Prompt 1: Arquitectura y Estructura de Datos Completa Prompt Optimizado:
Actúa como un programador experto en JavaScript. Diseña la estructura de datos COMPLETA para un 'Sistema de Reservas' que use localStorage. Necesito: 1. Array de 'Usuarios' con: id, nombre, correo, password, rol (admin/operador/cliente) 2. Array de 'Reservas' con: id, clienteId, fecha, hora, estado 3. Función para inicializar datos SI están vacíos (verificar con if) 4. Funciones para obtener y guardar datos (JSON.parse y JSON.stringify) 5. Función generarNuevoId() que busque el máximo y sume 1 CRÍTICO: Los archivos deben estar en la RAÍZ del proyecto (login.html, registro.html), NO dentro de /pages/. Solo las páginas protegidas van en /pages/. Comenta detalladamente por qué localStorage requiere JSON.stringify/parse.

Resultado: Se creó data.js con arquitectura completa, incluyendo funciones auxiliares para IDs únicos. Leccion:
Lección: Siempre especificar la estructura de carpetas desde el inicio para evitar reorganizaciones.

--

Prompt 2: Sistema de Autenticación con Protección de Rutas Prompt Optimizado:
Crea un sistema de autenticación completo en JavaScript. Debe incluir: 1. Función autenticarUsuario(correo, password) que busque en localStorage 2. Si coincide: guardar usuario en sessionStorage (SIN password por seguridad) 3. Función verificarSesion() que redirija a login si no hay sesión 4. Función protegerPagina(rolesPermitidos) que verifique rol Y sesión 5. Función redirigirSegunRol() para enviar a dashboard o cliente 6. Función cerrarSesion() que limpie sessionStorage IMPORTANTE: Explica la diferencia entre localStorage (persistente) y sessionStorage (temporal) en comentarios detallados.

Resultado: auth.js implementado con todas las funciones de seguridad y redirección según rol. Leccion: Lección: La protección de rutas debe implementarse desde el inicio, no después.

--

Prompt 3: CRUD Completo con Persistencia Garantizada Prompt Optimizado:
Implementa las funciones CRUD para reservas en JavaScript. CRÍTICO: CREATE: crearReserva(clienteId, fecha, hora) - Generar ID único, crear objeto con estado 'pendiente' - Agregar al array con .push() - GUARDAR en localStorage inmediatamente READ: - leerTodasLasReservas() - leerReservasPorCliente(clienteId) usando .filter() - obtenerHistorialCliente(clienteId) ordenado por fecha DESC UPDATE: actualizarEstadoReserva(id, nuevoEstado) - Buscar con .findIndex(), modificar, GUARDAR DELETE: eliminarReserva(id) - Usar .splice(), GUARDAR Comenta CADA uso de JSON.parse y JSON.stringify explicando por qué es necesario.

Resultado: crud.js completo con todas las operaciones y función de historial que faltaba inicialmente. Leccion: Lección: Definir TODAS las operaciones CRUD desde el inicio, incluyendo historial y filtros.

--

Prompt 4: Sistema de JOIN con Explicación Educativa Prompt Optimizado:
Crea obtenerReservasCompletas() que simule un JOIN de SQL: 1. Usar .map() para recorrer cada reserva 2. Dentro del map, usar .find() para buscar el usuario por clienteId 3. Retornar objeto combinado con ...reserva más clienteNombre, clienteCorreo 4. Si no encuentra usuario, usar valores por defecto Incluye comentarios extensos explicando: - Por qué esto es equivalente a SQL JOIN - Ventajas de separar datos (evitar duplicación) - Cómo mejora la UX (mostrar nombres en vez de IDs) - Analogía con bases de datos relacionales Crea también obtenerReservasCompletasPorCliente(clienteId) para filtrar.

Resultado: join.js implementado con comentarios educativos sobre relaciones entre tablas.
Leccion: Lección: El JOIN es crucial para UX pero debe explicarse bien para fines educativos.

--

Prompt 5: Interfaz Cliente con Validaciones y Historial Prompt Optimizado:
Crea la interfaz completa del CLIENTE con Bootstrap 5: HTML (cliente.html): - Navbar con nombre de usuario y botón cerrar sesión - Formulario para crear reserva (fecha, hora) - Tabla de 'Reservas Activas' (solo pendientes y confirmadas) - Tabla de 'Historial Completo' (TODAS las reservas del cliente) JavaScript (cliente.js): 1. protegerPagina(['cliente']) al cargar 2. Validación de fecha: input.min = fecha de hoy (YYYY-MM-DD) 3. Validación de hora: 08:00 - 20:00 4. Doble validación: HTML5 + JavaScript 5. Función cargarReservasCliente() que filtre por usuarioActivo.id 6. Función cargarHistorialCliente() que muestre TODO ordenado por fecha 7. Botón cancelar solo en reservas pendientes Comenta por qué se filtran las reservas (privacidad y seguridad).

Resultado: cliente.html y cliente.js con tabla de historial que faltaba en la primera versión.
Leccion: Lección: Siempre incluir historial completo, no solo reservas activas.

--

Prompt 6: Dashboard Admin/Operador con Botones Diferenciados Prompt Optimizado:
Crea el dashboard para ADMIN y OPERADOR con roles diferenciados: ADMIN puede: - Ver todas las reservas (con JOIN mostrando nombres de clientes) - Cambiar estado: Confirmar, Cancelar (botones verdes/amarillos) - Eliminar permanentemente (botón rojo) - Acceder a Gestión de Usuarios (botón en navbar) OPERADOR puede: - Ver todas las reservas (con JOIN) - Confirmar y Cancelar reservas - Reprogramar fecha/hora (botón azul) - Filtrar reservas de HOY Incluye: 1. Estadísticas en cards (total, pendientes, confirmadas, canceladas) 2. Botones según rol usando if(usuarioActivo.rol === 'admin') 3. Tabla con datos JOIN (clienteNombre, clienteCorreo) 4. Comentarios explicando cómo onclick identifica qué reserva modificar CRÍTICO: Admin debe tener botones para CAMBIAR estados, no solo eliminar.

Resultado: dashboard.html y dashboard.js con botones completos para admin (inicialmente solo tenía eliminar).
Lección: Especificar TODOS los botones y acciones por rol para evitar funcionalidades faltantes.

--

Prompt 7: Filtro de Agenda Diaria con Comparación Exacta Prompt Optimizado:
Implementa filtro de reservas del día actual para OPERADOR: 1. Obtener fecha de hoy en formato YYYY-MM-DD: const año = hoy.getFullYear(); const mes = String(hoy.getMonth() + 1).padStart(2, '0'); const dia = String(hoy.getDate()).padStart(2, '0'); 2. Filtrar reservas donde reserva.fecha === fechaHoy (comparación exacta) 3. Ordenar por hora ascendente para ver agenda cronológica 4. Botón 'Ver Reservas de Hoy' que llame filtrarReservasHoy() 5. Botón 'Mostrar Todas' para volver a vista completa Comenta por qué padStart(2, '0') es necesario (formato consistente 05 no 5) y por qué usamos comparación === en strings de fecha.

Resultado: Función filtrarReservasHoy() con manejo correcto de formato de fechas.
Leccion: Lección: Formato de fechas debe ser consistente (YYYY-MM-DD) para comparaciones exactas.

--

Prompt 8: Gestión Completa de Usuarios para Admin Prompt Optimizado:
Crea sistema COMPLETO de gestión de usuarios (solo ADMIN): Archivo: usuarios.js - crearUsuario(nombre, correo, password, rol) - actualizarUsuario(id, datosNuevos) - eliminarUsuario(id) que también elimine SUS reservas - validarDatosUsuario() con regex de email HTML: gestion-usuarios.html - Tabla con todos los usuarios - Modal Bootstrap para crear/editar - Botones editar y eliminar (NO para admin ID 1) - protegerPagina(['admin']) JavaScript: gestion-usuarios.js - Cargar tabla de usuarios - prepararNuevoUsuario() y prepararEditarUsuario() - Validación de correo único - Confirmación antes de eliminar (alert con advertencia) IMPORTANTE: Eliminar usuario debe eliminar TODAS sus reservas en cascada.

Resultado: Sistema completo de gestión de usuarios que faltaba en requisitos iniciales.
Leccion: Lección: Gestión de usuarios es requisito obligatorio para admin, debe incluirse desde inicio.

--

Prompt 9: Sistema de Registro Público con Validaciones Prompt Optimizado:
Implementa registro público para nuevos CLIENTES: HTML: registro.html (en RAÍZ, no en /pages/) - Formulario con: nombre, correo, password, confirmarPassword - Validaciones HTML5: required, minlength, type='email' - Link a login.html JavaScript: registro.js - registrarCliente(nombre, correo, password) - Verificar que passwords coincidan - Validar correo único (buscar en usuarios existentes) - Crear usuario con rol 'cliente' automáticamente - Usar

función crearUsuario() de usuarios.js - Redirigir a login.html tras registro exitoso Actualizar login.html: - Agregar link 'Regístrate aquí' que apunte a registro.html Comenta por qué se valida en cliente Y servidor (aquí solo cliente por alcance).

Resultado: Sistema de registro completo con validaciones y redirección.
Leccion: Lección: Registro de clientes es requisito obligatorio, debe estar en raíz del proyecto

--

Prompt 10: CSS Profesional sin Problemas de Scroll Prompt Optimizado:
Crea CSS profesional y limpio con estas especificaciones CRÍTICAS: 1. Colores corporativos AZUL (NO morado): --primary-color: #2563eb (azul) --success-color: #10b981 (verde) --danger-color: #ef4444 (rojo) --warning-color: #f59e0b (naranja) 2. NO usar: - background-attachment: fixed (causa problemas de scroll) - Gradientes excesivos (solo en login y botón principal) - Animaciones molestas en fondo - backdrop-filter exagerado 3. Usar: - Fuente: Inter (más profesional que Poppins) - Cards con borde sutil (1px solid) - Sombras suaves (0 1px 3px) - Grises neutros para backgrounds (#f8fafc, #f1f5f9) - Border-radius: 0.5rem a 0.75rem (no muy redondeado) 4. Mantener scroll normal: - body background: var(--gray-50) (color sólido) - Sin position: fixed innecesarios - overflow: auto en tablas 5. Botones sin efectos excesivos: - Hover: translateY(-1px) y sombra - No ripple effects complicados - Transición suave: 0.2s ease Objetivo: Look corporativo, limpio, profesional, sin problemas de usabilidad.

Resultado: CSS profesional con colores azules, sin gradientes excesivos ni problemas de scroll.
Leccion: Lección: CSS debe probarse en scroll, evitar background fixed y efectos que afecten usabilidad.