypeScript Deep Dive](https://basarat.gitbook.io/typescript)
* [React Patterns](https://reactpatterns.com)
* [Node.js Best Practices](https://github.com/goldbergyoni/nodebestpractices)

---

**Última actualización:** Octubre 15, 2024

**Versión:** 1.0.0

**Mantenido por:** Joaquín Marín & Benjamin Vilches