Instituto Politécnico Nacional

Escuela Superior de Cómputo

Sistemas Distribuidos

Práctica 8:

“Progressive Web Apps”

Alumno:

Villarreal Razo Carlos Gabriel 2022630459

Grupo: 7CM1

Maestro: Carreto Arellano Chadwick

Fecha de realización: 2 de mayo de 2025

Fecha de entrega: 7 de mayo de 2025

Antecedentes.

En prácticas anteriores se exploraron arquitecturas distribuidas con servicios web y microservicios, permitiendo la creación de aplicaciones accesibles desde distintos entornos mediante APIs. No obstante, estas soluciones dependen completamente de la conexión a internet para su funcionamiento, por lo cual, ante esta limitación, surge la necesidad de mejorar la experiencia del usuario en aplicaciones web, especialmente en entornos con conectividad limitada o intermitente.

Una Progressive Web App (PWA) representa un enfoque moderno para el desarrollo de aplicaciones web que funcionan como si fueran aplicaciones nativas, éstas combinan lo mejor del desarrollo web y móvil, permitiendo que una aplicación web se instale en el dispositivo del usuario, funcione sin conexión a internet, y ofrezca una interfaz responsiva. Esto se logra mediante el uso de tecnologías clave como el Service Worker, que permite interceptar y manejar solicitudes de red, almacenar contenido en caché, y realizar sincronizaciones en segundo plano, así como el uso de un archivo manifest.json, que proporciona información sobre cómo debe comportarse la app al instalarse.

El service worker es un script en JavaScript que el navegador ejecuta en segundo plano, separado de la página web, lo cual permite realizar tareas que no requieren interacción directa con la interfaz de usuario, permitiendo agregar funcionalidades avanzadas como:

- Cacheo de archivos estáticos para que la aplicación funcione offline.
- Intercepción y control de solicitudes de red.
- Actualización en segundo plano de recursos.
- Sincronización en segundo plano de datos pendientes.
- Recepción de notificaciones push.

Al actuar como un proxy entre la red y la aplicación, el service worker puede decidir si servir los datos desde la red o desde la caché, ofreciendo una experiencia rápida y confiable incluso en condiciones de conectividad deficiente. Para utilizarlo, el sitio debe servirse desde HTTPS (excepto en localhost) y se debe registrar explícitamente desde el navegador del usuario.

Planteamiento del Problema

En aplicaciones web tradicionales, el funcionamiento depende completamente de una conexión activa a internet. Cuando esta conexión se pierde o es inestable, los usuarios no pueden acceder a la información ni realizar acciones importantes, como registrar datos o consultar contenido previamente disponible. Esta limitación afecta gravemente la experiencia del usuario, especialmente en entornos donde la conectividad es intermitente, como zonas rurales, espacios públicos o durante desplazamientos.

Por lo cual, se identificó la necesidad de desarrollar una solución que permita que una aplicación web pueda seguir funcionando aun cuando no haya conexión disponible, especificamente se requiere que los usuarios puedan consultar artículos previamente cargados, agregar nuevos en modo offline y que dichos datos se sincronicen automáticamente con el servidor cuando se restablezca la conexión.

Propuesta de Solución.

Se busca desarrollar una Progressive Web App (PWA) que permita a los usuarios consultar y agregar artículos desde una interfaz web, incluso cuando no haya conexión a internet. Esta solución integrará tecnologías modernas como Service Worker, IndexedDB y Background Sync, con el fin de brindar una experiencia de uso fluida, confiable y adaptable a las condiciones de red.

El sistema estará compuesto por una API construida con Flask, que se encargará de servir los artículos existentes y registrar nuevos artículos enviados por el cliente. En el lado del cliente, se desarrollará una interfaz web que utilizará un Service Worker para cachear archivos estáticos y gestionar peticiones de red. Cuando no haya conexión, los artículos nuevos se almacenarán localmente en IndexedDB, mediante la biblioteca idb-keyval, y se sincronizarán automáticamente con el servidor en segundo plano cuando la conexión sea restablecida, utilizando la funcionalidad de Background Sync del navegador.

Materiales y Métodos.

Hardware.

- Procesador Inter Core i5 de 11ª generación.
- Tarjeta gráfica NVIDIA GTX 1650.
- 16 GB de RAM DDR4.

Software.

- Lenguaje: Python 3.11 y JavaScript.
- Visual Studio Code (VSCode).
- Compilador y ejecución a través de la terminal integrada de VSCode.

Paquetes/Bibliotecas.

- idb-keyval.
- Fetch API.
- Background Sync.
- Flask.
- Jsonify.

Métodos de app.py.

- **GET /api/articulos:** retorna una lista de artículos en formato JSON.

- **POST /api/AgregarArticulos:** recibe un artículo en formato JSON y lo agrega a la lista o base de datos simulada; devuelve un identificador de confirmación.
- **pagarUsuario():** Permite agregar un usuario a la cola de pagos que será consumida por otro microservicio.

Métodos service-worker.js.

- **install:** cachea archivos estáticos esenciales al momento de la instalación del Service Worker.
- **activate:** elimina cachés antiguos para mantener solo la versión actual.
- **fetch:** intercepta todas las peticiones de red; sirve recursos desde la caché si no hay conexión y guarda en caché las respuestas de API dinámicas.
- **sync:** escucha el evento sync-articulos para enviar artículos guardados en IndexedDB cuando vuelve la conexión.
- **syncArticulos():** función que recupera artículos almacenados y los reenvía al servidor, luego limpia la base local.

Métodos service-worker.js.

- **registerSync():** registra una tarea de sincronización en segundo plano (sync-articulos) cuando se guarda un artículo offline.
- **saveForSync(articulo):** guarda un nuevo artículo localmente en IndexedDB para sincronización posterior.
- **DOMContentLoaded:** al cargar la página, recupera y muestra la lista de artículos desde el servidor, o caché si no hay red.
- **beforeinstallprompt:** muestra un botón personalizado para instalar la app cuando el navegador detecta que la PWA es instalable.

Desarrollo de la Solución.

Para la implementación de esta práctica se desarrolló una Progressive Web App (PWA) compuesta por un backend en Flask y una interfaz web que se comporta como una aplicación nativa. El servidor, implementado en app.py, expone una API REST sencilla con dos rutas principales: una para consultar los artículos registrados y otra para agregar nuevos. Esta API es consumida por el frontend definido en index.html y controlado por app.js, el cual maneja la lógica del cliente, incluida la visualización de artículos, el envío de nuevos datos y la gestión de la instalación de la aplicación como una app independiente mediante un botón personalizado.

app.py.

```python
from flask import Flask, render_template, jsonify, request

app = Flask(__name__)

articulos_db = [
    {"id": 1, "titulo": "Papas sol"},
    {"id": 2, "titulo": "Teclado mecánico"},
    {"id": 3, "titulo": "Bolsa totis surtidos"}
]

@app.route('/')
def home():
    return render_template('index.html')

@app.route('/api/articulos')
def articulos():
    return jsonify(articulos_db)

@app.route('/api/AgregarArticulos', methods=['POST'])
def agregar_articulo():
    articulo = request.get_json()

    #Asignar un nuevo ID al artículo
    nuevo_id = 1
    if articulos_db:
        nuevo_id = max(item["id"] for item in articulos_db) + 1
```

```
    articulo["id"] = nuevo_id

    #Guardamos el articulo en la "base de datos"
    articulos_db.append(articulo)

    print(f"Artículo agregado: {articulo}")
    print(f"Total de artículos: {len(articulos_db)}")

    return jsonify({"status": "ok", "id": nuevo_id}), 201


if __name__ == '__main__':
    app.run(debug=True)
```

index.html.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>PWA con Flask</title>
    <link rel="manifest" href="{{url_for('static', filename='manifest.json')}}">
    <meta name="theme-color" content="#317EFB"/>
    <script src="https://cdn.jsdelivr.net/npm/idb-keyval@6/dist/idb-keyval-
iife.min.js"></script>
    <link rel="stylesheet" href="{{url_for('static', filename='css/styles.css')}}">
</head>
<body>
    <h1>Hola desde una PWA con Flask</h1>
    <hr>

    <h1>Artículos</h1>
    <ul id="lista-articulos"></ul>

    <h2>Agregar artículo</h2>
    <form id="form-articulo">
        <input type="text" id="titulo" placeholder="Título del artículo" required>
        <button type="submit">Agregar</button>
    </form>

    <!-- <link rel="manifest" href="{{url_for('static', filename='manifest.json')}}"> -->
    <link rel="icon" href="{{url_for('static', filename='icon.png')}}">
    <div id = "status"></div>
    <script src="{{url_for('static', filename='js/app.js')}}"></script>
```

```
</body>
</html>
```

app.js.

```
if ('serviceWorker' in navigator) {
    navigator.serviceWorker.register('/static/service-worker.js')
    .then(() => {
        console.log("Service Worker registrado");
    }).catch((error) => {
        console.log("Error al registrar el Service Worker:", error);
    });
}

let deferredPrompt;

window.addEventListener('beforeinstallprompt', (e) => {
    e.preventDefault();
    deferredPrompt = e;

    //Botón personalizado para instalar la app
    const installBtn = document.createElement('button');
    installBtn.textContent = 'Instalar App';
    document.body.appendChild(installBtn);

    installBtn.addEventListener('click', () => {
        deferredPrompt.prompt();
        deferredPrompt.userChoice.then((choiceResult) => {
            if (choiceResult.outcome === 'accepted') {
                console.log('Instalación aceptada');
            } else {
                //Borramos el botón si el usuario cancela la instalación
                installBtn.remove();
                console.log('Instalación cancelada');
            }
            deferredPrompt = null;
        });
    });
});

document.addEventListener('DOMContentLoaded', () => {
    console.log("DOM cargado, obteniendo artículos...");
    const statusElement = document.getElementById('status');
    statusElement.textContent = "Cargando artículos...";
```

```
    fetch('/api/articulos')
        .then(res => {
            if (!res.ok) {
                throw new Error('Error en la respuesta del servidor: ' + res.status);
            }
            return res.json();
        })
        .then(data => {
            console.log("Artículos recibidos:", data);
            const lista = document.getElementById('lista-articulos');
            lista.innerHTML = ''; // Limpiar lista existente

            if (data.length === 0) {
                statusElement.textContent = "No hay artículos disponibles";
                return;
            }

            data.forEach(art => {
                const li = document.createElement('li');
                li.textContent = art.titulo;
                lista.appendChild(li);
            });

            statusElement.textContent = `${data.length} artículos cargados`;
        })
        .catch(err => {
            console.error('Error al cargar artículos:', err);
            statusElement.textContent = "Error al cargar artículos. ¿Quizá offline?";
        });
});

//Para agregar un artículo cuando haya conexión
document.getElementById('form-articulo').addEventListener('submit', async (e) => {
    e.preventDefault();
    const tituloInput = document.getElementById('titulo');
    const titulo = tituloInput.value.trim();

    if (!titulo) {
        alert('Por favor ingrese un título');
        return;
    }

    const articulo = { titulo };
    const statusElement = document.getElementById('status');
    statusElement.textContent = "Enviando artículo...";
```

```
    try {
        console.log("Intentando enviar artículo:", articulo);
        const response = await fetch('/api/AgregarArticulos', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(articulo)
        });

        if (response.ok) {
            const result = await response.json();
            console.log("Respuesta del servidor:", result);

            //Agregamos el artículo a la lista
            const lista = document.getElementById('lista-articulos');
            const li = document.createElement('li');
            li.textContent = articulo.titulo;
            lista.appendChild(li);

            //Limpiamos el campo de entrada
            tituloInput.value = '';

            statusElement.textContent = `Artículo "${titulo}" agregado con éxito (ID:
${result.id})`;
            alert('Artículo enviado correctamente');
        }else{
            throw new Error('Respuesta no exitosa del servidor: ' + response.status);
        }
    } catch (err) {
        console.error('Error al enviar artículo:', err);
        statusElement.textContent = "Sin conexión. Guardando para sincronización...";

        try {
            await saveForSync(articulo);

            // Aun sin conexión, agregamos el artículo a la lista
            const lista = document.getElementById('lista-articulos');
            const li = document.createElement('li');
            li.textContent = articulo.titulo + ' (pendiente de sincronización)';
            lista.appendChild(li);

            tituloInput.value = '';

            registerSync();
```

```
            statusElement.textContent = `Artículo "${titulo}" guardado para sincronización
posterior`;
            alert('Guardado para envío posterior');
        } catch (storageErr) {
            console.error('Error al guardar para sincronización:', storageErr);
            statusElement.textContent = "Error al guardar el artículo";
            alert('Error al guardar el artículo');
        }
    }
});

async function saveForSync(articulo) {
    console.log("Guardando para sincronización:", articulo);
    try {
        const store = new idbKeyval.Store('articulos-db', 'articulos-store');
        const prev = await idbKeyval.get('pendientes', store) || [];
        prev.push(articulo);
        await idbKeyval.set('pendientes', prev, store);
        console.log("Guardado exitoso para sincronización");
    } catch (err) {
        console.error("Error en saveForSync:", err);
        throw err;
    }
}

function registerSync() {
    if ('serviceWorker' in navigator && 'SyncManager' in window) {
        navigator.serviceWorker.ready.then(sw => {
            return sw.sync.register('sync-articulos')
                .then(() => {
                    console.log('Sincronización registrada correctamente');
                })
                .catch(err => {
                    console.error('Error al registrar sincronización:', err);
                });
        });
    } else {
        console.warn('Background Sync no está soportado en este navegador');
    }
}
```

En el lado del cliente, se implementó un Service Worker (service-worker.js) que gestiona el almacenamiento en caché de recursos estáticos para el funcionamiento offline, y permite la sincronización en segundo plano de

artículos almacenados localmente usando IndexedDB (mediante idb-keyval). Cuando no hay conexión, los artículos son guardados en local y luego sincronizados automáticamente cuando la conexión se restablece, gracias al uso de la API Background Sync del navegador. Esta arquitectura permite a la aplicación seguir funcionando incluso sin acceso a internet, cumpliendo con los principios fundamentales de una PWA: fiabilidad, velocidad y capacidad de instalación.

Service-worker.js.

```
const CACHE_NAME = 'pwa-cache-v1';
const STATIC_FILES = [
    '/',
    '/static/js/app.js',
    '/static/manifest.json',
    '/static/icon.png'
];

//Cache estático en instalación
self.addEventListener('install', (event) => {
    event.waitUntil(
        caches.open(CACHE_NAME).then((cache) => cache.addAll(STATIC_FILES))
    );
    console.log('ServiceWorker Instalado');
});

//Para limpiar cachés viejos
self.addEventListener('activate', (event) => {
    event.waitUntil(
        caches.keys().then((keys) =>
            Promise.all(keys.map((key) => {
                if (key !== CACHE_NAME) return caches.delete(key);
            }))
        )
    );
    console.log('ServiceWorker Activado');
});

//Cache dinámico en fetch
self.addEventListener('fetch', (event) => {
    event.respondWith(
```

```
        caches.match(event.request).then((cachedResponse) => {
            if (cachedResponse) return cachedResponse;

            return fetch(event.request).then((networkResponse) => {
                if (
                    event.request.url.includes('/api/')
                ) {
                    return caches.open(CACHE_NAME).then((cache) => {
                        cache.put(event.request, networkResponse.clone());
                        return networkResponse;
                    });
                }

                return networkResponse;
            }).catch(() => {
                //Devolvemos una respuesta por defecto si no hay conexión
                // if (event.request.mode === 'navigate') {
                //     return caches.match('/offline.html');
                // }
            });
        })
    );
});

//Para cargar articulos automaticamente cuando haya conexión
self.addEventListener('sync', function(event) {
    if (event.tag === 'sync-articulos') {
        event.waitUntil(syncArticulos());
    }
});

async function syncArticulos() {
    const stored = await getLocalArticulos();

    for (const articulo of stored) {
        try {
            await fetch('/api/AgregarArticulos', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify(articulo)
            });
        } catch (e) {
            console.error('Error al enviar artículo en segundo plano:', e);
```

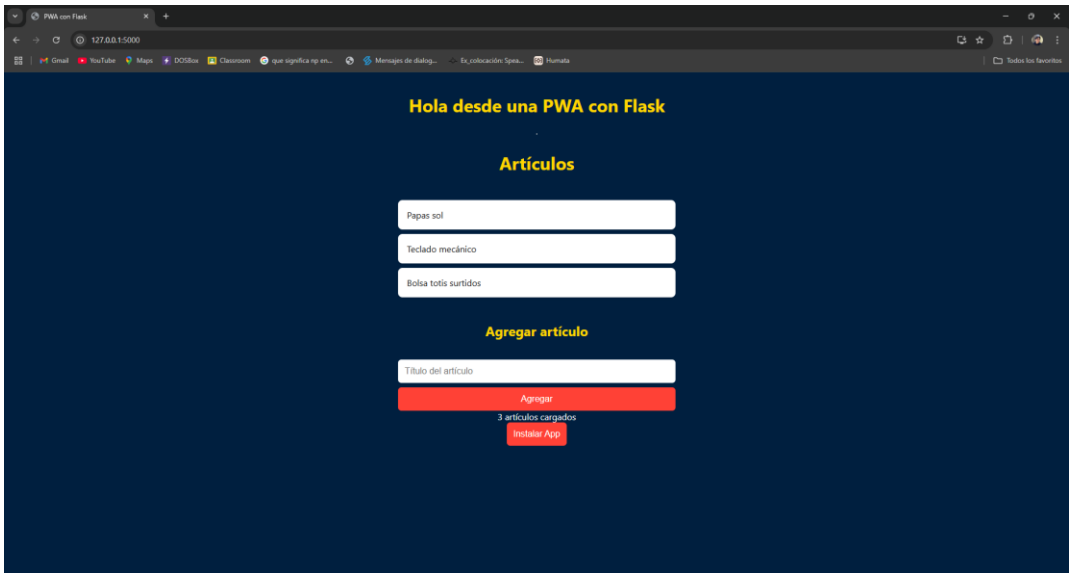
```
        }
    }

    await clearLocalArticulos();
}

//Almacenar en IndexedDB
importScripts('https://cdn.jsdelivr.net/npm/idb-keyval@6/dist/idb-keyval-iife.min.js');
const store = new idbKeyval.Store('articulos-db', 'articulos-store');

function getLocalArticulos() {
    return idbKeyval.get('pendientes', store).then(data => data || []);
}

function clearLocalArticulos() {
    return idbKeyval.del('pendientes', store);
}
```

Link del Repositorio de GitHub de la Práctica.

https://github.com/ClsRazo/PracticaPWA.git

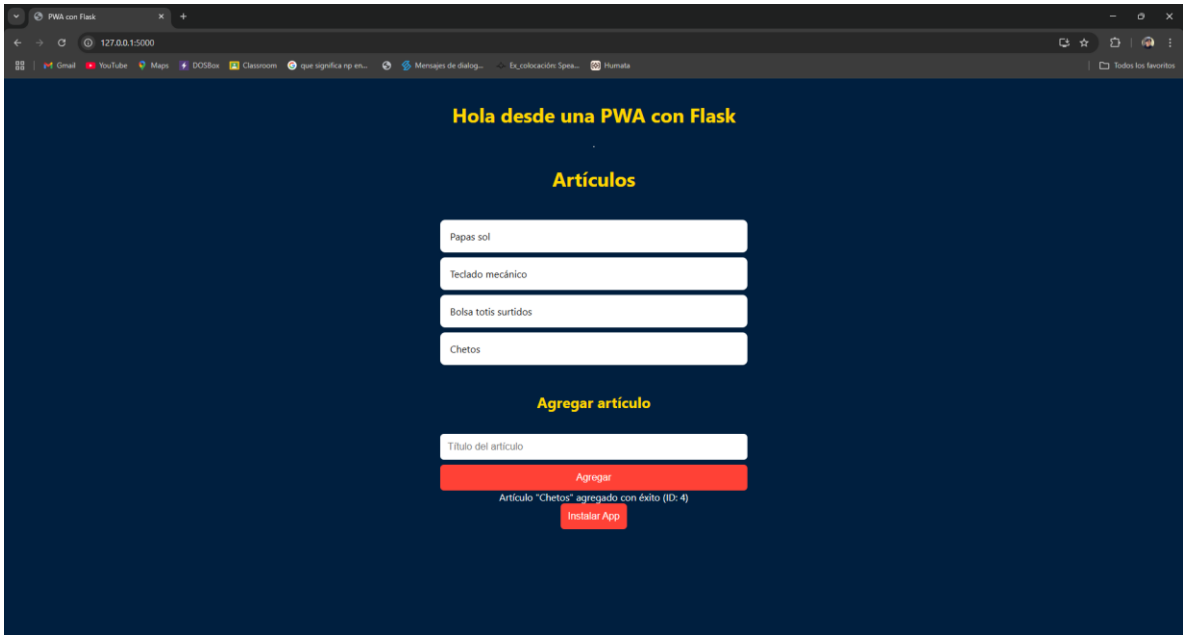
Resultados.

Aplicación desde el navegador:



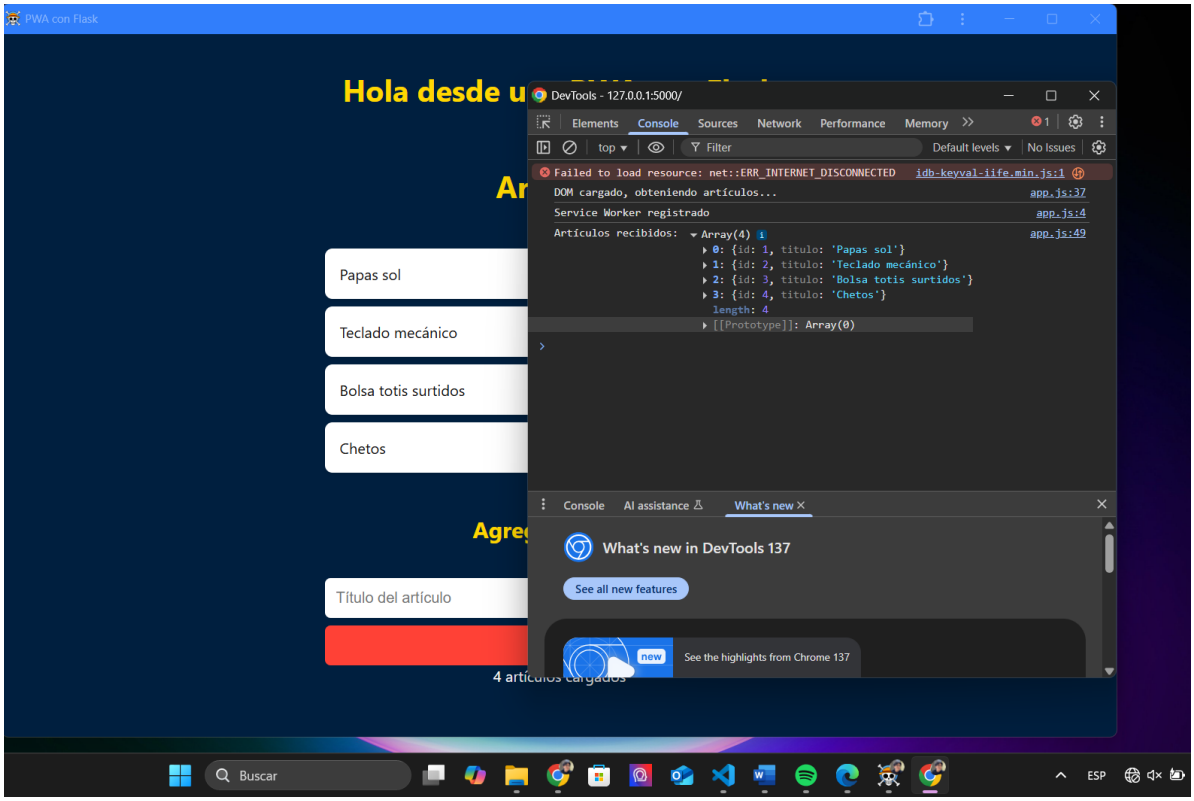
Se puede observar que el navegador la reconoce como PWA, ya que, permite descargarla con un botón ubicado en la parte superior derecha de la ventana.

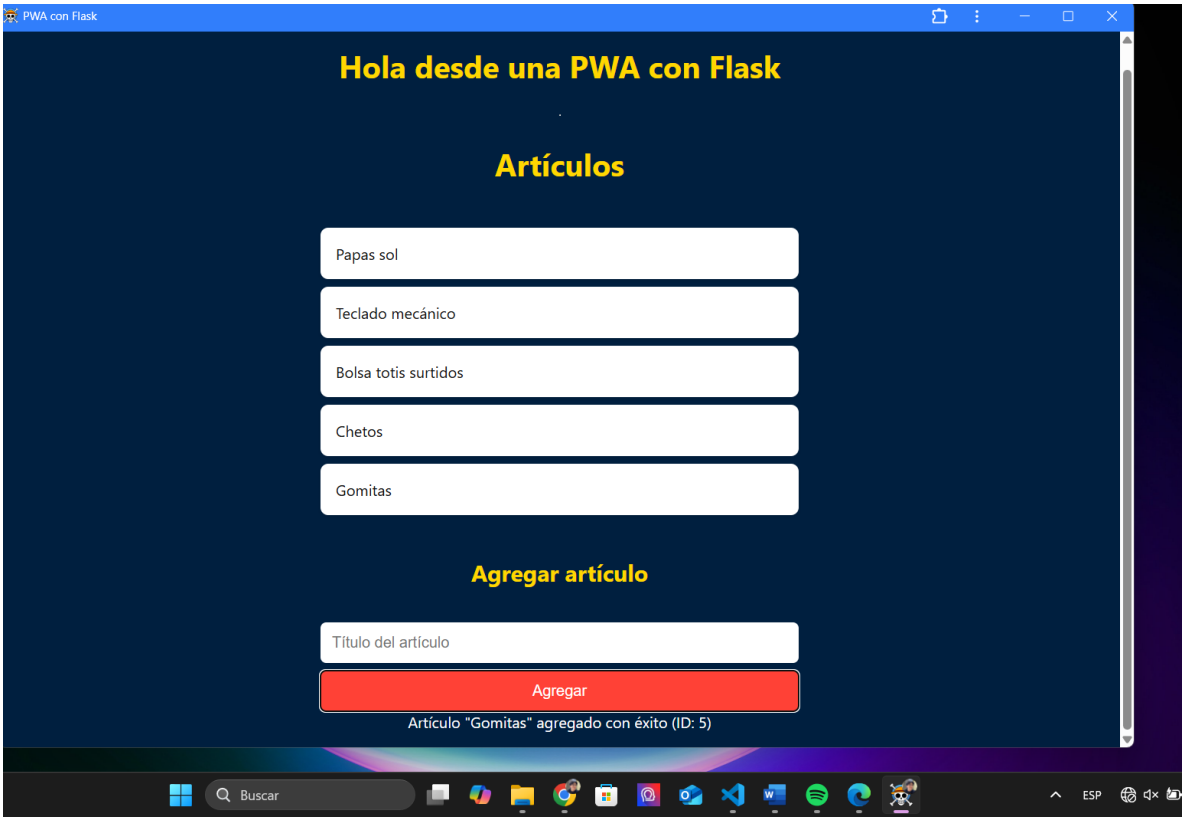
Además, se pueden ver artículos existentes en la "base de datos" y se pueden agregar nuevos:



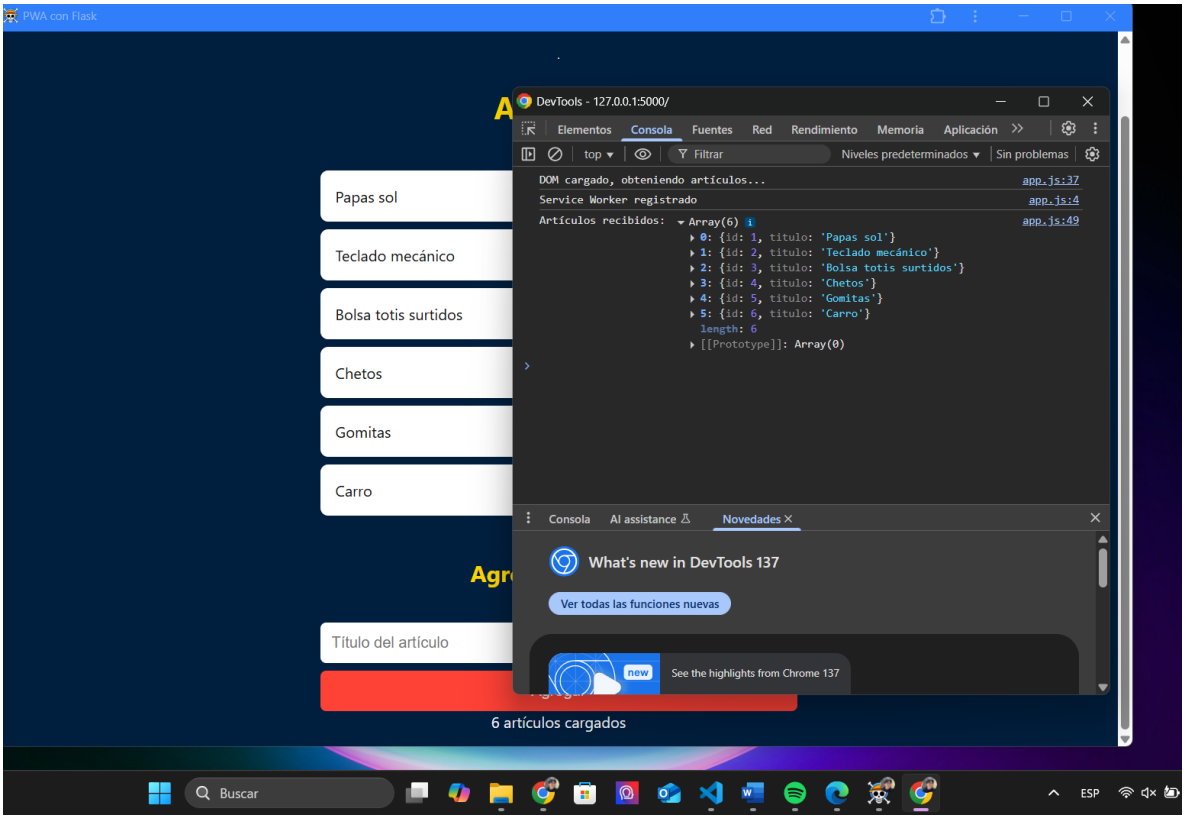
Al descargar la app:

También, se pueden visualizar los artículos existentes sin conexión y agregar nuevos, los cuales, se cargarán una vez se recupere la conexión.





Después de volvernos a conectar a internet:



Conclusiones.

Derivado de la culminación de la práctica, logré comprender cómo una Progressive Web App (PWA) puede mejorar significativamente la experiencia del usuario al permitir que una aplicación web siga funcionando incluso sin conexión a internet. A través de la integración de elementos como el Service Worker, bibliotecas como IndexedDB y Background Sync, etc., fue posible implementar una solución que cachea recursos, almacena información localmente y la sincroniza automáticamente cuando se restablece la conexión, simulando el comportamiento de una aplicación nativa moderna.

Además, la incorporación de un botón personalizado para instalar la aplicación, junto con el uso de manifest.json, me permitió ver cómo una PWA puede integrarse de forma natural en el dispositivo del usuario, como si se tratara de una app móvil, además, me sorprendió como el mismo navegador ya reconoce la PWA.

Esta práctica me ayudó a consolidar conocimientos sobre desarrollo web avanzado y sobre cómo crear aplicaciones más robustas, responsivas y tolerantes a fallos, adaptándose a diferentes condiciones de red y mejorando la disponibilidad del sistema en todo momento.