

# Desarrollo de Aplicaciones Multiplataforma Online



**Florida**

Universitària

## Proyecto Final de Ciclo de Desarrollo de Aplicaciones Multiplataforma Online

**Desarrollo e implementación de  
una aplicación web minimalista  
para la planificación de eventos  
y la gestión de tareas utilizando  
React, Express y PostgreSQL**

**Apellidos y nombre del autor/a:  
PACHECO ROMERALO, DIEGO**

**Curso Académico:  
2024-2025**

**Fecha Entrega:  
21-05-2025**

## ÍNDICE DEL PROYECTO

1. Resumen del Proyecto .....	1
2. Justificación y Objetivos del Proyecto .....	1
2.1. Contexto y Justificación.....	1
2.2. Objetivos específicos .....	2
3. Desarrollo del Proyecto .....	2
3.1. Metodologías utilizadas.....	2
3.2. Tecnologías y Dependencias utilizadas.....	3
3.2.1. Frontend:.....	3
3.2.2. Backend: .....	4
3.2.3. Base de datos: .....	4
3.3. Descripción de los Componentes .....	5
3.3.1. Frontend.....	5
3.3.2. Backend .....	7
3.3.3. Base de datos .....	16
3.4. Problemas y dificultades encontradas en el desarrollo.....	18
3.5. Resultados obtenidos.....	19
4. Conclusiones del Proyecto .....	20
5. Bibliografía y Webgrafía .....	21

## **1. Resumen del Proyecto**

Este proyecto consiste en el desarrollo de una aplicación web minimalista orientada a facilitar la gestión eficiente, intuitiva y directa de eventos mediante un calendario interactivo y listas de tareas. Su principal ventaja radica en la claridad visual y la simplicidad operativa, reduciendo al mínimo las distracciones y evitando integraciones superfluas que generalmente presentan soluciones comerciales existentes en el mercado.

Para adaptarse a distintas necesidades y mejorar la experiencia del usuario, la aplicación cuenta con diversas vistas dinámicas como mensual, semanal, diaria y agenda, destacándose esta última por compactar eficazmente la información de los eventos.

Adicionalmente, ofrece la funcionalidad específica de checklist, especialmente diseñada para el control y seguimiento de tareas puntuales que pueden ser fácilmente marcadas como completadas. La aplicación posibilita la gestión integral de eventos y tareas mediante operaciones de creación, edición y eliminación ágiles y dinámicas, siempre con el objetivo primordial de proporcionar al usuario rapidez, practicidad y eficiencia.

## **2. Justificación y Objetivos del Proyecto**

### **2.1. Contexto y Justificación**

El proyecto nace de la necesidad detectada en usuarios que demandan herramientas digitales para la gestión diaria que sean claras, prácticas y minimalistas, frente a las opciones más extendidas, como Google Calendar, caracterizadas por interfaces saturadas de funciones innecesarias. Esta propuesta se dirige principalmente a usuarios individuales y pequeñas organizaciones que valoran la productividad y la eficiencia por encima de integraciones complejas, buscando una herramienta digital sencilla, directa y eficaz.

Este contexto representa una excelente oportunidad para desarrollar una solución basada en tecnologías modernas, que responda directamente a los requerimientos del mercado actual, ofreciendo un valor añadido significativo mediante una experiencia de usuario mejorada y una gestión más ágil y accesible.

## **2.2. Objetivos específicos**

- Desarrollar una interfaz gráfica intuitiva, accesible, minimalista y de rápida respuesta.
- Garantizar la seguridad robusta mediante el uso avanzado del sistema de autenticación JWT (JSON Web Token).
- Implementar un stack tecnológico actual y escalable utilizando React para el frontend, Express para el backend, y PostgreSQL con Sequelize para la gestión eficiente de la base de datos.
- Adoptar la metodología Atomic Design en el frontend para potenciar la escalabilidad, la reutilización y el mantenimiento de los componentes desarrollados.
- Obtener conocimientos técnicos y prácticos sobre las tecnologías más demandadas en el sector del desarrollo web actual.

## **3. Desarrollo del Proyecto**

### **3.1. Metodologías utilizadas**

Scrum: La metodología ágil Scrum se ha implementado mediante dos sprints claramente definidos, bajo supervisión continua del tutor de prácticas, facilitando así una adaptación rápida y una mejora constante del proceso de desarrollo.

Control de versiones: Se ha utilizado GitHub para mantener controlado el desarrollo del proyecto, permitiendo una evolución organizada, segura y transparente del código fuente.

## 3.2. Tecnologías y Dependencias utilizadas

### 3.2.1. Frontend:

**react / react-dom** Núcleo de la SPA: gestionan el Virtual DOM y el ciclo de vida de los componentes que conforman tu jerarquía atómica.

**react-router-dom v7** Permite navegación cliente sin recarga; define rutas públicas y privadas y controla redirecciones tras el login.

**@mui/material, @mui/icons-material y @mui/x-data-grid** Implementan Material Design (botones, diálogos, iconos y grids) garantizando coherencia visual y accesibilidad desde los átomos hasta las páginas.

**@emotion/react y @emotion/styled** Motor CSS-in-JS que MUI utiliza internamente; permite generar estilos dinámicos y tematizar la aplicación (modo claro/oscuro).

**AXIOS** Cliente HTTP con promesas que encapsulas en AuthService, EventService, TaskService para hablar con la API Express.

**react-hook-form, @hookform/resolvers y yup** Trio para formularios: controla estados y valida esquemas (login, registro, perfil) con feedback instantáneo.

**react-big-calendar** Componente de calendario empresarial que renderiza vistas mes/semana/día/agenda y expone slots y eventos seleccionables.

**react-datepicker** Selector enriquecido de fecha/hora que se integra en el diálogo de creación/edición de eventos.

**date-fns y dayjs** Utilidades ligeras de fechas: date-fns se usa para localizar el calendario; dayjs para cálculos rápidos en filtros de tareas.

**react-dnd y react-dnd-html5-backend** Infraestructura de arrastrar-y-soltar preparada para futuras mejoras (reordenar tareas o mover eventos).

**web-vitals** Registra métricas CLS/FID/LCP en entorno de desarrollo para optimizar la experiencia.

**react-scripts** Conjunto de scripts Create-React-App que compilan, sirven y construyen el bundle usando Babel y Webpack.

**@testing-library/react, @testing-library/dom, @testing-library/user-event y jest-dom** Utilidades de pruebas que permiten montar componentes en memoria, simular interacciones y hacer aserciones de accesibilidad.

### 3.2.2. Backend:

**express** Framework HTTP minimalista sobre Node que define controladores, rutas y middleware de tu API REST.

**cors** Inserta cabeceras CORS para que la SPA (puerto 3000) pueda consumir la API (puerto 5000) durante desarrollo y producción.

**dotenv** Cargas variables sensibles (JWT\_SECRET, credenciales de BD) desde un archivo .env ignorado por Git.

**bcrypt** Algoritmo de hash seguro usado para cifrar contraseñas al registrar y para verificar durante el login.

**jsonwebtoken** Firma y verifica JSON Web Tokens; integra fechas de expiración y se valida en authMiddleware.js.

**pg y pg-hstore** Driver nativo de PostgreSQL y conversor hstore↔JSON; se usan como capa de transporte de Sequelize.

**sequelize + sequelize-cli** ORM que define modelos (User, Event, Task), relaciones y migraciones automatizadas.

**nodemon (dev)** Observa cambios en el código y reinicia el servidor Express, acelerando la iteración.

### 3.2.3. Base de datos:

**PostgreSQL 17.4** Sistema gestor relacional que almacena tablas `users`, `events`, `tasks`; soporta transacciones ACID y consultas SQL avanzadas.

**pg\_dump 17.4** Utilidad CLI empleada para copias de seguridad completas antes de despliegues o refactorizaciones del esquema.

### 3.3. Descripción de los Componentes

#### 3.3.1. Frontend

La adopción de Atomic Design no ha sido un mero capricho metodológico: ha definido cómo se recoge y se combina el naming de cada parte de la interfaz a fin de lograr que el sistema sea escalable y fácil de mantener. Con esta filosofía, la jerarquía visual se crea de forma ascendente, así como una molécula biológica a partir de simples átomos; cada nivel añadido provee contexto, pero nunca viola la encapsulación del previo.

#### Átomos

Los átomos son unidades indivisibles que no tienen conocimiento del resto de la aplicación; su único propósito es exponer props predecibles y estilos coherentes con el tema de MUI.

#### *Ejemplos reales del proyecto*

- **ButtonIcon.jsx** → botón redondo que recibe un ícono Lucide-React y un callback; se reutiliza en calendarios y listas.
- **TextInput.jsx** → envoltorio de `<TextField>` de MUI con validación integrada mediante *react-hook-form*.
- **SwitchDarkMode.jsx** → interruptor que emite un evento global al *ThemeContext* para alternar claro / oscuro.

Al centralizar aquí la tipografía, los colores y la accesibilidad (aria-labels, roles), evitamos “estilos huérfanos” que luego son difíciles de perseguir.

## Moléculas

Cuando dos o más átomos colaboran obtenemos moléculas: pequeñas unidades funcionales con un cometido muy acotado.

- **LoginForm.jsx** (input + password + botón de envío + feedback de error).
- **TaskFilter.jsx** (checkbox pendiente + dropdown orden + badge de contador).

Cada molécula decide únicamente **cómo** combinar sus átomos; el **qué** (lógica de negocio, bussines) queda fuera.

## Páginas

Las páginas son la materialización de rutas React Router. Orquestan organismos, consumen los *contexts* globales y disparan *side-effects* (Axios).

**CalendarPage.jsx** → renderiza la vista de agenda/mes/semana, gestiona diálogos de detalle y alta de eventos, y aplica GlobalStyles para tematizar react-big-calendar.

**TaskList.jsx** → lista dinámica con edición inline, filtro reactivo y diálogo de borrado.

**LoginPage.jsx** → compone Heading, LoginForm y enlaces de navegación; delega la autenticación al AuthContext.

Separar así las páginas permite que el loader de React Router se centre en data fetching y transiciones sin contaminar la jerarquía inferior.

## Gracias al Atomic Desig ganamos:

### 1. Reutilización inmediata

El mismo átomo SubmitButton.jsx aparece en el login, registro, edición de perfil y diálogo de nueva tarea; una sola corrección se propaga a toda la interfaz.

### 2. Consistencia visual

Al no permitir “atajos” de CSS suelto, todos los componentes respetan los *design tokens* del tema MUI y la accesibilidad WCAG.



### 3. Escalabilidad controlada

Nuevas funcionalidades que se añaden creando o extendiendo organismos, sin refactorizar páginas completas.

### 4. On-boarding rápido

Cualquier desarrollador que entre al proyecto puede abrir la carpeta components/atoms y entender la API visual básica antes de tocar servicios o contexto.

En resumen, **Atomic Design** ha sido la columna vertebral que ha alineado desarrollo y diseño, minimizando la deuda técnica y preparando la base para futuras iteraciones (temas, internacionalización, nuevas vistas).

#### 3.3.2. Backend

Nuestro backend está escrito en Node .js 22.14 LTS con el framework Express 4.21; sigue un modelo en capas que separa claramente responsabilidades y facilita la evolución del código sin “efecto dominó”. A nivel de carpeta la estructura por debajo de \backend se compone por:

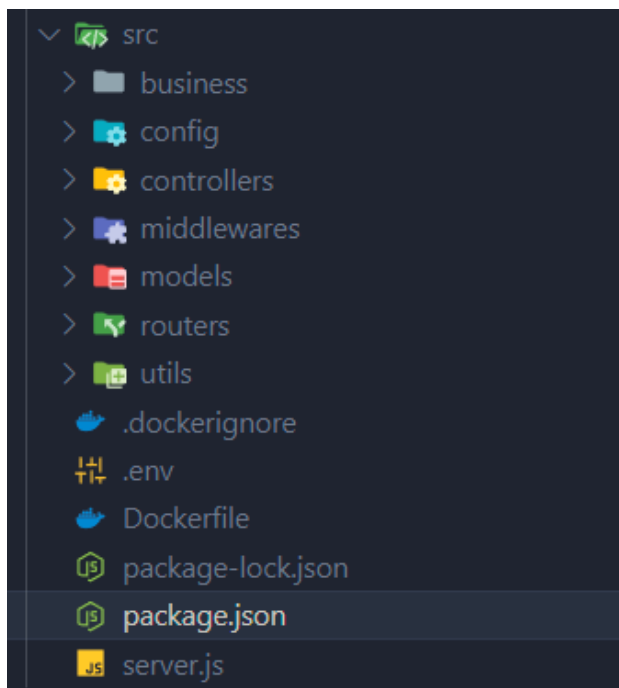


Figura 1: Árbol estructural del backend

#### 1. Capa Router – interfaz HTTP

- Ficheros como *authRoutes.js*, *eventRoutes.js*, *taskRoutes.js*,

*userRoutes.js* exponen los endpoints REST bajo el prefijo */api/\**.

- Cada ruta configura *middlewares* específicos (por ejemplo, *authMiddleware* para endpoints protegidos) y delega la lógica a su **controller** correspondiente.
- El enrutado es puramente declarativo: no contiene ni SQL, ni llamadas a terceros, ni validación compleja



```
authRoutes.js X
backend > src > routers > authRoutes.js > ...
1  const express = require('express');
2  const router = express.Router();
3  const { login, me, register } = require('../controllers/authController');
4  const authMiddleware = require('../middlewares/authMiddleware');
5
6  router.post('/login', login);
7  router.post('/register', register);
8  router.get('/me', authMiddleware, me);
9
10 module.exports = router;
11
```

Figura 2: Contenido de *authRoutes.js*



```
authMiddleware.js M X
backend > src > middlewares > authMiddleware.js > ...
1  const jwt = require('jsonwebtoken');
2
3  const authMiddleware = (req, res, next) => {
4    const authHeader = req.headers.authorization;
5    if (!authHeader) {
6      return res.status(401).json({ message: 'Token not attached' });
7    }
8
9    const token = authHeader.split(' ')[1];
10    if (!token) {
11      return res.status(401).json({ message: 'Invalid token' });
12    }
13
14    try {
15      const decoded = jwt.verify(token, process.env.JWT_SECRET);
16      req.user = decoded;
17      next();
18    } catch (error) {
19      res.status(403).json({ message: 'Expired token or invalid' });
20    }
21  };
22
23  module.exports = authMiddleware;
```

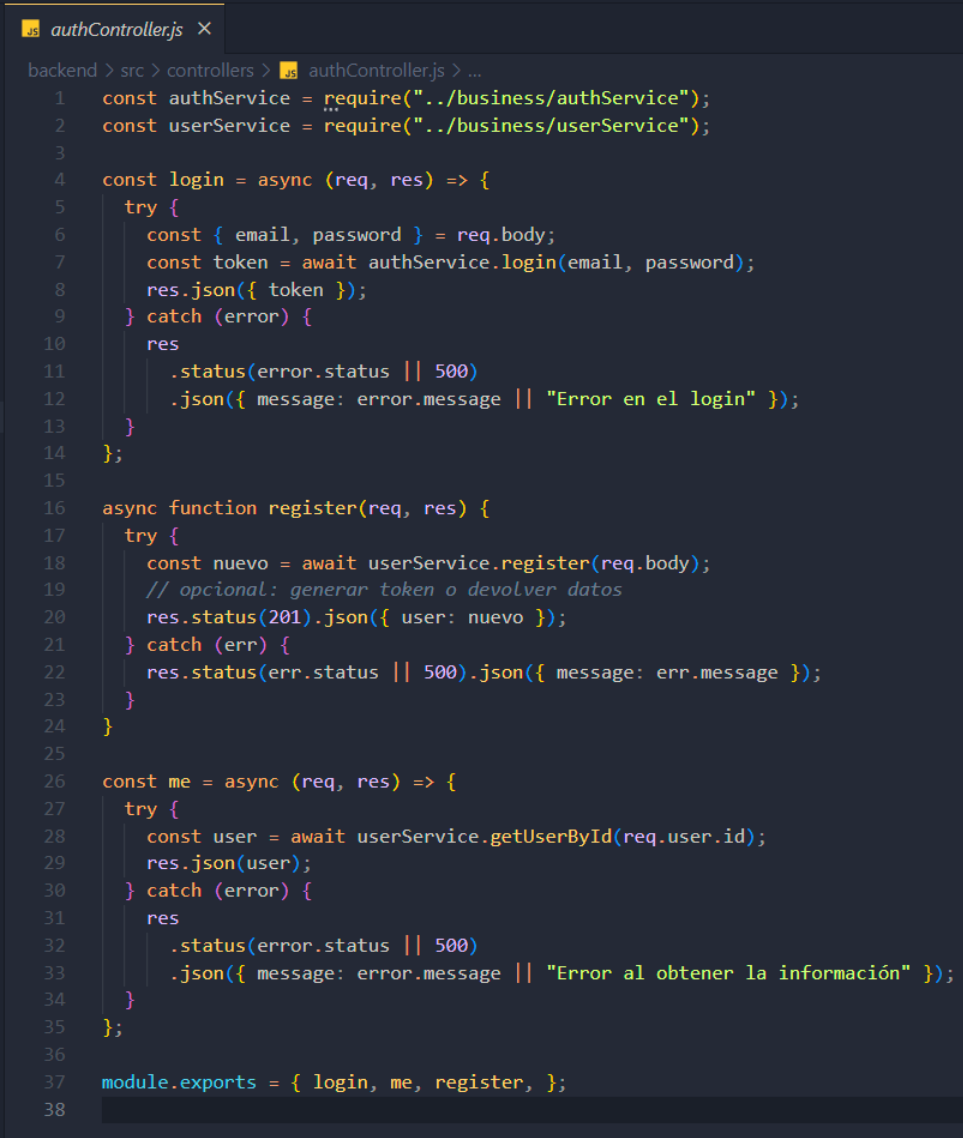
Figura 3: Contenido de *authMiddleware.js*

## 2. Capa Controller – orquestación de la petición

- Recibe el objeto *req*, extrae parámetros / *body* y llama al **service** de

negocio.

- Envuelve todas las llamadas en try/catch; ante un error instancia `createError(status, message)` y lo delega al *error handler* global.
- Convierte la salida de los servicios en respuestas JSON normalizadas:

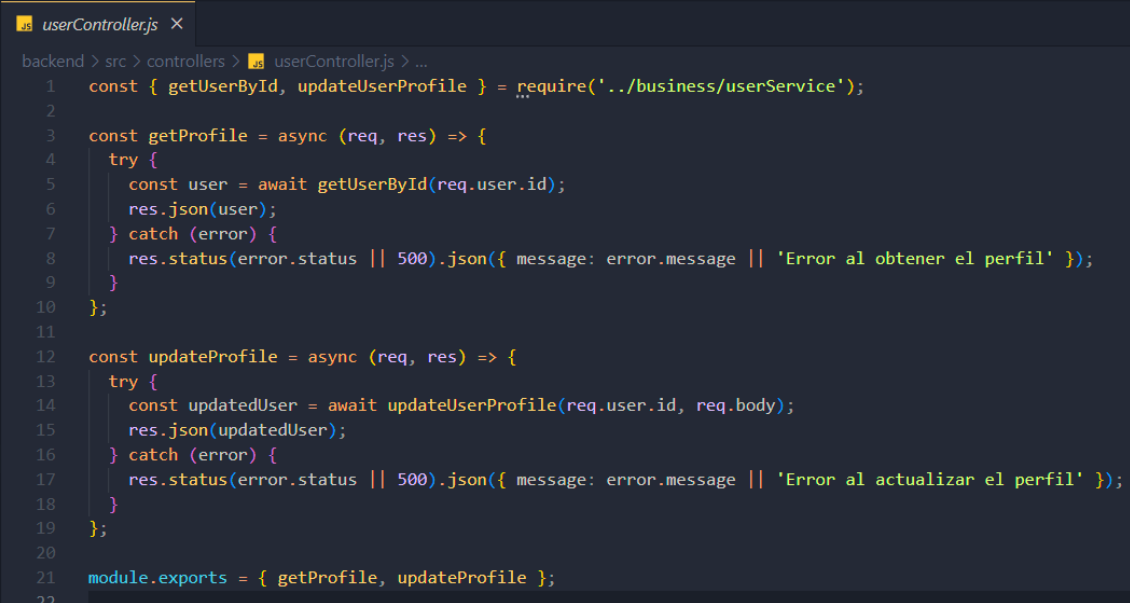


```
authController.js X
backend > src > controllers > authController.js > ...
1  const authService = require("../business/authService");
2  const userService = require("../business/userService");
3
4  const login = async (req, res) => {
5    try {
6      const { email, password } = req.body;
7      const token = await authService.login(email, password);
8      res.json({ token });
9    } catch (error) {
10     res
11       .status(error.status || 500)
12       .json({ message: error.message || "Error en el login" });
13   }
14 };
15
16 async function register(req, res) {
17   try {
18     const nuevo = await userService.register(req.body);
19     // opcional: generar token o devolver datos
20     res.status(201).json({ user: nuevo });
21   } catch (err) {
22     res.status(err.status || 500).json({ message: err.message });
23   }
24 }
25
26 const me = async (req, res) => {
27   try {
28     const user = await userService.getUserById(req.user.id);
29     res.json(user);
30   } catch (error) {
31     res
32       .status(error.status || 500)
33       .json({ message: error.message || "Error al obtener la información" });
34   }
35 };
36
37 module.exports = { login, me, register, };
38
```

Figura 4: Contenido de authController.js

En este caso, el authController se encarga de las llamadas de login, registro y recuperar los datos del usuario.

A su vez, *eventController.js*, *taskController.js* y *userController.js* contienen los métodos CRUD para cada caso, siendo estos listar, crear, actualizar y borrar.



```
1  const { getUserById, updateUserProfile } = require('../business/userService');
2
3  const getProfile = async (req, res) => {
4    try {
5      const user = await getUserById(req.user.id);
6      res.json(user);
7    } catch (error) {
8      res.status(error.status || 500).json({ message: error.message || 'Error al obtener el perfil' });
9    }
10   };
11
12  const updateProfile = async (req, res) => {
13    try {
14      const updatedUser = await updateUserProfile(req.user.id, req.body);
15      res.json(updatedUser);
16    } catch (error) {
17      res.status(error.status || 500).json({ message: error.message || 'Error al actualizar el perfil' });
18    }
19   };
20
21  module.exports = { getProfile, updateProfile };
22
```

Figura 5: Contenido de *userController.js*

En la figura 5 se muestra el contenido de *userController.js*, se pueden apreciar 2 métodos: **getUserById()** y **updateProfile()** los cuales se encargan de listar el usuario filtrando por su ID y actualizar los datos del usuario teniendo en cuenta su ID, de este modo nos aseguramos de que no se editan por error datos de otro usuario de la base de datos.

### 3. Capa Business (Service) – reglas de dominio

- Ficheros como *eventService.js*, *taskService.js*, *authService.js*, *userService.js*.
- Contienen la lógica **transaccional**: comprobaciones de integridad, promoción de eventos, aplicación de políticas (p.e. “un usuario no puede solaparse dos eventos”).
- Se comunican con los **models** Sequelize, nunca con la BD de forma cruda; esto permite mockearlos en tests unitarios sin levantar PostgreSQL.

```
JS eventService.js X
backend > src > business > JS eventService.js > ...
1 // src/business/eventService.js
2 const { Event } = require('../models');
3 const createError = require('../utils/createError');
4
5 async function listEventsByUser(userId) {
6   return await Event.findAll({
7     where: { userId },
8     order: [['start', 'ASC']],
9   });
10 }
11
12 async function createEvent(userId, { title, start, end }) {
13   return await Event.create({ title, start, end, userId });
14 }
15
16 async function updateEvent(userId, id, changes) {
17   const ev = await Event.findOne({ where: { id, userId } });
18   if (!ev) throw createError(404, 'Evento no encontrado');
19   return await ev.update(changes);
20 }
21
22 async function deleteEvent(userId, id) {
23   const ev = await Event.findOne({ where: { id, userId } });
24   if (!ev) throw createError(404, 'Evento no encontrado');
25   await ev.destroy();
26   return;
27 }
28
29 module.exports = {
30   listEventsByUser,
31   createEvent,
32   updateEvent,
33   deleteEvent,
34 };
35
```

Figura 6: Contenido de eventService.js

#### 4. Capa Model – persistencia Sequelize

- Cada tabla del diagrama ER (Users, Events, Tasks) dispone de su \*.js con la definición de atributos, índices y **asociaciones** (User.hasMany(Event)).

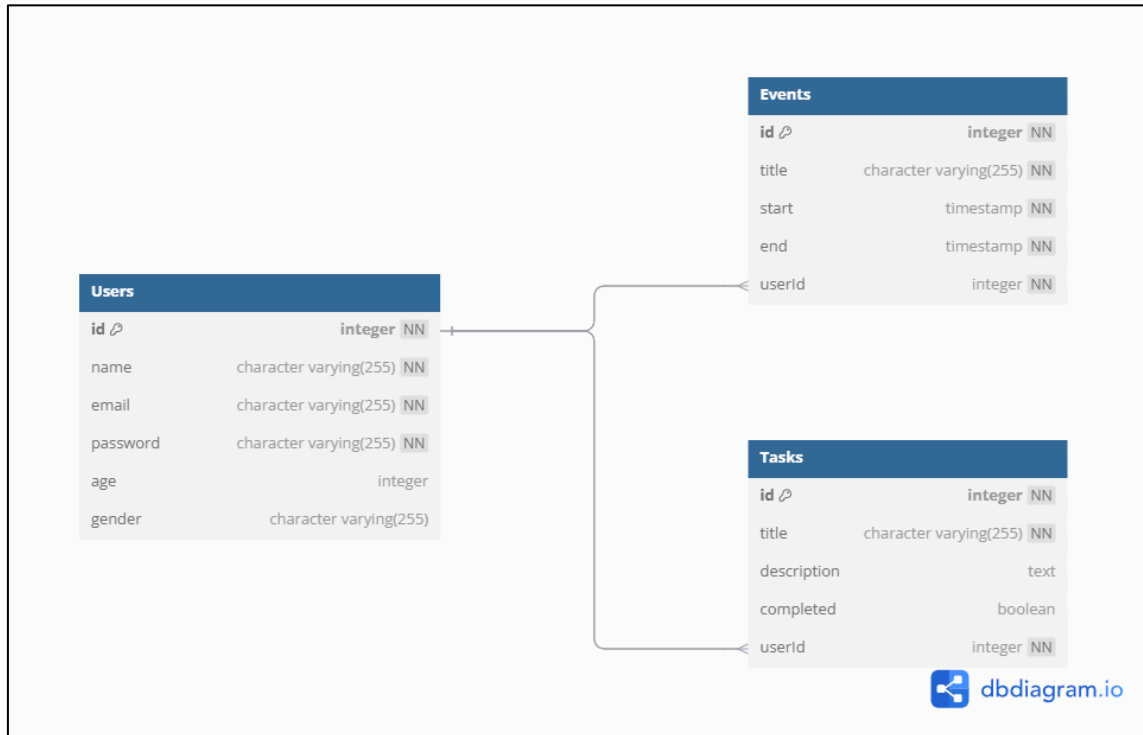


Figura 7: Diagrama ER de la base de datos del proyecto

## 5. Middleware transversal

- **authMiddleware.js** -> Lee el header Authorization: Bearer <token> y verifica la firma con **jsonwebtoken**; si es válido adjunta req.user con el payload del token. Rechaza con 401 si falta o expira.

```
authMiddleware.js M X
backend > src > middlewares > authMiddleware.js > ...
1  const jwt = require('jsonwebtoken');
2
3  const authMiddleware = (req, res, next) => {
4    const authHeader = req.headers.authorization;
5    if (!authHeader) {
6      return res.status(401).json({ message: 'Token not attached' });
7    }
8
9    const token = authHeader.split(' ')[1];
10   if (!token) {
11     return res.status(401).json({ message: 'Invalid token' });
12   }
13
14   try {
15     const decoded = jwt.verify(token, process.env.JWT_SECRET);
16     req.user = decoded;
17     next();
18   } catch (error) {
19     res.status(403).json({ message: 'Expired token or invalid' });
20   }
21 };
22
23 module.exports = authMiddleware;
```

Figura 8: Contenido de authMiddleware.js

## 6. Autenticación y gestión de sesión con JWT

### 1. Login (POST /api/auth/login)

- El controller valida credenciales; si son correctas genera un token:

```
const token = jwt.sign(
  { id: user.id, email: user.email },
  process.env.JWT_SECRET,
  { expiresIn: "12h" }
);
```

Figura 9: Demostración de la firma de credenciales del token

- Devuelve { token } (véase la captura Postman).

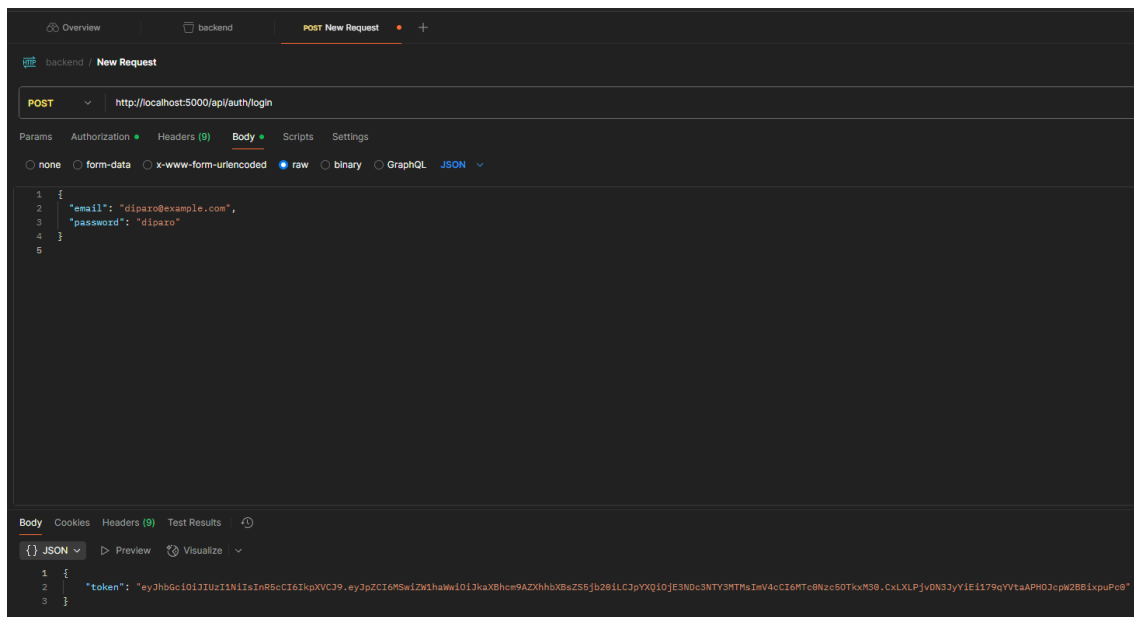


Figura 10: Petición POST para obtener el Token JWT

## 2. Registro (POST /api/auth/register)

```
async function register({ name, email, password, age, gender }) {  
  // 1) Evitar duplicados  
  const exists = await User.findOne({ where: { email } });  
  if (exists) {  
    throw createError("El email ya está registrado", 409);  
  }  
  
  // 2) Encriptar contraseña  
  const hash = await bcrypt.hash(password, 10);  
  
  // 3) Crear usuario  
  const user = await User.create({  
    name,  
    email,  
    password: hash,  
    age,  
    gender,  
  });  
  
  const { password: __, ...userData } = user.toJSON();  
  return userData;  
}
```

Figura 11: Función asíncrona del método para el registro



## 7. Configuración y variables de entorno

- .env guarda secretos (JWT\_SECRET, DB\_PASS, PORT) y se carga con **dotenv** antes de arrancar Express.

```
backend > .env
1  DB_HOST=localhost
2  DB_PORT=5432
3  DB_USER=diparo
4  DB_PASSWORD="diparo"
5
6  DB_NAME=backendtask
7  JWT_SECRET=Jwt@token_gen_dunno123712
8  PORT=5000
9  |
```

Figura 12: Variables del entorno del proyecto

- El archivo `src/config/database.js` lee dichas variables y crea la instancia Sequelize para ser inyectada en los modelos.

```
backend > src > config > database.js > ...
1  require('dotenv').config();
2  const { Sequelize } = require('sequelize');
3
4  const sequelize = new Sequelize(
5    process.env.DB_NAME,
6    process.env.DB_USER,
7    process.env.DB_PASSWORD,
8    {
9      host: process.env.DB_HOST,
10     port: process.env.DB_PORT,
11     dialect: 'postgres',
12     logging: false,
13   }
14 );
15
16 module.exports = sequelize;
17
```

Figura 13: Fichero para la creación de la instancia de Sequelize

- El fichero config/config.json (requerido por *sequelize-cli*) permite ejecutar `npx sequelize-cli db:migrate` en local o CI.

```
backend > src > config > {} config.json > ...
1  {
2    "development": {
3      "username": "diparo",
4      "password": "diparo",
5      "database": "backendtask",
6      "host": "127.0.0.1",
7      "dialect": "postgres"
8    },
9    "test": {
10     "username": "diparo",
11     "password": "diparo",
12     "database": "backendtask",
13     "host": "127.0.0.1",
14     "dialect": "postgres"
15   },
16   "production": {
17     "username": "root",
18     "password": null,
19     "database": "database_production",
20     "host": "127.0.0.1",
21     "dialect": "postgres"
22   }
23 }
24
```

Figura 13: Contenido del fichero de configuración para *sequelize-cli*

### 3.3.3. Base de datos

Para la persistencia usamos PostgreSQL 17.4 como motor relacional y Sequelize 6.37 como ORM. El objetivo es mantener coherencia referencial y, al mismo tiempo, permitir una evolución rápida del esquema durante la fase de desarrollo.

Esquema y relaciones principales:

- **Users** – credenciales y metadatos del usuario (email único, contraseña hasheada con bcrypt).
- **Events** – título, fecha inicio, fecha fin, FK userId → Users.id.
- **Tasks** – título, descripción, estado completed, FK userId.

Las relaciones son 1-N; las claves foráneas están definidas con ON DELETE CASCADE para evitar registros huérfanos.

En la Figura 7 se hace referencia al esquema ER de la base de datos.

En lugar de mantener migraciones manuales { **alter: true** } inspecciona la tabla real y genera los ALTER TABLE necesarios (crear/renombrar columnas, añadir FK, etc.).

```
sequelize.sync({alter:true})
  .then(() => {
    console.log('Database connected successfully');
    app.listen(PORT, () => {
      console.log(`Server running at port ${PORT}`);
    });
  })
  .catch(err => {
    console.error('Database connection error :', err);
  });
```

Esto nos da feedback inmediato cuando modificamos un modelo: no existen pasos extra de sequelize-cli migrate.

En síntesis, la combinación **PostgreSQL 17.4 + Sequelize sync(alter)** nos ofrece la robustez de un SGBD relacional y la velocidad de iteración de alto rendimiento, sin sacrificar consistencia ni claridad en la aplicación.

### 3.4. Problemas y dificultades encontradas en el desarrollo

- **React-Big-Calendar** y el dilema “construir o integrar”:

Al arrancar el proyecto planteé diseñar un componente de calendario propio en React para tener control absoluto del marcado semántico y el estilo. Llegué a prototipar un grid mensual (se muestra en el vídeo del segundo sprint), pero pronto aparecieron los problemas por falta de experiencia:

- Gestión de zonas horarias y cambio de horario de verano — recrear la lógica que ya resuelven librerías consolidadas resultaba poco realista en el calendario escolar del PFC.
  - Tras evaluar alternativas (FullCalendar, DevExtreme Scheduler) opté por React-Big-Calendar por ser MIT, ligero y basado en “date-fns”. No obstante, la librería impone clases CSS internas “rbc-\*” y un sistema de plantillas algo rígido. Para adecuarla al estilo minimalista invertí horas en:
  - Reescribir las reglas Sass → MUI GlobalStyles para sobrescribir selectivamente colores y bordes.
  - Crear un CustomToolbar que sustituyera el toolbar nativo y expusiera botones traducidos (Hoy, Ant., Sig.) en castellano.
  - Migrar a la vista Agenda porque por defecto no mostraba horas de manera compacta.
- **Adopción simultánea de tecnologías nuevas (React 19, JWT, Sequelize)**

Venía de un contexto académico basado en Java Swing y JDBC; pasar a un stack **full-stack JavaScript** implicó una curva pronunciada

**Actualización MUI 6 → 7** — justo a mitad del sprint MUI publicó la v7; al ejecutar npm i se actualizó automáticamente y rompió:

Imports antiguos @mui/styles quedaron obsoletos.

Las dependencias internas de MUI cambiaron versiones peer, forzando a reinstalar @emotion/\* y @mui/icons-material.

### 3.5. Resultados obtenidos

- **Inicio de sesión y seguridad confiables**  
El sistema de registro y acceso funciona de manera fluida. Cada usuario inicia sesión con total seguridad y las áreas privadas quedan protegidas de forma automática, sin que se perciban pasos adicionales ni contratiempos.
- **Gestión completa de eventos y tareas**  
Crear, consultar, actualizar o eliminar un evento del calendario, así como añadir y marcar tareas, se realiza con pocos clics y sin demoras perceptibles. La experiencia recuerda a las aplicaciones profesionales, pero con la ligereza de una herramienta hecha a medida.
- **Calendario interactivo adaptable a cada necesidad**  
La aplicación ofrece cuatro formas de ver la agenda: vista mensual para planificar a largo plazo, vista semanal para organizar la semana de un vistazo, vista diaria para el detalle de la jornada y una práctica vista agenda que reúne todo lo importante de forma compacta.
- **Lista de tareas rápida y directa**  
El checklist resulta muy cómodo para anotar quehaceres puntuales y marcarlos como completados al instante. El filtro “solo pendientes” ayuda a centrarse en lo que queda por hacer, evitando distracciones.
- **Almacenamiento sólido y ordenado**  
La información de usuarios, eventos y tareas se guarda en una base de datos relacional que mantiene todo bien relacionado y sin inconsistencias. De cara al usuario final, esto se traduce en que los datos siempre aparecen correctos y disponibles.

- **Interfaz clara y moderna**

La combinación de componentes visuales actuales y un diseño minimalista hace que la aplicación se sienta familiar y agradable. Los colores se adaptan a modo claro u oscuro, los botones son accesibles y el aspecto general transmite profesionalidad sin sobrecargar la pantalla.

En conjunto, el proyecto cumple los objetivos planteados: brinda una forma sencilla y agradable de planificar el tiempo, gestionar tareas y mantener la información segura, todo ello sin la complejidad habitual de las plataformas comerciales más grandes.

#### **4. Conclusiones del Proyecto**

Realizar esta aplicación ha sido mucho más que escribir código: ha supuesto un recorrido completo por todo el ciclo de vida de un producto digital, desde la idea inicial hasta una versión plenamente funcional. Trabajar con un *stack* moderno (React en el navegador, Express en el servidor y PostgreSQL como base de datos) me obligó a abandonar la zona de confort y afrontar conceptos totalmente nuevos —autenticación con tokens, diseño de componentes reutilizables, organización modular— que hoy ya forman parte de mi repertorio profesional.

La experiencia también me enseñó la importancia de los **detalles “invisibles”** de un proyecto: llevar control de versiones de forma disciplinada, documentar decisiones de arquitectura y escribir mensajes de *commit* comprensibles. Todo ello refuerza la calidad del resultado y, sobre todo, hace que cualquier colaborador pueda incorporarse sin fricciones.

En lo tecnológico, domino ya la construcción de una SPA con rutas protegidas, conozco los fundamentos de la seguridad básica en web y sé modelar datos relacionales de manera coherente. En lo humano, he desarrollado una mayor tolerancia al error —propio y ajeno— y una mentalidad de mejora continua: cada obstáculo, desde un componente que se resiste a un “bug” de última hora, es ahora una oportunidad para aprender algo nuevo.

Mirando hacia delante, el proyecto se convierte en un trampolín. La base está establecida: puedo añadir autenticación social con OAuth, empaquetar todo en contenedores Docker o escalar la aplicación para varios equipos sin reescrituras profundas. Más importante aún, dispongo de un método de trabajo y un conjunto de competencias —técnicas y blandas— que se adaptan bien a los entornos de desarrollo ágiles y colaborativos que dominan el mercado actual.

En síntesis, el camino recorrido reafirma mi vocación por el desarrollo *full-stack* y me deja una lección clave: la combinación de curiosidad, disciplina y trabajo iterativo es la mejor garantía de éxito cuando se aborda cualquier reto tecnológico.

## 5. Bibliografía y Webgrafía

Meta Platforms, Inc. (2024). *React documentation*. Recuperado el 20 de mayo de 2025, de <https://react.dev/learn>

Auth0. (s. f.). *Introduction to JSON Web Tokens*. JWT.io. Recuperado el 20 de mayo de 2025, de <https://jwt.io/introduction>

StrongLoop & Express Contributors. (2024). *Using middleware*. En *Express.js documentation*. Recuperado el 20 de mayo de 2025, de <https://expressjs.com/en/guide/using-middleware.html>

PostgreSQL Global Development Group. (2024). *PostgreSQL 17.4 documentation*. Recuperado el 20 de mayo de 2025, de <https://www.postgresql.org/docs/current/index.html>

MUI Team. (2024). *Material UI: All components*. Recuperado el 20 de mayo de 2025, de <https://mui.com/material-ui/all-components/>

Rowan, D. (2024). *Sequelize v6 documentation*. Sequelize. Recuperado el 20 de mayo de 2025, de <https://sequelize.org/docs/v6/>