APLICACIÓN METEOROLÓGICA - AEMET REACT

Proyecto: Desarrollo Web en Entorno Cliente (DWEC)
Curso: 2º Desarrollo de Aplicaciones Web (DAW) - 2025/2026
Alumno: David Aznar Moreno

ÍNDICE

---

1. DESCRIPCIÓN DEL PROYECTO

1.1. Objetivo

Desarrollar una aplicación web completa que permita consultar la predicción meteorológica, yo lo enfoque en la predicción de municipios de Andalucía utilizando la API oficial de AEMET (Agencia Estatal de Meteorología).

1.2. Funcionalidades Principales

La aplicación permite al usuario:

- **Seleccionar provincia y municipio** de Andalucía mediante selectores desplegables
- **Consultar dos tipos de predicción:**
  - Predicción diaria: los pronosticos para el dia de hoy y los próximos 6 días (AEMET devuelve hasta 7 días)
  - Predicción horaria: pronósticos hora por hora para las próximas 12 horas
- **La informacion metereológica de la predicción que se muestra es:**
  - Estado del cielo (despejado, nublado, lluvia, tormenta, nieve, etc.)
  - Temperaturas máximas y mínimas
  - Probabilidad de lluvia
  - Velocidad y dirección del viento
  - Humedad
- **Cambiar entre modo claro y modo oscuro**
- **Interfaz responsive** que se adapta a pantallas móviles, tablets y ordenadores

1.3. Arquitectura Cliente-Servidor

El proyecto sigue una arquitectura desacoplada con dos aplicaciones independientes:

Frontend (React):

- Puerto 5173
- Interfaz de usuario interactiva

- Gestión de estado con hooks de React
- Comunicación con el backend mediante fetch API

Backend (Node.js + Express):

- Puerto 3000
- Actúa como proxy entre el frontend y la API de AEMET
- Oculta la clave de API (seguridad)
- Maneja problemas de encoding de la API oficial
- Gestiona errores y timeouts

Flujo de comunicación:

```
Usuario - React (frontend) - Express (backend) -API AEMET - Express - React =
Usuario
```

el frontend NO se comunica directamente con AEMET.

1.4. Datos Mostrados

La app presenta la información de manera clara , comprensible de manera visual y atracttiva:

- **Vista diaria:** Tarjetas individuales por cada día con iconoes e informacion sobre la temperatura, estado del cielo, probabilidad de lluvia y viento
- **Vista horaria:** Tabla con iconos e informacion sobre temperaturas hora por hora, las condiciones del cielo, la probabilidad de lluvia, la humedad, la cantiddad de lluvia si la hay y el viento
- **Estados visuales:** Indicadores de carga de la informacion, mensajes de error describiendo el tipo de error y aviso cuando no hay ningun resultado.

---

2. INSTRUCCIONES DE INSTALACIÓN Y EJECUCIÓN

2.1. Requisitos Previos

Antes de instalar el proyecto, es necesario tener:

- **Node.js** version 14 (https://nodejs.org/)
- **npm** se instala automáticamente con Node.js
- **Editor de código** yo he utilizado Visual Studio Code
- **Clave de API de AEMET** se obtiene mediante el correo electronico en la web de AEMET (https://opendata.aemet.es/centrodedescargas/inicio)

2.2. Obtener la Clave de AEMET

1. Hay que acceder a https://opendata.aemet.es/centrodedescargas/inicio
2. Registrarse con un email
3. AEMET te envia la clave API al correo
4. Hay que guardar la clave para luego configurar el backend

2.3. Instalación del Backend

Abriendo una terminal y ejecutando los siguientes comandos:

```
# Se navega a la carpeta del backend
cd backend

# se instalan las dependencias
npm install

# Se crea el archivo de variables de entorno
puede ser desde la terminal o desde visual creando un archivo nuevo y guardandolo
como .env
```

Abrir el archivo `.env` con un editor de texto(yo use visual studio code) y añadir la clave de AEMET:(El archivo `.env` NO debe subirse a repositorios públicos por seguridad.)

```
AEMET_API_KEY=tu_clave_aqui
PORT=3000
```

2.4. Instalación del Frontend

En otra terminal diferente :

```
# Se navega a la carpeta del frontend
cd frontend

# Se instalan las dependencias con npm
npm install
```

2.5. Ejecución de la Aplicación

Es necesario tener ** las 2 terminales abiertas a la vez**:

Terminal 1 - Backend:

```
cd backend
node server.js
```

Debe aparecer:

```
 Servidor corriendo en http://localhost:3000
```

Terminal 2 - Frontend:

```
cd frontend
npm run dev
```

Debe aparecer :

```
Local: http://localhost:5173/
```

2.6. Acceder a la Aplicación

Abrir el navegador y acceder a: **http://localhost:5173**

La aplicación se podra ver desde el navegador y se podran usar todas sus funcionalidades

3. DECISIONES TÉCNICAS

3.1. Elección de React como Framework Frontend

Decisión: Utilizar React con componentes funcionales y hooks.

Justificación:

React te permite:

- **Dividir la interfaz en componentes reutilizables:** Cada parte de la aplicación (cabecera, formulario, resultados) es un componente independiente que puedo modificar sin que afecten al resto.
- **Gestionar el estado de forma eficiente:** Con `useState` controlo qué provincia está seleccionada, si hay errores, si estamos cargando datos, etc. Cuando cambio un estado, React actualiza automáticamente la interfaz.
- **Renderizado automático:** No se necesita manipular el DOM manualmente con `document.getElementById()`. React se encarga de actualizar lo que ve el usuario cuando cambian los datos.

3.2. Arquitectura Frontend-Backend

Decisión: Crear un backend con Node.js + Express que actúe como intermediario entre React y la API de AEMET.

Seguridad:

- La clave de API de AEMET nunca sale del servidor
- Si pusiera la clave en React, cualquiera podría verla abriendo las herramientas de desarrollador del navegador
- El archivo `.env` con la clave nunca se sube a Git

Flexibilidad:

- Si AEMET cambia algo en su API, solo hay que modificar el backend
- El frontend no se altera ni cambia
- Puedo añadir mas elementos o funcionalidades

Manejo de problemas:

- AEMET tiene problemas de caracteres españoles mal codificados
- El backend los soluciona antes de enviar los datos al frontend
- La API de AEMET funciona en dos pasos la primera petición devuelve una UR y la segunda petición obtiene los datos reales.

3.3. Separación en Componentes Independientes

Decisión: hedividido la App en 3 componentes principales: Cabecera, FormularioBusqueda y ResultadoTiempo.

Cabecera.jsx:

- Solo se encarga del logo y el botón de modo oscuro
- Si quiero cambiar el diseño del header o anadir mas contenido solo modifico este archivo
- Se puede reutilizar en otros proyectos o mas paginas

FormularioBusqueda.jsx:

- Maneja solo la lógica del formulario
- Controla qué municipios mostrar según la provincia seleccionada
- se comunica con App.jsx a través de propiedades (props)

ResultadoTiempo.jsx :

- Es el más dificil y complejo porque es el que procesa los datos de AEMET
- Contiene funciones helper para formatear las fechas, encontrar los máximos, mapear los datos
- Muestra las dos vistas diferentes, la predición diaria y por horas horaria

Ventajas de esta distribucion:

- Es más fácil de entender, cada componente tiene una funcion clara
- Es mas fácil de mantener, si hay un fallo en el formulario, sé e dónde hay que buscarlo
- Testeable, se puede probar cada componente por separado

3.4. Comunicación entre Componentes con Props

Los componentes padres pasan datos a los hijos mediante las propiedadades props, y los hijos notifican eventos al padre mediante las funciones callback.

En React, los datos vanm de arriba hacia abajo . El componente padre (`App.jsx`) controla el estado global y los componentes hijos solo muestran la info o notifican eventos.

¿Por qué no se puede o acceder directamente a las variables del padre?

Esta es una pregunta que me hice al principio. En JavaScript puedo acceder a cualquier variable desde cualquier parte. Por qué en React no? es por la encapsulacionn y la reutilización. Si cabecera.jsx pudiera

acceder a las variables del pade: App.jsx dependeria uno del otro con l oque no podria reutilizar cabecera en otro proyecto , si cambio algo en App.jsx podria romper algo en cabecera y seria dificil saber que componente esta usando que datos.

Como se que props va a necesitar un componente nuevo? Cuando creo un componente hijo hay que preguntrarse ¿Qué info necesita mostrar? esas son props de datos ¿Qué puede hacer el usuario? esas son props de funciones ¿Qué estado global necesita saber? mass props de datos

Ejemplo:

```
// App.jsx (el padre) pasa los  datos a Cabecera (el hijo)
<Cabecera
  modoOscuro={modoOscuro}           // Valor actual del tema
  toggleModo={cambiarModoOscuro}    // Función para cambiarlo
/>

// Cabecera.jsx ( el hijo) recibe las props
export default function Cabecera({ modoOscuro, toggleModo }) {
  // Usa modoOscuro para mostrar el icono correcto segun ewl modo
  // Llama a toggleModo() cuando el usuario hace clic en el icono
}
```

Los componentes son independientes y se pueden reutilizar. Cabecera no necesita saber cómo funciona App, solo tiene que usar lo que se le pasa.

Como se que props va a necesitar un componente nuevo? Cuando creo un componente hijo hay que preguntrarse ¿Qué info necesita mostrar? esas son props de datos ¿Qué puede hacer el usuario? esas son props de funciones ¿Qué estado global necesita saber? mass props de datos

Por ejemplo en: `ResultadoTiempo.jsx`:

```
export default function ResultadoTiempo({ resultado, tipo }) {
  // resultado: los datos meteorológicos que debo mostrar
  // tipo: si es predicción diaria u horaria (afecta cómo muestro los datos)
}
```

solo necesuito 2 props porque lo unico que hace es mostrar datos n otiene que modificar nada solo presentar la info de forma bonita, al contrario que `FormularioBusqueda.jsx` que es el ejemplo que mas props utiliza y el mas coimplejo ya que recibe 9 props diferentes de `App.jsx`:

```
<FormularioBusqueda
  provinciaCodigo={provinciaCodigo}
  setProvinciaCodigo={setProvinciaCodigo}
  municipioCodigo={municipioCodigo}
  setMunicipioCodigo={setMunicipioCodigo}
  tipo={tipo}
  setTipo={setTipo}
```

```
    onSearch={buscarTiempo}
    cargando={cargando}
    cambiarTipo={cambiarTipoBusqueda}
  />
```

Porque recibe tantas props? **provinciaCodigo** y **municipioCodigo**:le digo qué provincia y municipio ha seleccionado el usuario. El hijo necesita saber esto para mostrar los valores correctos en los selectores.

setProvinciaCodigo y **setMunicipioCodigo**: le paso las funciones para que el hijo pueda actualizar estos valores cuando el usuario seleccione otra opción.

tipo: le digo si el usuario quiere predicción diaria u por horas.

setTipo: funcion para cambiar el tipo de predicción.

onSearch: La funcion que busca el tiempo. Cuando el usuario pulsa "Buscar", el hijo ejecuta esta función del padre.

cargando:le digo si estamos esperando datos para que deshabilite los botones y evite que haya busquedas duplicads

cambiarTipo: una funcion especial que cambia el tipo y busca automáticamente

En FormularioBusqueda.jsx las recibo así:

```
export default function FormularioBusqueda({
  provinciaCodigo,
  setProvinciaCodigo,
  municipioCodigo,
  setMunicipioCodigo,
  tipo,
  setTipo,
  onSearch,
  cargando,
  cambiarTipo
}) {
  // Ahora puedo usar todas directamente sin props.
}
```

Al principio intente poner toda la lógica en un solo componente gigante para evitar complicaciones con las propspero fue un error. El codigo se volvió imposible de leer y mantener y me daba fallos constantemente.

Cuando aprendí como tenia que dividir los componentes y usar las props correctamente, todo fue mas facil porque podia ver cada componente por separado y reutilizarlos , econtrar los fallos mas facil y asi trabajr por separado sin preocuparme de romperlo todo

3.5. Servicio Separado para Peticiones HTTP al backend

Decidi crear un archivo `servicioTiempo.js` que tenga todas las peticiones al backend.

-Si la URL del backend cambia, solo hay que modificar este archivo -Si quiero añadir algo lo hago aquí y no tengo que cambiar nada en otros componentes

Manejo de errores en un solo lugar: -Todos los errores de kla red se manejan en un solo lugar -Implementé timeouts de 10 segundos para evitar esperas infinitas -Clasifico los errores segun el tipo que sean (sin conexión, servidor caído, timeout)

3.6. Datos Estáticos (provincias y municipios) en vez de pedirlos al servidor de AEMET

Guarde las provincias y municipios de Andalucía en un archivo JavaScript (`andalucia.js`) en vez de pedirlos continuamente al servidor.

Estos datos no cambian. Por eso es mejor tenerlos guardados localmente porque: -No hay que esperar a una petición HTTP y cargan mas rapido -La app puede cargar aunque no haya internet -Ahorro peticiones innecesarias al servidor para no saturarlo -No necesito crear un endpoint adicional en el backend para mas peticiones

He incluido aproximadamente 10-13 municipios por provincia los que estan más poblados, para no hacer un archivo muy grande. Si en unl futuro se necesitan todos los municipios de Andalucía que son más de 700, se podrían pedir a la API de AEMET, o anadirlos manualemnte

3.7. Gestión de Estados en React

Use 8 estados diferentes en `App.jsx` para controlar los aspectos de la app.

Estados utilizados:

```
const [provinciaCodigo, setProvinciaCodigo] = useState('');  // provincia
seleccionada
const [municipioCodigo, setMunicipioCodigo] = useState('');  // municipio
seleccionado
const [tipo, setTipo] = useState('diaria');                  // El tipo de
predicción
const [cargando, setCargando] = useState(false);             // ¿esta cargando?
const [error, setError] = useState(null);                    // ¿hay algun  error?
const [sinResultados, setSinResultados] = useState(false);   // ¿no hay datos?
const [resultado, setResultado] = useState(null);            // datos
meteorológicos
const [modoOscuro, setModoOscuro] = useState(false);         // tema oscuro/claro
```

Cada estado controla un aspecto de la interfaz de la app: -si `cargando === true` =muestro el icono de carga -Si `error !== null` = muestro el mensaje de error -Si `resultado !== null` = muestro los datos

Podría haber usado un solo estado gigante, pero cambiar un solo valor sería más dificl y lioso

3.8. Validaciones en Frontend y Backend

Implemente validaciones tanto en el frontend com oen el backend para asegurarme que se introducen los datos correctos

Frontend: Mejor experiencia de usuario Botón "Buscar" se deshabilita si falta la provincia o el municipio por seleccionar Validación antes de enviar peticiones al backend

Backend: Validación de que el código de municipio no este vacío Verificación de que existe la API key

Ejemplo en frontend:

```
if (!provinciaCodigo || !municipioCodigo) {
  setError('Por favor, selecciona provincia y municipio');
  return;
}
```

Ejemplo en backend:

```
if (!codigo || codigo.trim() === '') {
  return res.status(400).json({ error: 'Código vacío' });
}
```

3.9. primer error principal que me encontre: el encoding de AEMET

Los datos de la AEMET venían con caracteres españoles mal codificados: "Meteorología" aparecía como "Meteorolog�a" "Cádiz" aparecía como "C�diz"

la API de AEMET envía los datos con encoding Latin1 (ISO-8859-1) pero el header dice UTF-8, esto hacia que los caracteres con tildes y ñ se corrompieran y se vieran mal.

Probe usando `.text()` directamente y no funcionó, cambiand headers de la petición pero AEMET no lo respetaba, y usar librerías de conversión me parecia demasiado complejo.

Solucion que tuve que buscar en internet

```
// Leer como bytes crudos
const buffer = await respuesta2.arrayBuffer();
// Intentar UTF-8 primero
let texto = new TextDecoder('utf-8').decode(buffer);
// Si veo el caracter de reemplazo �, probar Latin1
if (texto.includes('�')) {
  texto = new TextDecoder('latin1').decode(buffer);
}
// y ya parsear el json
const datos = JSON.parse(texto);
```

Este problema me llevó varias horas descubrirlo, pero aprendí mucho sobre como funcionan los encodings de caracteres.

3.9.1 segundo error principal que me encontre: API de AEMET funciona en 2 pasos

La primera petición a AEMET no devuelve los datos, sino un JSON con una URL. La documentación de AEMET no lo explica claramente o yo no fui capaz de entenderlo.

```
// Primera petición:
const respuesta1 = await fetch('https://opendata.aemet.es/...');
const info = await respuesta1.json();

console.log(info);
// Resultado: {
//   datos: "https://opendata.aemet.es/opendata/sh/12345abc",
//   metadatos: "..."
// }

// ¡Los datos reales están en info.datos!
```

Solución: Creé la función `pedirAemetDosPasos()` que maneja esto automáticamente.

Problema 3: CORS bloqueaba las peticiones

```
Access to fetch at 'http://localhost:3000' from origin 'http://localhost:5173'
has been blocked by CORS policy
```

El navegador bloquea por seguridad las peticiones entre diferentes orígenes (puertos diferentes).

Solución:

```
// En server.js:
const cors = require('cors');
app.use(cors());  // Permitir peticiones desde cualquier origen
```

Para producción, debería especificar solo mi frontend:

```
app.use(cors({
  origin: 'https://mi-dominio.com'
}));
```

Problema 4: Estado desincronizado al cambiar búsquedas rápido

Si el usuario cambiaba de municipio rápidamente mientras cargaba, a veces se mostraban datos del municipio anterior. Las peticiones asíncronas no tienen orden garantizado. Si hago: -Buscar Sevilla (tarda 2 segundos) -Buscar Cádiz (tarda 1 segundo)

Cádiz termina primero, luego Sevilla sobrescribe los resultados.

Solución:

```
const buscarTiempo = async () => {
  setResultado(null);  // Limpiar resultado anterior ANTES de buscar
  setCargando(true);

  const datos = await obtenerPrediccion(...);

  // Solo actualizar si no hay nueva búsqueda en marcha
  if (datos) {
    setResultado(datos);
  }

  setCargando(false);
};
```

También podría usar `useEffect` cleanup para cancelar las peticiones anteriores, pero esta solución más simple funcionó

Problema 5: Municipios desaparecían al cambiar provincia

Al seleccionar una nueva provincia, el municipio seleccionado anteriormente quedaba en el estado pero ya no existía en la lista filtrada.

Selecciono Sevilla - Municipio: Sevilla capital Cambio a Cádiz - Todavía dice "Sevilla capital" pero no existe en Cádiz

Solución:

```
// En App.jsx:
const alCambiarProvincia = (nuevaProvincia) => {
  setProvinciaCodigo(nuevaProvincia);
  setMunicipioCodigo('');  // ¡RESETEAR municipio al cambiar provincia!
};
```

Problema 6: Iconos meteorológicos no cargaban

Las rutas de los iconos daban 404.

Estaba usando rutas absolutas incorrectas:

```
import icono from '/src/assets/iconos/despejado.svg';
```

Solución: Usar rutas relativas:

```
import icono from '../assets/iconos/despejado.svg';
```

Y para importación dinámica:

```
const iconos = import.meta.glob('../assets/iconos/*.svg', { eager: true });
```

3.10. Timeouts y Manejo de Errores

Decisión: Implementar timeouts de 10 segundos en las peticiones HTTP usando `AbortController`.

AEMET normalmente responde en 1-2 segundos, si tarda mas de 10 segundos, es porque puede haber un problema y el usuario debe saber que pasa para no desesperarse o abandonar la pagina

Como lo implemente:

```
const controlador = new AbortController();
const timeout = setTimeout(() => controlador.abort(), 10000);

fetch(url, { signal: controlador.signal })
  .catch(error => {
    if (error.name === 'AbortError') {
      return { error: 'La petición tardó más de 10 segundos' };
    }
  });
```

Tambien he identificado y manejado 3 tipos de errores que pueden suceder y que sucedian constantemente de hecho: 1.Sin conexión a internet: `TypeError` de fetch 2.Que elBackend se haya caído: Status 500 del servidor 3.Timeout: `AbortError` cuando se cancela la petición

---

4. CONCLUSIONES

4.1. Aprendizaje Técnicos

React y su ecosistema:

Al comenzar el proyecto, React me parecía complejo. Los hooks (`useState`, `useEffect`) tienen reglas que al principio no entendía. Sin embargo, tras hacer y pelearme con el proyecto cada vez me gusta mas react

Trabajo con APIs reales:

La API de AEMET me enseñó que las APIs reales no son perfectas: -La documentación no siempre es clara(sobretodo las de paginas institucionales o por lo menos las del gobierno) -Hay problemas (encoding, dos pasos, etc.) -Es necesario manejar timeouts, las APIs pueden fallar todo el rato

Arquitectura frontend-backend:

Entiendo ahora por qué hay separar responsabilidades: -El backend oculta por ejemplo API keys y la complejidad -El frontend se enfoca en la experiencia del usuario

4.2. Problemas superados

El problema del encoding:

Fue la dificultad más grande. Me llevó horas entender por qué "Cádiz" se veía como "C�diz". Aprendí: -Diferentes sistemas de codificación de caracteres (UTF-8 vs Latin1) -Cómo leer respuestas HTTP como bytes crudos -La clase `TextDecoder` y sus opciones

El sistema de dos pasos de AEMET:

No está bien documentado. Al principio pensaba que mi código estaba mal. Cuando descubrí que era el comportamiento esperado de la API, y creé la función `pedirAemetDosPasos()` para reutilizar esa lógica comprendi como funcionaba rtealemnte y el problema.

Estado desincronizado:

Si el usuario cambiaba de municipio rápidamente mientras carga el resultado, a veces se mostraban datos del municipio que habia buscado anteriormente. Aprendí a limpiar el estado anterior antes de hacer una nueva búsqueda.

4.3. Decisiones de Diseño

Diseno de logotipo y favicon personalizado: He disenado con illustrator un logotipo con sus respectivos colroesd corporativos acorde a la imagen de marca que tenia la web. El logotipo tien un diseno simple y minimalista, con un fondo blanco y un icono de un sol y nubes sobre una A capital, representando la A de Andalucia.

Modo oscuro con colores corporativos de forma prederteminada: He puesto por defecto la interfaz de la app en modo oscuro para que lavisualizacion sea mas agradable a la vista, y porque para mi gusto era el que mas me gusta y siempre perfiero el modo oscuro en las apps o webs.

Modo claro con colores corporativos: He diseñado la paleta de colores basándome en el logo de la aplicación (tonos azules). He creado 16 variables CSS que controlan todos los colores, permitiendo cambiar el tema con un simple atributo `data-theme="dark"`.

Diseno por tarjetas

el diseno por tarjetas me parecio mas agradable y mas visual, y asi el usuarios puede diferenciar facilmente los datos de cada municipio. o de cada tip ode prediccion

Iconos meteorológicos: He usado **Meteocons** iconos para el clima en fornmato SVG animados de Bas Milius un programador web que encontre en internet: De los 123 iconos animados de la librería Meteocons que descargue de internet, y los mapeé a los 34 códigos que usa AEMET. para que la app se viera mas visual y atractiva

Fuente elegida: la del sistema nativo del dispositivo

```
font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, sans-serif;
```

¿Por qué elegi esta en vez de una personalizada? porque carga instantánea no hay que descargar la fuente Familiaridad la gente ya conoce esta fuente Esta optimizada para cada sistema operativo

Diseño responsive:

He probado la aplicación en: Móvil (375px) Tablet (768px) Desktop (1920px)

La interfaz se adapta correctamente a todos los tamaños usando CSS Grid, Flexbox y media queries.

El layout de la app de 4 secciones se ve igual en pantllas grandes como en moviles, ya que es mas comodo hacer scroll vertical que horizontal, el usuarios puede ver los resultados de la busqueda sin tener que desplazarse horizontalmente

Ver la aplicación funcionando con datos reales:

Cuando busqué "Sevilla" y aparecieron los datos reales del tiempo, con los iconos correctos y toda la información bien puesta, fue muy reconfortante. Ver que algo que había construido desde cero y me dio tantos quebraderos de cabeza funcionaba y se veia como esperaba.

Después de horas de frustración viendo caracteres extraños, cuando finalmente "Córdoba" se mostraba correctamente en vez de "C�rdoba" o cuando se recibia correctamente toda la info metereologica y que no me salia el dichoso "-" que yo mismo habia configurado para cuando n ose recibian datos o no habian, fue muy satisfactorio que funcionara todo bien.

Logicamnte la app se encuentra en un estado inicial en el que todavia se podrian mejorar muchas cosas y poner mas funciones , que detallo en la siguiente seccion

4.5. Lo Que Mejoraría con Más Tiempo

1. Caché de respuestas: Guardar respuestas en la memoria durante 1 hora para evitar saturar la API de AEMET con peticiones repetidas.

2. Más provincias: Actualmente solo Andalucía. Extender a toda España sería fácil: solo añadir datos al array.

3. Gráficas de temperatura: Usar Chart.js para mostrar la evolución de la temperatura en un gráfico de líneas.

4. Persistencia del modo oscuro: Usar `localStorage` para recordar la preferencia del usuario entre las sesiones.

5. Tests automatizados: Usar Jest + React Testing Library para probar que los componentes funcionan correctamente.

6. Accesibilidad mejorada: Añadir `aria-label`, navegación por teclado completa y soporte para lectores de pantalla, eso lo he visto en la asignatura de diseno de interfaces.

4.8. Reflexión Final

Este proyecto me ha enseñado que desarrollar una aplicación web moderna no es solo escribir código que "funcione". Es:

- **Entender al usuario:** ¿Qué necesita? ¿Cómo hacer la interfaz intuitiva?

- **Manejar errores elegantemente:** El usuario no debe ver errores técnicos crudos
- **Escribir código mantenible:** Otros (o yo en el futuro) deben poder entenderlo
- **Tomar decisiones justificadas:** Cada elección técnica tiene una razón
- **Resolver problemas reales:** Encoding, APIs complejas, timeouts, etc.
- **Crear algo visualmente agradable:** El diseño importa tanto como la funcionalidad

ANEXOS

A. IMPORTANTE!!! VER CARPETA ADJUNTA (CAPTURAS DE PANTALLA APP) PARA VER VISUALMENTE LA INTERFAZ DE LA APP Y SU FUNCIONAMIENTO

B. Repositorio Git

Contenido del repositorio:

- Código completo del frontend
- Código completo del backend
- README.md con documentación técnica
- .gitignore configurado correctamente (excluye node_modules y .env)

C. Recursos Utilizados

APIs:

- AEMET OpenDat(https://opendata.aemet.es/) - Datos meteorológicos oficiales

Librerías de iconos:

- Meteoconshttps://github.com/basmilius/weather-icons por Bas Milius - Iconos meteorológicos

Documentación consultada:

- React Documentation (https://react.dev/)
- Express Documentation (https://expressjs.com/)
- MDN Web Docs (https://developer.mozilla.org/)
- Plantillas Markdown: https://github.com/fhernanb/Plantillas
- como personalizar y escribir Marksdowns: https://help.onparallel.com/es/articles/6272953-como-anadir-formato-a-mis-plantillas-markdown
- CURSO REACT.JS - Aprende desde cero - midulive - https://www.youtube.com/watch?v=7iobxzd_2wY
- Haz tu primera App con Node.js + React - Mudir Solutions - https://www.youtube.com/watch?v=hnCDdOgapWU
- Cómo CONSUMIR una API REST con JAVASCRIPT y Fetch + Promises con gestión de Errores - midulive - https://www.youtube.com/watch?v=FJ-w0tf3d_w
- ¿Cómo obtener información gratuita de Aemet de forma puntual - AEMET - https://www.youtube.com/watch?v=wGNYqLOq4fE

- Manejo de errores en React - Linder Tech - https://www.youtube.com/watch?v=65gC6yQXEAc

Herramientas:

- Visual Studio Code - Editor de código
- Git - Control de versiones
- Chrome DevTools - Depuración y testing responsive
- Adobe Illustrator - Diseño del logo
- Extension para convertir Markdowns a PDF en Visual Studio Code - https://marketplace.visualstudio.com/items?itemName=yzhang.markdown-pdf-enhanced

---