Eclipse

Aplicación de consulta meteorológica con React y Node

DWEC

Cristina Martín-Fontecha Blázquez

1. DESCRIPCIÓN DEL PROYECTO

En este proyecto consiste en crear una aplicación web que permite consultar la predicción meteorológica utilizando la API de AEMET. La aplicación está dividida en dos partes:

- **Backend**, desarrollado con Node.js y Express, que se encarga de comunicarse con la API de AEMET, procesar los datos y exponer rutas propias hacia el frontend.
- **Frontend**, desarrollado con React, que ofrece dos métodos de búsqueda (por municipio mediante entrada manual, y por provincia y municipio mediante desplegables). Una vez seleccionado el municipio, la aplicación muestra la predicción diaria desde el día actual hasta seis días más. Además, para los días que lo permiten, también se puede ver la predicción por horas.

2. INSTRUCCIONES DE INSTALACIÓN Y EJECUCIÓN

2.1. Backend

Para el backend parto directamente de la plantilla. Esta plantilla ya incluye:

- Node configurado
- package.json con dependencias
- Middlewares como cors y express.json
- Ejemplos de endpoints
- La estructura básica del proyecto

Por este motivo no necesito ejecutar `npm init` ni instalar dependencias adicionales. Simplemente copio la plantilla, la renombro como mi proyecto y ejecuto:

```
npm install
```

Con esto se instalan las dependencias incluidas en la plantilla y ya puedo añadir mis propios endpoints y la lógica necesaria para consultar la API de AEMET.

Para arrancar el backend utilizo:

```
node server.js
```

2.2. Frontend

Para el frontend creo una carpeta nueva dentro del proyecto llamada frontend. Dentro de ella genero un proyecto React con Vite:

```
cd frontend
npm create vite@latest
```

Elijo:

- Plantilla: **React**
- Lenguaje: **JavaScript**

Después, instalo *React Router*, que necesito para gestionar la navegación entre páginas:

```
npm install react-router-dom
```

Para arrancar el frontend ejecuto:

```
npm run dev
```

Así, puedo comprobar que tanto el backend como el frontend funcionan correctamente.

2.3. Postman para comprobaciones

Para probar las rutas del backend utilizo Postman. Lo descargo desde la página oficial e instalo la versión para Windows. Aunque al principio no pruebo nada, me resulta útil más adelante para comprobar que los endpoints devuelven los datos esperados antes de conectarlos con el frontend.

3. FUNCIONALIDAD DE LA APLICACIÓN

Decido incluir dos métodos de búsqueda en la aplicación, ya que son los más prácticos para mi gusto. En un principio pensé en añadir un tercer método basado en código postal, pero comprobé que los identificadores de los municipios en AEMET no coinciden con los códigos postales, así que descarto esa opción.

3.1. Búsqueda manual por municipio

En esta opción el usuario escribe el nombre del municipio y pulsa el botón “Aceptar”.

El frontend envía ese nombre al backend, que busca el municipio en el listado oficial de AEMET.

Este método funciona, pero es muy exacto: si el usuario no escribe el nombre tal cual aparece en la API, no se encuentra.

Por ese motivo añado un segundo método más cómodo.

3.2. Búsqueda con desplegables

En esta opción el usuario selecciona primero una provincia.

Después, el segundo desplegable se rellena automáticamente con los municipios pertenecientes a esa provincia.

Aquí me encuentro con un problema: La API de AEMET en /api/maestro/municipios no relaciona directamente municipios con provincias. Pero, cada municipio tiene un campo id_old, cuyos dos primeros dígitos coinciden con el código de la provincia.

Para solucionarlo:

i. Creo un archivo provincias.js en el backend con los códigos de provincia.
ii. Creo un archivo provincias.json en el frontend para rellenar el desplegable de provincias.
iii. Filtro los municipios en el backend usando los dos primeros dígitos de id_old.

3.3. Predicción diaria

Cuando el usuario pulsa “Aceptar” en cualquiera de los dos métodos de búsqueda, lo redirijo a /prediccion-dias.

En esta página muestro una tabla horizontal con el día actual seguido de los 6 días siguientes.

La API de AEMET divide algunos días en varias franjas horarias (00–24, 00–06, 06–12, etc.).

Yo decido usar la franja 00–24, que representa el día completo.

En algunos casos esta franja aparece vacía. Para solucionarlo, en el backend selecciono la primera franja disponible, ya que no es posible hacer una media de valores como el estado del cielo.

La página incluye un botón “Volver” para regresar al inicio.

3.4. Predicción por horas

En los días que lo permiten (solo los primeros días tienen predicción horaria), aparece un botón dentro de la tabla de predicción diaria que lleva a la página /prediccion-horas.

En esta página muestro una tabla con:

- Las horas del día
- El estado del cielo
- La temperatura
- La precipitación
- El viento

También incluyo un botón “Volver” para regresar a la predicción diaria.

4. DECISIONES TÉCNICAS

En este apartado voy explicando las decisiones que voy tomando mientras desarrollo el proyecto. Mi intención es dejar claro porqué elijo unas soluciones y cómo voy conectando el backend con el frontend para que todo funcione de forma coherente.

4.1. Organización del Backend

Mantengo todos los endpoints dentro de server.js, igual que en la plantilla.

Para evitar que server.js tenga mucho código, creo una carpeta **/funciones**, donde guardo funciones específicas que procesan los datos de AEMET. Esto hace que el código sea más modular y fácil de mantener. Las funciones que utilizo son:

- **obtenerMunicipios.js**: descarga el listado de municipios y soluciona el problema de las tildes en el desplegable de los municipios.
- **obtenerPrediccionDiaria.js**: obtiene la predicción diaria real desde la URL que devuelve AEMET.
- **calcularTemperaturaGeneral.js**: calcula una temperatura media del día cuando AEMET ofrece datos por franjas.
- **calcularProbPrecipitacion.js**: selecciona la probabilidad de precipitación del periodo 00–24 o, si no existe, la primera disponible.
- **calcularViento.js**: obtiene la dirección y velocidad del viento del periodo 00–24 o del primer valor disponible.
- **combinarHoras.js**: une todas las variables horarias (temperatura, estado del cielo, viento, precipitación) para que cada hora tenga toda la información junta.

4.2. Búsqueda manual por municipio

La API de AEMET no permite buscar municipios por nombre directamente, así que decido hacerlo así:

i. El usuario escribe un nombre.

ii. El backend descarga el listado completo de municipios.
iii. Busco una coincidencia exacta por nombre en minúsculas.
iv. Si existe, devuelvo el ID.
v. Si no, envío un mensaje de error.

4.3. Búsqueda por provincia y municipio

Aquí me encuentro con un problema: la AEMET **no relaciona municipios con provincias**.
Sin embargo, descubro que el campo id_old empieza con dos dígitos que coinciden con el código de la provincia. Por eso decido:

- Crear un archivo provincias.js en el backend con los códigos oficiales.[1]
- Crear un archivo provincias.json en el frontend para rellenar el desplegable.
- Filtrar los municipios en el backend usando los dos primeros dígitos de id_old.

4.4. Predicción diaria

La API de AEMET divide algunos días en franjas horarias (00–24, 00–06, 06–12…). Mi decisión es usar siempre la franja **00–24**, porque representa el día completo.
Cuando esta franja no existe o está vacía, utilizo **la primera franja disponible**.
Además, uso funciones específicas para procesar cada dato:

- temperatura general
- probabilidad de precipitación
- viento

4.5. Predicción por horas

La AEMET devuelve las horas separadas por variables (temperatura por un lado, viento por otro…). Para poder mostrar una tabla ordenada, necesito unirlas.
Por eso utilizo la función **combinarHoras.js**, que detecta las horas comunes entre todas las variables, ordena las horas y devuelve un objeto con toda la información junta.

4.6. Navegación entre páginas

He decidido crear tres página para hacer uso de React Router DOM:

- /
- /prediccion-dias
- /prediccion-horas

Para navegar utilizo:

- <Link>
- useNavigate()
- useSearchParams() para leer parámetros como el id del municipio o el día seleccionado

[1] Códigos obtenidos de la web:
https://oficinavirtual.comercio.gob.es/AFORIXUpdater/tablaCodificacion.aspx?tabla=provincias

4.7. Separación en componentes

Organizo el frontend en:

- **/pages** → estructura general de cada pantalla
- **/components** → buscadores, tablas, iconos, formateador de fechas, header, footer…

4.8. Iconos meteorológicos

Decido añadir iconos tipo emoji para representar el estado del cielo. Los elijo porque son más visuales, no requieren instalar librerías, funcionan bien en los navegadores y hacen la interfaz más bonita. Los emojis los he obtenido de una web[2].

4.9. Diseño

He decido añadir un **header** y un **footer** porque sin ellos la aplicación se veía demasiado vacía y no me convencía. El header lleva un logo que he hecho yo misma, para darle un toque más personal y profesional, igual que he elegido un nombre para la aplicación. Con el header puedo mostrar el nombre del proyecto de forma más visible.

El footer lo uso para poner mis datos y el año del proyecto creo que ayuda a cerrar visualmente la página y a que todo quede más ordenado.

En cuanto al **CSS**, mi idea es mantener un estilo limpio y estéticamente agradable. No quiero nada muy elaborado para no perder demasiado tiempo en esto, pero lo necesito para que se vea bien. Por eso:

- He usado colores suaves y tonos azules para relacionarlo con el clima.
- He organizado el contenido en contenedores con bordes redondeados.
- He hecho que las tablas sean horizontales y con scroll cuando hace falta (en el caso de la tabla de predicción horaria), ya que ocupa mucho espacio y así queda mejor.
- He añadido algunos detalles como hover en los botones para que la interfaz sea más interactiva.
- He ajustado el diseño con media queries para que no se rompa en pantallas pequeñas.

He hecho el CSS como lo he hecho hasta ahora, que es primero añadiendo clases en el código, luego haciendo un CSS básico y modificando viendo la aplicación para ir viendo cómo queda y poder cambiar cosas poco a poco. Finalmente, he tenido que quitar algunas de las clases que había puesto en el código, añadido otras o cambiarlas.

4.10. Gestión de errores

Intento que los errores sean claros tanto en backend como en frontend:

- Si AEMET falla, muestro un mensaje explicando qué ha pasado.
- Si el usuario no escribe nada, aviso antes de hacer la petición.
- Si un municipio no existe, informo.

[2] https://emojipedia.org/es

5. CONCLUSIONES

Después de hacer el proyecto, siento que he aprendido a trabajar de forma más completa con React, especialmente en lo relacionado con la navegación entre páginas y la organización del frontend en componentes. También he entendido mejor cómo conectar un backend con un frontend y cómo intercambiar datos entre los dos.

Además, he aprendido a manejar mejor los errores, ya que en tareas anteriores la mayoría de veces un error lo presentaba con un alert y ya está.

Creo que me ha servido para aprender un poco más acerca de cómo puede ser un proyecto real, que es algo que no tenía claro.

Por otro lado, me ha costado mucho llevar todo el GitHub al día, no estoy acostumbrada a hacer las cosas así y a veces simplemente iba haciendo el código y se me olvidaba hacer los commits. Es algo que me ha servido para intentar poco a poco tenerlo más en cuenta.

Pero, lo que más dificultades me ha generado ha sido trabajar con la API de AEMET. La estructura de sus datos no me ha resultado fácil de comprender, algunas franjas horarias aparecen vacías y ciertos valores no siguen un patrón uniforme. Esto me ha obligado a crear funciones específicas para limpiar y reorganizar la información y creo que es una de las cosas que más tiempo lleva a la hora de trabajar con una API así. Comprenderla y entender cómo proporciona los datos.