10. Documento de Arquitectura

Proyecto: Plataforma Web de Gestión de Citas Médicas Multicentro

1. Introducción

La Plataforma de Citas Médicas es una aplicación web que permite a pacientes reservar, modificar y cancelar citas en distintos centros sanitarios, y a médicos y administradores gestionar su actividad diaria desde paneles personalizados. El sistema soporta cuatro roles principales: **paciente, médico, administrador de sistema y administrador de centro**.

Este documento describe, de forma técnica pero entendible, **cómo está construida la plataforma por dentro**: qué piezas tiene, cómo se conectan entre sí y qué decisiones tecnológicas se han tomado.

El objetivo es que cualquier miembro del equipo (y el profesor) pueda leerlo y responder a la pregunta:

"¿Qué partes tiene el sistema y qué hace cada una?"

---

2. Visión general de la arquitectura

A nivel alto, la arquitectura sigue un esquema muy clásico de aplicación web:

- **Capa de presentación (Frontend)**
  Interfaz web en HTML, Tailwind CSS y JavaScript que corre en el navegador del usuario.
- **Capa de lógica de negocio (Backend / API REST)**
  Servidor Node.js + Express que expone endpoints /api/* para autenticación, usuarios, centros y citas.
- **Capa de datos (Persistencia)**
  Base de datos (actualmente ficheros JSON, con migración prevista a MongoDB/Mongoose) donde se almacenan usuarios, citas y centros médicos.

La comunicación entre frontend y backend se hace mediante **HTTP + JSON**: el navegador llama a la API REST, y el backend responde con datos en formato JSON.

A nivel conceptual podemos imaginar tres cajas:

1. **Navegador del usuario** (HTML+JS, dashboards)
2. **Servidor de aplicación** (Node/Express, API REST, seguridad)
3. **Capa de datos** (JSON/MongoDB)

3. Arquitectura del Frontend

3.1. Estructura de archivos

El frontend vive en la carpeta `/web` y está organizado así:

```
/web/
├── css/
│   ├── custom.css              # Estilos personalizados, dark mode, animaciones
│   └── navigation.css          # Estilos de la barra de navegación y menús
├── js/
│   ├── common.js               # Validaciones y utilidades comunes
│   ├── navigation.js           # Navbar, footer, logout, protección básica
│   ├── navigation-config.js    # Configuración de rutas y menús
│   ├── navigation-enhanced.js  # Lógica avanzada de navegación
│   └── api.js / modules/*      # Conexión con la API, dashboards
└── *.html                      # Landing, login y dashboards por rol
```

3.2. Páginas y roles

- **Invitado:**
  `index.html`, `medical_appointment_login_page.html`, `password_recovery.html`, `__faq.html`.
- **Paciente:**
  `patient_dashboard.html`, `book_new_appointment.html`, `online_payment_screen.html`, etc.
- **Médico:**
  `doctor_dashboard.html`, con su propio menú y vistas de agenda.
- **Admin sistema / centro:**
  `administrator_dashboard.html`, `medical_center_dashboard.html`, con gestión de usuarios y centros.

Cada dashboard se genera a partir de plantillas HTML + componentes reutilizables (navbar, sidebar, cards) y se conecta a la API mediante funciones JavaScript que llaman a los endpoints del backend.

3.3. Responsabilidades del frontend

El frontend se encarga de:

- Mostrar la interfaz para cada rol (paciente, médico, admins).
- Validar formularios en el navegador (email, contraseñas, teléfonos, etc.).
- Gestionar el ciclo básico de sesión en el cliente:
  - Guardar el **token JWT** y los datos básicos del usuario en `localStorage`.
  - Verificar si el usuario está autenticado antes de entrar a un dashboard (`protectPage(role)`).

- Consumir la API:
    - Login y registro.
    - Listado y creación de citas.
    - Gestión de centros y usuarios (vía dashboards de admin).

En resumen, el frontend es la cara visible: se ocupa de que la experiencia de usuario sea clara y de traducir clics en **peticiones HTTP** al backend.

---

4. Arquitectura del Backend

4.1. Estructura del proyecto

El backend se encuentra en la carpeta `/backend` y sigue una arquitectura en capas muy típica de Node + Express:

```
backend/
├── src/
│   ├── server.js              # Punto de entrada de la app Express
│   ├── config/
│   │   └── db.js              # Configuración de la base de datos
│   ├── models/                # Modelos de datos
│   │   ├── User.js
│   │   ├── Appointment.js
│   │   └── Center.js
│   ├── controllers/           # Lógica de negocio por recurso
│   │   ├── authController.js
│   │   ├── appointmentController.js
│   │   └── userController.js
│   ├── routes/                # Definición de rutas REST
│   │   ├── auth.routes.js
│   │   ├── appointment.routes.js
│   │   └── user.routes.js
│   ├── middlewares/           # Seguridad y validación
│   │   ├── auth.middleware.js
│   │   └── roles.middleware.js
│   └── utils/                 # Scripts auxiliares (seed, logger, etc.)
├── package.json
└── .env.example
```

La idea es sencilla:

- Las **rutas** reciben la petición (`/api/appointments`, `/api/auth/login`, etc.).
- Los **controladores** implementan la lógica de negocio (crear cita, validar credenciales).
- Los **modelos** definen cómo se guardan los datos.
- Los **middlewares** añaden seguridad (verificar token, rol, validación de cuerpo).

4.2. Endpoints principales

La API expone recursos REST para los conceptos clave del dominio:

- **Autenticación `/api/auth`**
  - `POST /login` – Login de usuario.
  - `POST /register` – Registro de pacientes.
  - `GET /me` – Datos del usuario autenticado.
- **Citas `/api/appointments`**
  - `POST /` – Crear una cita.
  - `GET /` – Listar citas (filtrado según rol).
  - `PATCH /:id/status` – Cambiar estado (pendiente, confirmada, cancelada, completada).
  - `DELETE /:id` – Cancelar una cita.
- **Centros médicos `/api/centers`**
  - `GET /` – Obtener listado de centros.
  - `POST /` – Crear centro (solo admin).
  - `PUT /:id` / `PATCH /:id/status` – Actualizar.
  - `DELETE /:id` – Eliminar centro.
- **Usuarios `/api/users`**
  - CRUD de usuarios, restringido a administradores.

4.3. Modelo de datos básico

El backend trabaja con tres modelos principales: `User`, `Appointment` y `Center`.

Ejemplo simplificado del modelo `User`:

```
{
  name: String,
  email: String,             // único, usado para login
  password: String,          // hasheada con bcrypt
  role: String,              // paciente | medico | admin_sistema | admin_centro
  phone: String,
  ID: String,                // DNI/documento
  healthCard: String,        // Tarjeta sanitaria
  specialty: String,         // solo médicos
  licenseNumber: String,     // solo médicos
  centerId: String,          // centro asignado
  createdAt: Date,
  lastAccess: Date
}
```

El modelo de `Appointment` se corresponde con la entidad **Cita** del diagrama de clases: almacena fecha, hora, tipo, estado y referencias al paciente, al médico y al centro.

---

5. Integración Frontend–Backend

Una parte clave de la arquitectura es cómo se hablan el frontend y el backend.

5.1. Flujo general

1. El usuario abre el navegador y carga una página HTML (por ejemplo `medical_appointment_login_page.html`).
2. El JavaScript de esa página llama a la API mediante `fetch` o funciones de un módulo `api.js`.
3. El backend Express procesa la petición, consulta/actualiza los modelos y devuelve JSON.
4. El frontend interpreta el JSON y actualiza la interfaz (por ejemplo, la lista de citas en el dashboard).

5.2. Ejemplo: Login + Dashboard

1. El paciente introduce su email y contraseña en el formulario de login.
2. El frontend envía `POST /api/auth/login` con las credenciales.
3. El backend valida el usuario, genera un **JWT** y lo devuelve.
4. El frontend guarda el token en `localStorage` y redirige al `patient_dashboard.html`.
5. En el dashboard, antes de mostrar nada, se ejecuta `protectPage('paciente')`:
    - Si no hay token válido → redirige de vuelta al login.
    - Si el rol no coincide → muestra error y hace logout.

5.3. Ejemplo: Reserva de cita

1. Desde el dashboard de paciente, el usuario abre “Nueva Cita”.
2. La página carga el listado de centros y médicos llamando a `/api/centers` y `/api/users?role=medico`.
3. Al confirmar el formulario, el frontend envía `POST /api/appointments`.
4. El backend valida la petición (datos, rol, token) y guarda la cita.
5. El frontend refresca la lista de citas y muestra un mensaje de éxito.

---

6. Seguridad y gestión de sesiones

La seguridad está diseñada en dos niveles: **backend** (el que manda) y **frontend** (para mejorar UX y evitar errores evidentes).

6.1. Autenticación JWT

- Al hacer login, el backend genera un **token JWT** firmado con `JWT_SECRET` y con fecha de expiración.

- En cada petición protegida, el frontend envía el token en el header `Authorization: Bearer <token>`.
- Un middleware `auth.middleware.js` valida el token, lo decodifica y adjunta los datos del usuario a `req.user`.

6.2. Autorización por roles

- Un middleware de roles (`roles.middleware.js`) comprueba si `req.user.role` tiene permiso para acceder al endpoint.
- Ejemplo: solo `admin_sistema` puede borrar usuarios; solo `medico` puede marcar una cita como completada.

6.3. Protección de dashboards

En el frontend:

- La función `protectPage(requiredRole)`:
  - Revisa si el usuario está autenticado.
  - Revisa si su rol coincide con el requerido para esa página.
  - Redirige al login si algo falla.

En el backend:

- **Helmet, CORS, Rate Limiting** y validación con Joi se usan para proteger contra XSS, CSRF básicos, abuso de la API y datos mal formados.

---

7. Despliegue y entornos

7.1. Entorno de desarrollo

- Backend levantado localmente con:
- `cd backend`
- `npm run dev`
- El mismo servidor Express sirve el frontend en `http://localhost:3000`.

7.2. Entorno de producción

La documentación contempla despliegue en **Vercel** o infraestructura similar:

- El backend se despliega como servicio Node/Express.
- El frontend se sirve como contenido estático desde el mismo dominio.
- Las variables sensibles (JWT_SECRET, MONGODB_URI, etc.) se configuran como **variables de entorno** en el panel del proveedor.

---

8. Escalabilidad y líneas futuras

La arquitectura actual cubre la **Fase 1** (funcionalidad crítica) del proyecto, pero está pensada para crecer:

Mejoras previstas:

- Migrar definitivamente la persistencia a **MongoDB Atlas** (en lugar de archivos JSON).
- Añadir módulos de:
  - **Pagos** (`Payment.js`, integración con Stripe/PayPal).
  - **Órdenes médicas** (`MedicalOrder.js`).
  - **Notificaciones** (email/SMS).
- Introducir **testing automático** (unitario y de integración) usando Jest y Supertest.
- Mejorar aún más la observabilidad con logs estructurados y métricas.

---

9. Conclusión

La Plataforma de Citas Médicas se apoya en una arquitectura **web por capas**, simple pero robusta:

- Un **frontend** ligero y centrado en la experiencia de usuario.
- Un **backend** Node/Express responsable de la lógica, la seguridad y el acceso a datos.
- Una **capa de persistencia** que evoluciona hacia MongoDB para soportar más volumen y complejidad.

El documento de arquitectura deja claro qué hace cada pieza y cómo se comunican, facilitando que el resto del equipo pueda seguir construyendo nuevas funcionalidades sin “romper” lo que ya existe.