**Índice**

[1. Antecedentes 3](#_Toc197541709)

[1.1 Introducción 3](#_Toc197541710)

[1.2 ¿Qué es una PWA? 3](#_Toc197541711)

[1.3 Service Workers y almacenamiento en caché 3](#_Toc197541712)

[1.4 Manifest y capacidades de instalación 3](#_Toc197541713)

[1.5 Comparativa: PWA vs Web tradicional 3](#_Toc197541714)

[2. Planteamiento del Problema 4](#_Toc197541715)

[3. Propuesta de Solución 4](#_Toc197541716)

[4. Materiales y Métodos 4](#_Toc197541717)

[4.1 Materiales 5](#_Toc197541718)

[4.2 Métodos 5](#_Toc197541719)

[5. Desarrollo de la Solución 5](#_Toc197541720)

[5.1 Estructura general de la PWA 5](#_Toc197541721)

[5.2 Backend en Node.js 6](#_Toc197541722)

[5.3 Frontend en React 6](#_Toc197541723)

[5.4 Implementación del Service Worker 6](#_Toc197541724)

[5.5 Funcionalidad offline y offline.html 6](#_Toc197541725)

[6. Resultados 7](#_Toc197541726)

[6.1 Modo online 7](#_Toc197541727)

[6.2 Modo offline 7](#_Toc197541728)

[6.3 Almacenamiento en caché 7](#_Toc197541729)

[6.4 Instalación como aplicación 7](#_Toc197541730)

[7. Conclusión 7](#_Toc197541731)

# 1. Antecedentes

## 1.1 Introducción

En esta práctica se desarrolló una Aplicación Web Progresiva (PWA) orientada a la consulta del clima en distintas ciudades mediante el consumo de una API externa (OpenWeatherMap). El objetivo principal fue implementar una aplicación accesible tanto en modo online como offline, haciendo uso de tecnologías modernas como React para el frontend, Node.js con Express para el backend, y herramientas clave del ecosistema PWA como Service Workers, la Cache API y un archivo de manifiesto (manifest.json).

Esta arquitectura busca ofrecer una experiencia de usuario fluida, confiable y rápida incluso en condiciones de conectividad limitada o intermitente. Las PWAs combinan lo mejor de las páginas web tradicionales con características avanzadas de las aplicaciones móviles, lo que las convierte en una alternativa ideal para entornos donde la instalación nativa no es viable o deseada.

El desarrollo se centró en aplicar los principios fundamentales de las PWAs: independencia de red, instalación como app, carga rápida y comportamiento similar al de una aplicación nativa, sin comprometer la escalabilidad ni la modularidad de la solución.

## 1.2 ¿Qué es una PWA?

Una Progressive Web App (PWA) es una aplicación construida con tecnologías web estándar como HTML, CSS y JavaScript, pero que incorpora capacidades avanzadas como funcionamiento offline, acceso al hardware del dispositivo (en ciertos casos), y posibilidad de instalación como si se tratara de una app móvil o de escritorio.

Su comportamiento está diseñado para ser progresivo, es decir, mejora de forma automática a medida que el navegador o el dispositivo ofrecen mayores funcionalidades. Esto permite una compatibilidad amplia, una experiencia consistente, y un desempeño óptimo, tanto en redes de alta velocidad como en escenarios con conectividad limitada.

## 1.3 Service Workers y almacenamiento en caché

El Service Worker es una de las piezas clave en una PWA. Se trata de un script en segundo plano que intercepta todas las solicitudes de red realizadas por la aplicación. Su principal función es permitir que la aplicación funcione correctamente sin conexión, mediante el almacenamiento en caché de archivos estáticos y dinámicos.

Durante su ciclo de vida (instalación, activación y ejecución), el Service Worker puede:

* Precargar recursos estáticos (index.html, bundle.js, estilos e imágenes).
* Cachear dinámicamente respuestas de API para futuras consultas offline.
* Proveer una respuesta personalizada (offline.html) cuando no hay red ni datos cacheados.

El uso adecuado de la Cache API, junto con una estrategia coherente de versionado y limpieza de cachés obsoletos, garantiza un comportamiento robusto y eficiente para el usuario final.

## 1.4 Manifest y capacidades de instalación

El archivo manifest.json es el componente responsable de permitir que la PWA pueda instalarse en el dispositivo del usuario y se integre con el sistema operativo como una app nativa.

Este archivo incluye información clave como:

* name y short\_name: nombre visible de la aplicación.
* icons: íconos en diferentes resoluciones para múltiples dispositivos.
* start\_url: punto de entrada tras la instalación.
* display: determina si la app se abre en modo standalone, fullscreen, etc.
* theme\_color y background\_color: colores usados en la pantalla de carga o al instalar.

Gracias al manifiesto, los navegadores pueden presentar la opción de "Agregar a la pantalla de inicio", mejorando significativamente la accesibilidad y presencia de la aplicación.

## 1.5 Comparativa: PWA vs Web tradicional

|  |  |  |
| --- | --- | --- |
| Característica | Web tradicional | PWA |
| Funcionamiento offline | No | Sí (Service Worker + Cache API) |
| Instalación | No | Sí, mediante manifest.json |
| Control de caché | Limitado (navegador) | Completo (precaché y dinámico) |
| Carga inicial | Lenta en red inestable | Rápida (precargada desde caché) |
| Experiencia de usuario | Dependiente del navegador | Equivalente a app móvil o desktop |

# 2. Planteamiento del Problema

En aplicaciones web tradicionales, la dependencia constante de una conexión activa a Internet puede afectar significativamente la experiencia del usuario. Esto se vuelve especialmente crítico en contextos donde la conectividad es inestable, intermitente o costosa.

El desafío planteado en esta práctica fue:  
¿Cómo construir una aplicación web que pueda continuar funcionando de forma útil y amigable aun cuando el usuario pierda la conexión a Internet?

Se buscaba construir una solución que:

* Permita al usuario consultar datos del clima de manera online y offline.
* Instale la aplicación como si fuera nativa en el dispositivo.
* Provea retroalimentación amigable (como mostrar un mensaje personalizado) si el recurso no está disponible sin conexión.

Al mismo tiempo, debía mantenerse la modularidad del sistema mediante una separación clara entre frontend (React), backend (Node.js + Express) y lógica de red (Service Worker). Esta división facilita la escalabilidad futura, pruebas aisladas y mantenimiento por capas.

# 3. Propuesta de Solución

La solución consistió en implementar una Aplicación Web Progresiva (PWA) dividida en dos grandes componentes:

1. Frontend (React): Interfaz interactiva donde el usuario puede ingresar el nombre de una ciudad para consultar el clima. También maneja los estados de conexión y los mensajes de error.
2. Backend (Node.js + Express): Encargado de recibir peticiones del frontend, consultar la API externa de OpenWeatherMap y devolver los datos procesados.

Se incorporó un Service Worker que intercepta solicitudes:

* Cachea recursos estáticos al instalar la app.
* Cachea dinámicamente las respuestas de la API de clima tras una consulta exitosa.
* Devuelve datos desde el caché si no hay red disponible.
* Redirige a offline.html cuando el recurso no está disponible y no hay datos guardados.

La aplicación fue configurada para que pudiera ser instalada en dispositivos compatibles mediante un archivo manifest.json, incluyendo íconos e información relevante para una experiencia nativa.

Con esta solución, se logró una PWA robusta, eficiente y accesible que combina lo mejor del desarrollo web moderno con prácticas de alta disponibilidad y resiliencia.

# 4. Materiales y Métodos

## 4.1 Materiales

Para el desarrollo e implementación de esta práctica se utilizaron las siguientes tecnologías y herramientas:

* Frontend:  
   • React: para construir la interfaz de usuario y manejar el ciclo de vida de los componentes.  
   • Fetch API: para realizar peticiones HTTP desde el cliente al backend.
* Backend:  
   • Node.js con Express: para construir el servidor que actúa como intermediario entre la aplicación React y la API de OpenWeatherMap.  
   • Axios: para realizar peticiones HTTP desde el servidor hacia la API externa.  
   • dotenv: para almacenar la clave de API de forma segura mediante variables de entorno.
* PWA (Service Worker):  
   • JavaScript y Cache API: para interceptar solicitudes, gestionar el almacenamiento en caché y proporcionar respuestas en modo offline.
* Herramientas de desarrollo y prueba:  
   • Visual Studio Code: entorno de desarrollo integrado.  
   • Google Chrome + DevTools: para pruebas, depuración y monitoreo del almacenamiento en caché.  
   • Postman: para validar la comunicación entre cliente y servidor durante el desarrollo.  
   • Lighthouse (Chrome DevTools): para evaluar la calidad de la PWA (auditoría de instalación, performance, accesibilidad y funcionalidad offline).
* Sistema operativo:  
   • Windows 10 en entorno de pruebas local.

## 4.2 Métodos

El proceso metodológico seguido se dividió en varias fases iterativas, de menor a mayor complejidad:

1. Construcción de la aplicación React: se creó una estructura básica con create-react-app, enfocándose en la interfaz y lógica para capturar ciudades y mostrar datos del clima.
2. Configuración del backend en Node.js: se implementó un servidor Express con una ruta GET que recibe una ciudad como parámetro, consulta la API de OpenWeatherMap y devuelve los datos al frontend.
3. Implementación del Service Worker:  
    • Precarga de recursos esenciales (index.html, bundle.js, offline.html).  
    • Intercepción de solicitudes a la API para almacenamiento dinámico.  
    • Gestión del almacenamiento en caché y versiones.
4. Diseño del archivo manifest.json: con campos como name, icons, start\_url, display, y theme\_color para permitir instalación y personalización visual.
5. Pruebas funcionales:  
    • Verificación de funcionamiento offline con datos en caché.  
    • Visualización del archivo offline.html como fallback.  
    • Auditoría con Lighthouse para validar la instalación como PWA y el correcto comportamiento sin red.

# 5. Desarrollo de la Solución

## 5.1 Estructura general de la PWA

La solución se dividió en tres componentes principales:

* **Frontend**: Aplicación en React ubicada en el directorio /client.
* **Backend**: Servidor Node.js ubicado en /server, que expone un endpoint /clima.
* **PWA assets**: Archivos como manifest.json, service-worker.js, íconos e offline.html se ubicaron en la carpeta pública (/public), accesible por el navegador.

## 5.2 Backend en Node.js

El backend se implementó en Express con las siguientes características:

* Escucha en el puerto 3001.
* Utiliza **CORS** para permitir la comunicación desde el cliente React en otro puerto.
* La ruta /clima?ciudad={nombre} utiliza **Axios** para consultar OpenWeatherMap y devuelve un JSON con los datos relevantes.

## 5.3 Frontend en React

El componente principal App.js maneja los siguientes elementos:

* Un formulario controlado para capturar el nombre de la ciudad.
* Función obtenerClima() que realiza una solicitud al backend.
* Manejo de errores de conexión y datos no encontrados.
* Uso de useState para manejar ciudad, datos y errores.

Si no hay conexión, se intenta responder desde caché utilizando la lógica implementada en el Service Worker.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

## 5.4 Implementación del Service Worker

El service-worker.js registra tres eventos clave:

* install: precachea archivos estáticos esenciales.
* activate: elimina versiones anteriores de caché.
* fetch: intercepta solicitudes a la red, responde desde caché si es posible, y cachea dinámicamente las respuestas nuevas.

El almacenamiento dinámico incluye las respuestas de la API de clima consultadas por el usuario. Se emplea control de versiones para evitar conflictos en actualizaciones.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

En este caso se muestra la consola del navegador mostrando logs del Service Worker durante instalación, activación y respuesta a solicitudes fetch.

## 5.5 Funcionalidad offline y offline.html

Cuando no hay conexión a internet y no se encuentran datos en el caché, el Service Worker responde con el archivo offline.html, que contiene un mensaje de error personalizado para el usuario. Esta medida garantiza una experiencia amigable incluso ante fallos de red.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

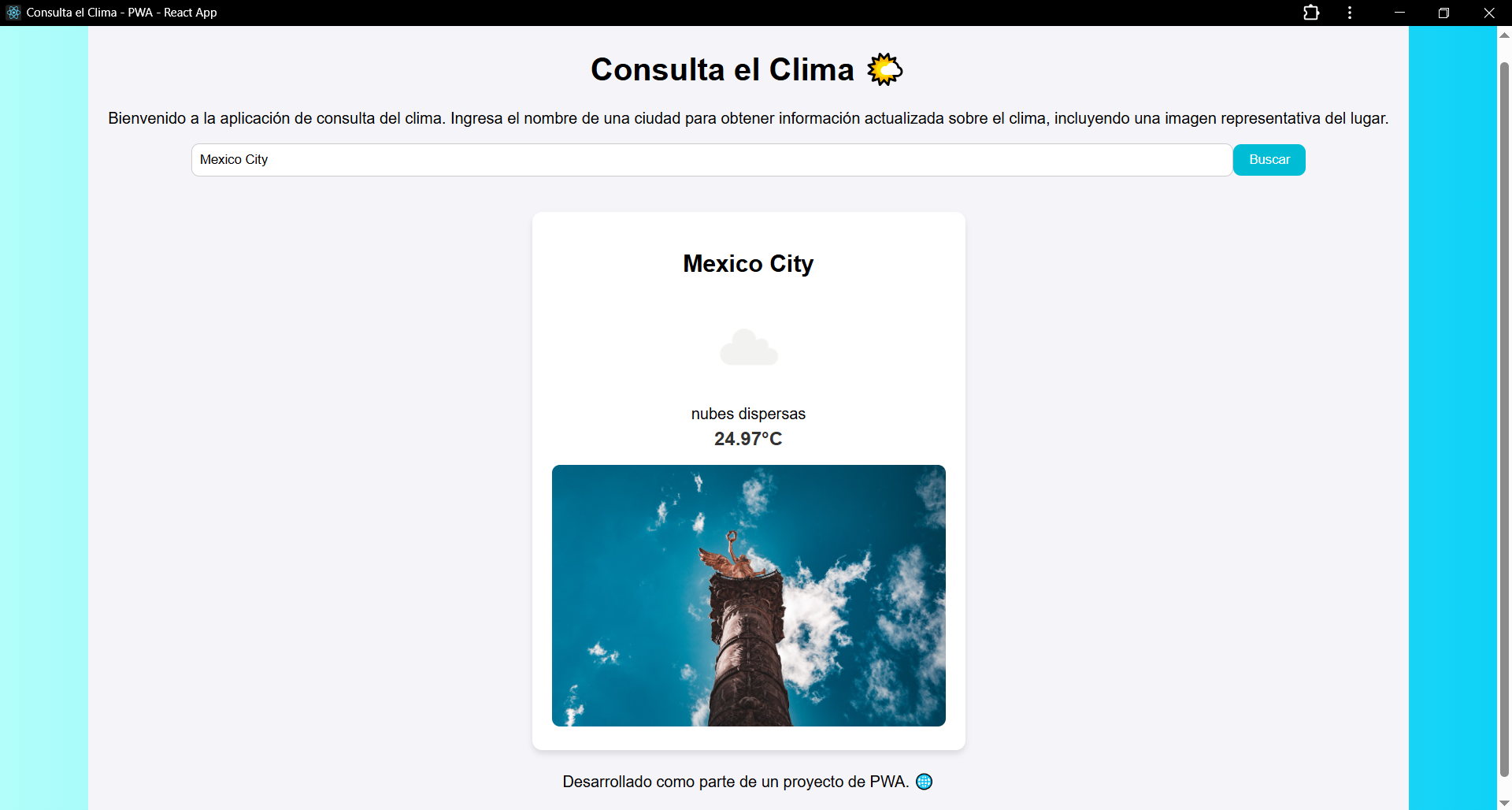
# 6. Resultados

Todos los códigos utilizados en esta práctica se encuentran en el siguiente repositorio en github.

## 6.1 Modo online

Durante la ejecución con conexión activa a Internet, se comprobó que:

* El usuario puede consultar el clima actual de cualquier ciudad ingresando su nombre en el formulario.
* La aplicación realiza una petición HTTP al backend, que a su vez consulta la API de OpenWeatherMap, retornando datos como temperatura, humedad y estado del clima.
* Las respuestas exitosas se almacenan automáticamente en el caché dinámico mediante el Service Worker, lo que permite su reutilización posterior en modo offline.



Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Se muestra la interfaz de la app mostrando los datos climáticos tras una búsqueda exitosa (por ejemplo, "Mexico City", “Tokyo”, “New York”, “Helsinki”).

También se tienen en cuenta aspectos como la responsividad, simulando el tamaño de una pantalla de celular.

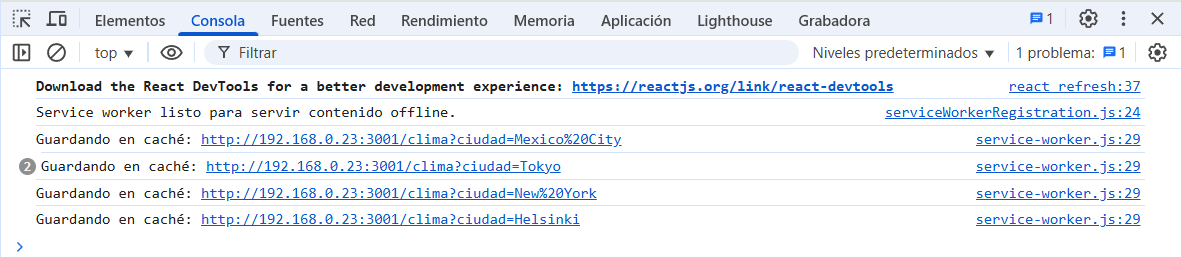
Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto. Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Es necesario mencionar que en este caso se le dio más prioridad al funcionamiento que al diseño, entonces este último aspecto se puede considerar para mejoras a futuro.

Del lado de la consola de Chrome se tiene lo siguiente con cada consulta del clima que es almacenada en caché preparando la desconexión y el funcionamiento del service – worker.



## 6.2 Modo offline

**6.2 Modo offline**

Al simular la pérdida de conexión mediante las herramientas de desarrollador (DevTools > Red > Sin Conexión), se observó el comportamiento esperado:

* Si el usuario consulta una ciudad **previamente buscada**, los datos se recuperan correctamente desde el **caché dinámico**, y se muestran sin requerir conexión.
* Si el usuario intenta consultar una **ciudad nueva** (no almacenada anteriormente), se muestra un **mensaje de error personalizado** indicando que no hay datos disponibles.
* Si el usuario accede a una ruta no registrada en caché (como una vista inexistente), el Service Worker responde con el archivo **offline.html**, informando que no hay conexión ni recursos disponibles.

Así, configuramos la red para que esta se desconecte y me permita verificar el funcionamiento del modo offline.

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

De esta forma se presentan las siguientes capturas para su correspondiente instancia.

* Resultado en pantalla para una ciudad almacenada sin conexión.

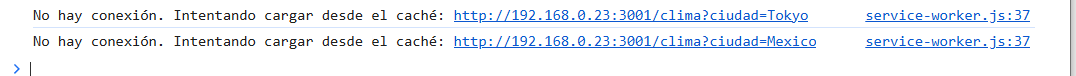
Para este primer caso se consulta Tokyo, ciudad que previamente ya habíamos consultado con el modo online. De esta forma, cuando realizamos la consulta se almacena en caché y posteriormente en este modo offline, se recupera. Esto es.

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.



Por su parte en consola se muestra la siguiente notificación.

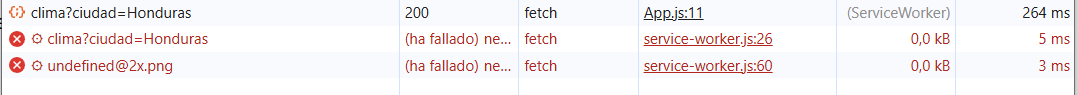


* Mensaje de error para una ciudad no cacheada.

Para este caso se consultó Honduras en un modo offline, el resultado fue el siguiente.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.



## 6.3 Almacenamiento en caché

El comportamiento del almacenamiento en caché fue inspeccionado utilizando DevTools en la pestaña **Application > Cache Storage**, donde se observaron dos cachés bien definidos:

* **Caché estática (precaché)**: contiene recursos esenciales como index.html, bundle.js, íconos y offline.html, registrados durante la instalación del Service Worker.
* **Caché dinámica**: almacena respuestas obtenidas desde la API de clima tras una solicitud exitosa.

También se validó la presencia de mecanismos de control de versiones para evitar conflictos al actualizar archivos entre sesiones.

## 6.4 Instalación como aplicación

Durante la prueba en navegadores compatibles (Chrome en Windows 10), se verificó que la aplicación cumple con los criterios para ser **instalada** como una app:

* El navegador detecta el archivo manifest.json y el Service Worker activo, lo que habilita la opción **“Instalar aplicación”** en la interfaz.
* Tras la instalación, la PWA se ejecuta en **modo standalone**, con apariencia similar a una aplicación nativa, sin barra de direcciones ni controles del navegador.
* Se comprobó su correcto funcionamiento en **modo escritorio**, accediendo a la aplicación desde el icono generado y consultando datos previamente almacenados en caché sin conexión.

# 7. Conclusión

La presente práctica permitió explorar y aplicar los principios fundamentales de una Aplicación Web Progresiva (PWA), combinando tecnologías del desarrollo web moderno con capacidades avanzadas como modo offline, instalación como app y almacenamiento en caché personalizado.

La solución implementada demostró ser resiliente, funcional y eficiente en diferentes escenarios de conectividad. Gracias al uso de Service Workers, fue posible interceptar solicitudes, gestionar almacenamiento en caché y proporcionar respuestas alternativas incluso sin red. Por su parte, el archivo manifest.json habilitó la instalación en distintos dispositivos, consolidando una experiencia similar a una app nativa.

El uso conjunto de React (frontend) y Node.js (backend) permitió mantener una arquitectura desacoplada, modular y escalable, ideal para futuras mejoras o integración con nuevas funcionalidades, como notificaciones push o sincronización en segundo plano.