Justificación tecnológica

Proyecto: Plataforma Web de Citas Médicas Multicentro

---

1. Introducción

En este proyecto hemos intentado elegir tecnologías que fueran:

- fáciles de entender para un equipo de estudiantes,
- modernas y usadas en proyectos reales,
- y suficientes para cubrir todo lo que necesita una plataforma de citas médicas (usuarios, citas, centros, seguridad, etc.).

En otras palabras: no se trata de usar "lo más de moda", sino lo que nos permite **entregar algo funcional y mantenible**en poco tiempo.

---

2. Criterios de selección

A la hora de elegir el stack, hemos seguido estos criterios:

1. **Curva de aprendizaje razonable** – que el equipo pueda aprenderlo sin morir en el intento.
2. **Ecosistema grande** – buena documentación, tutoriales y ejemplos.
3. **Soporte para aplicaciones web modernas** – API REST, JSON, despliegues en la nube, etc.
4. **Facilidad de despliegue** – poder subir el proyecto a un servicio tipo Vercel sin demasiadas complicaciones.

Con esto en mente, el stack elegido es:

- **Backend:** Node.js + Express
- **Base de datos:** MongoDB (Atlas en la nube)
- **Frontend:** HTML5 + Tailwind CSS + JavaScript (sin framework pesado)
- **Seguridad:** JWT + bcrypt
- **Infraestructura:** Vercel + GitHub

---

3. Backend: Node.js + Express

¿Por qué JavaScript también en el servidor?

Usar **Node.js** significa que tanto en el frontend como en el backend usamos **el mismo lenguaje: JavaScript**. Eso tiene ventajas muy prácticas:

- El equipo no tiene que cambiar de “chip mental” entre frontend (JS) y backend (otro lenguaje).
- Podemos **reutilizar conceptos** (promesas, async/await, manejo de JSON).
- Es muy sencillo trabajar con **datos en formato JSON**, que es justo lo que usamos en la API.

Además, Node.js está muy pensado para **aplicaciones web y APIs** que manejan muchas peticiones concurrentes, que es exactamente el caso de una plataforma de citas.

¿Por qué Express?

Express es un framework muy ligero sobre Node.js. Permite:

- Definir rutas como `/api/auth/login`, `/api/appointments`, etc., con muy poco código.
- Organizar la lógica en **controladores y middlewares**, lo que hace el código más limpio.
- Integrar fácilmente librerías de seguridad (Helmet, CORS, rate limiting), validación y logging.

No hemos elegido frameworks más grandes (como NestJS o Spring Boot en Java) para no complicar la arquitectura: **Express es suficiente y mucho más sencillo para un proyecto académico de tamaño medio.**

---

4. Base de datos: MongoDB / MongoDB Atlas

Modelo de datos flexible

En la plataforma tenemos entidades como:

- usuarios (pacientes, médicos, administradores),
- citas,
- centros médicos.

Los datos de estas entidades encajan muy bien en **documentos JSON**, por ejemplo:

```
{
  "name": "María López",
  "email": "maria.lopez@example.test",
  "role": "paciente"
}
```

MongoDB almacena exactamente este tipo de estructuras, sin obligarnos a un esquema rígido de tablas y joins. Eso facilita:

- evolucionar los modelos (añadir campos nuevos sin migraciones complejas),
- trabajar de forma natural con los objetos que ya usamos en el código JavaScript.

MongoDB Atlas en la nube

En lugar de instalar una base de datos en local en un servidor, usamos **MongoDB Atlas**, que es un servicio gestionado:

- se encarga de las copias de seguridad, alta disponibilidad y escalado,
- nos da una URL de conexión (`MONGODB_URI`) que podemos usar directamente en el backend,
- es perfecto para entornos de demo/prototipo como este.

En desarrollo podemos simular la persistencia con **archivos JSON**, pero la arquitectura está pensada para usar una base de datos real en producción.

---

5. Frontend: HTML5 + Tailwind CSS + JavaScript

HTML + Tailwind en vez de un framework grande

Hemos decidido no usar React, Angular o Vue por dos motivos:

1. El proyecto ya es complejo a nivel de funcionalidades (4 roles, varios dashboards).
2. Para la asignatura es más importante **entender bien la arquitectura y los flujos** que pelearse con un framework.

Por eso usamos:

- **HTML5** para la estructura de las páginas,
- **Tailwind CSS** para los estilos, con clases utilitarias que nos permiten maquetar más rápido sin escribir mucho CSS a mano,
- **JavaScript "puro"** para la lógica del cliente (validaciones, llamadas a la API, manejo de la sesión en `localStorage`, etc.).

Este enfoque da como resultado un frontend:

- ligero,
- fácil de desplegar como contenido estático,
- y suficientemente potente para construir dashboards y formularios complejos.

Ventajas de Tailwind

- Diseño moderno con poco esfuerzo (colores, espaciado, responsive).
- Coherencia visual entre todas las páginas.
- No necesitamos mantener un gran archivo CSS lleno de clases personalizadas: la mayoría de estilos son utilidades Tailwind.

---

6. Seguridad: JWT + bcrypt

Para la seguridad hemos optado por dos piezas muy comunes en aplicaciones web modernas:

JSON Web Tokens (JWT)

- Tras el login, el backend genera un **token JWT** firmado con un secreto.

- El token se envía al frontend, que lo guarda (por ejemplo en `localStorage`) y lo manda en el header `Authorization` en cada petición.
- El backend puede verificar la firma del token y saber quién es el usuario sin tener que guardar sesiones en memoria.

Ventajas:

- Encaja muy bien con una **API REST stateless**.
- Es una solución estándar: hay muchas librerías, ejemplos y buenas prácticas.

bcrypt para las contraseñas

- Las contraseñas **no se guardan nunca en texto plano**, sino hasheadas con `bcrypt`.
- Aunque alguien accediera a la base de datos, no podría leer directamente las contraseñas de los usuarios.

En conjunto, JWT + bcrypt nos dan una seguridad razonable para una aplicación académica, y son tecnologías que se usan también en proyectos reales.

---

7. Infraestructura: Vercel + GitHub

GitHub para el control de versiones

- Permite trabajar en equipo, crear ramas, hacer pull requests y revisar cambios.
- Sirve también como "historial" del proyecto y punto central donde está todo el código.

Vercel para el despliegue

- Vercel está muy orientado a proyectos web con frontend estático y backend ligero (serverless o Node).
- Se integra directamente con GitHub: cada push puede generar un despliegue nuevo.
- Simplifica mucho la parte de "montar un servidor": no tenemos que configurar Nginx ni máquinas virtuales.

Esto encaja muy bien con el objetivo de la asignatura: **más tiempo diseñando, programando y probando, menos tiempo peleando con la infraestructura.**

---

8. Alternativas consideradas (y por qué no)

De forma muy resumida:

- **PHP + MySQL**
  Alternativa clásica, pero el equipo ya maneja mejor JavaScript y MongoDB, y el modelo JSON encaja mejor con la API.

- **Java + Spring Boot + PostgreSQL**
  Muy sólido para proyectos grandes, pero la curva de aprendizaje es claramente superior y el setup inicial es más pesado para un proyecto académico de este tamaño.
- **React o Angular en el frontend**
  Dan más potencia para aplicaciones SPA complejas, pero añaden mucha complejidad (build, routing, estado global). Para este trabajo, un frontend con HTML + JS + Tailwind es más que suficiente.

En resumen: hemos preferido un stack **simple, coherente y moderno**, antes que algo muy pesado que solo añadiría dificultad sin aportar mucho en esta fase.

---

9. Conclusión

La combinación de:

- **Node.js + Express** en el backend,
- **MongoDB / MongoDB Atlas** como base de datos,
- **HTML5 + Tailwind CSS + JavaScript** en el frontend,
- **JWT + bcrypt** para la seguridad,
- y **Vercel + GitHub** para el despliegue,

nos da una plataforma que:

- cumple todos los requisitos funcionales de la aplicación de citas médicas,
- es relativamente fácil de entender y mantener por un equipo de estudiantes,
- y utiliza tecnologías muy cercanas a las que se usan hoy en día en proyectos reales.

Es un stack pensado para aprender, pero también para poder evolucionarlo en el futuro hacia una aplicación real si se quisiera.