SPRAWOZDANIE

**Zadanie 4 – Stwórz i opublikuj własną aplikację PWA**

Adam Zajler, 131892

chx94178@student.chorzow.merito.pl

Spis treści

[1. Wstęp 2](#_Toc198485589)

[1.1. Wprowadzenie do Progressive Web Apps (PWA) 2](#_Toc198485590)

[1.2. Cel projektu 2](#_Toc198485591)

[2. Opis Funkcjonalności Aplikacji 3](#_Toc198485592)

[2.1. Główne funkcje 3](#_Toc198485593)

[2.2 Działanie w trybie offline 8](#_Toc198485594)

# Wstęp

## 1.1. Wprowadzenie do Progressive Web Apps (PWA)

Progresywne Aplikacje Internetowe (PWA) to nowoczesne podejście do tworzenia aplikacji webowych, które łączą najlepsze cechy stron internetowych i aplikacji natywnych. Dzięki wykorzystaniu nowoczesnych technologii webowych, takich jak Service Workery, Local Storage, IndexedDB, PWA oferują użytkownikom doświadczenia zbliżone do aplikacji instalowanych na urządzeniach mobilnych.   
Kluczowe cechy PWA to m.in. responsywność, możliwość pracy w trybie offline, instalowalność na ekranie głównym urządzenia oraz wysoka wydajność.

## 1.2. Cel projektu

Celem niniejszego projektu było zaprojektowanie, zaimplementowanie oraz wdrożenie pełnoprawnej aplikacji PWA typu "Lista Zadań" (ToDo List). Aplikacja została stworzona w czystym języku JavaScript, bez użycia frameworków jak React, Vue czy Angular.   
Projekt zakładał implementację kluczowych mechanizmów PWA, w tym pliku manifestu, własnego Service Workera, wsparcia dla trybu offline poprzez buforowanie zasobów, integrację z IndexedDB do przechowywania danych lokalnie oraz interakcję z zewnętrznym API. Końcowym etapem było hostowanie aplikacji na darmowym serwerze oraz przygotowanie szczegółowego sprawozdania dokumentującego proces realizacji.

# 2. Opis Funkcjonalności Aplikacji

## 2.1. Główne funkcje

Aplikacja została zaprojektowana jako proste, ale funkcjonalne narzędzie do zarządzania codziennymi zadaniami. Główne funkcje aplikacji obejmują:

1. Zarządzanie zadaniami (Strona główna - index.html):
   1. Wyświetlanie listy zadań pobranych z lokalnej bazy danych IndexedDB.
   2. Możliwość dodawania nowych zadań za pomocą formularza "szybkiego dodawania" bezpośrednio na stronie głównej.
   3. Oznaczanie zadań jako ukończone (wizualne przekreślenie, aktualizacja statusu w IndexedDB).
   4. Usuwanie zadań z listy (oraz z IndexedDB).
   5. Wyświetlanie informacji o terminie i opisie zadania (jeśli zostały dodane).  
      Obraz zawierający tekst, zrzut ekranu, wyświetlacz, oprogramowanie

      Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, System operacyjny

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, System operacyjny

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

1. Dodawanie szczegółowych zadań (Formularz – form.html):
   1. Dedykowana podstrona z rozbudowanym formularzem pozwalającym na dodanie zadania z tytułem, opisem, terminem wykonania oraz priorytetem
   2. Prosta walidacja danych formularza (np. wymagany tytuł).
   3. Zapisywanie nowych zadań w bazie IndexedDB.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Ikona komputerowa

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, numer

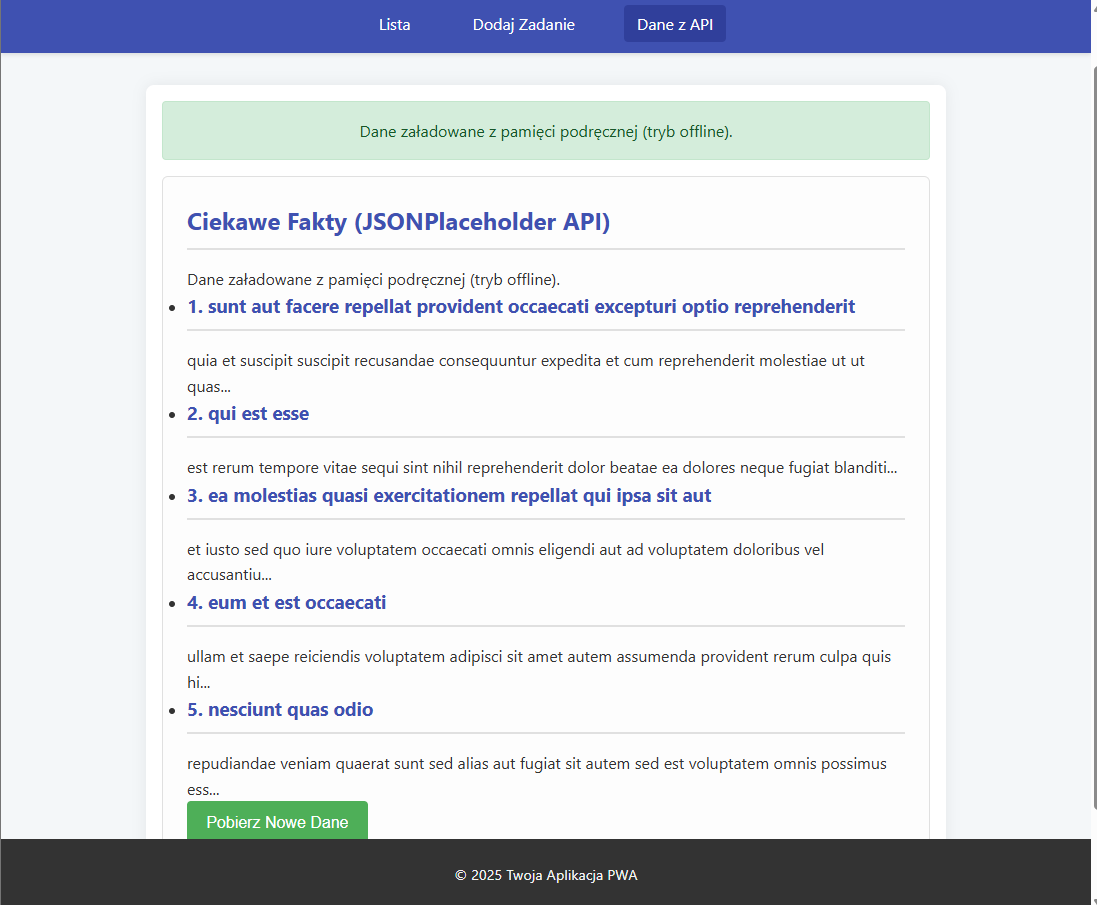
Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.  
Obraz zawierający tekst, zrzut ekranu, oprogramowanie, System operacyjny

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

1. Wyświetlanie danych z zewnętrznego API (Dane z API – api.data.html):
   1. Integracja z publicznym API (np. JSONPlaceholder) w celu pobrania i wyświetlenia przykładowych danych (np. listy postów).
   2. Możliwość ręcznego odświeżenia danych z API.
   3. Implementacja mechanizmu buforowania odpowiedzi API, umożliwiająca przeglądanie danych w trybie offline.

Obraz zawierający tekst, elektronika, zrzut ekranu, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.



## 2.2 Działanie w trybie offline

Aplikacja została zaprojektowana z myślą o działaniu w trybie offline:

* Buforowanie zasobów aplikacji: Za pomocą Service Workera, kluczowe zasoby aplikacji (pliki HTML, CSS, JavaScript, manifest, ikony) są buforowane podczas pierwszej wizyty. Umożliwia to uruchomienie i nawigację po aplikacji bez dostępu do internetu.
* Lokalne przechowywanie danych: Zadania użytkownika są przechowywane w IndexedDB, co oznacza, że są dostępne i modyfikowalne nawet w trybie offline. Wszelkie zmiany są zapisywane lokalnie.
* Buforowanie danych z API: Dane pobierane z zewnętrznego API są również buforowane. Przy braku połączenia aplikacja próbuje załadować ostatnio pobrane dane z pamięci podręcznej.

# 3. Opis Techniczny Krok po Kroku

## 3.1. Konfiguracja środowiska

Do realizacji projektu wykorzystano następujące narzędzia i technologie:

* Edytor kodu: Webstorm
* Przeglądarka internetowa (testowa): Google Chrome (najnowsza wersja)
* Języki programowania: HTML5, CSS3, JavaScript (ES6+).
* System kontroli wersji: Git

## 3.2. Struktura projektu

Projekt został zorganizowany w następujący prosty sposób:

* index.html - Strona główna aplikacji
* form.html - Strona z formularzem dodawania zadań
* api-data.html - Strona wyświetlająca dane z API
* style.css - Główny arkusz stylów
* app.js - Główny plik JS aplikacji
* manifest.json - Plik manifestu PWA
* service-worker.js - Skrypt Service Workera
* js - Folder na dodatkowe skrypty JS
* form.js - Logika formularza z form.html i IndexedDB
* api.js - Logika pobierania i buforowania danych API

## 3.3. Tworzenie i podłączenie manifestu

Plik **manifest.json** dostarcza przeglądarce metadanych o aplikacji, umożliwiając jej "instalację" i kontrolę nad sposobem wyświetlania.

Zawartość przykładowego pliku **manifest.json**:  
Obraz zawierający tekst, zrzut ekranu, oprogramowanie, komputer

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

Podłączenie do HTML: Manifest jest podłączany do każdego pliku HTML w sekcji **<head>**:

Obraz zawierający tekst, zrzut ekranu, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

## 3.4. Tworzenie i rejestracja Service Workera

Service Worker (SW) to skrypt działający w tle, niezależnie od strony internetowej, umożliwiający m.in. przechwytywanie żądań sieciowych, zarządzanie pamięcią podręczną i obsługę powiadomień push.

Kluczowe elementy implementacji SW:

* Instalacja: Podczas instalacji SW, zdefiniowana lista zasobów (np. HTML, CSS, JS, ikony) jest pobierana i zapisywana w pamięci podręcznej (Cache API). Użyto **self.skipWaiting()** aby nowy SW aktywował się szybciej.
* Aktywacja: Podczas aktywacji, SW usuwa stare, nieużywane wersje pamięci podręcznej i przejmuje kontrolę nad klientami (self.clients.claim() ).
* Przechwytywanie żądań (fetch): SW przechwytuje wszystkie żądania sieciowe pochodzące z aplikacji. Dla zasobów zdefiniowanych w **urlsToCache** zastosowano strategię \*Cache First\*: najpierw sprawdzana jest pamięć podręczna; jeśli zasób tam jest, jest zwracany. W przeciwnym razie, zasób jest pobierany z sieci, a następnie dodawany do cache.
* Rejestracja Service Workera (w `app.js`):  
  Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

  Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

## 3.5. Integracja z API

Strona `api-data.html` integruje się z zewnętrznym API (JSONPlaceholder) w celu pobrania i wyświetlenia danych.

\* \*\*Pobieranie danych:\*\* Użyto standardowego `fetch` API do wysyłania żądań GET.

\* \*\*Wyświetlanie danych:\*\* Pobrane dane (posty) są dynamicznie renderowane na stronie.

\* \*\*Buforowanie odpowiedzi API:\*\* Zaimplementowano własną logikę buforowania w `js/api.js` (niezależnie od głównego SW dla zasobów aplikacji):

\* Odpowiedzi z API są zapisywane w dedykowanej pamięci podręcznej (`api-data-cache-v1`) przy użyciu Cache API.

\* Przy braku połączenia lub błędzie sieci, aplikacja próbuje załadować dane z tej pamięci podręcznej.

\* Przycisk "Pobierz Nowe Dane" pozwala na wymuszenie próby pobrania świeżych danych z sieci.

### 3.6. Strategia cache i konfiguracja offline

\* \*\*Zasoby aplikacji (App Shell):\*\* Strategia \*Cache First\* zaimplementowana w `service-worker.js`. Kluczowe pliki są buforowane podczas instalacji SW.

```javascript

- fragment z service-worker.js

self.addEventListener('fetch', event => {

event.respondWith(

caches.match(event.request)

.then(response => {

return response || fetch(event.request).then(networkResponse => {

- Opcjonalne dodawanie do cache odpowiedzi sieciowych dla zasobów z urlsToCache

if (event.request.method === 'GET' && urlsToCache.includes(new URL(event.request.url).pathname)) {

const responseToCache = networkResponse.clone();

caches.open(CACHE\_NAME).then(cache => {

cache.put(event.request, responseToCache);

});

}

return networkResponse;

});

})

.catch(() => { /\* obsługa błędu, np. strona fallback \*/ })

);

});

```

\* \*\*Dane dynamiczne (API):\*\* Strategia \*Network first, then cache\* (z ręcznym zarządzaniem w `js/api.js`). Aplikacja najpierw próbuje pobrać dane z sieci. Jeśli się uda, aktualizuje cache. Jeśli sieć zawiedzie, próbuje odczytać z cache.

\* \*\*Dane użytkownika (Zadania):\*\* Przechowywane w IndexedDB, co zapewnia ich dostępność i modyfikowalność offline. Zmiany są synchronizowane z IndexedDB.

### 3.7. Wdrożenie na hostingu

Aplikacja PWA musi być hostowana na serwerze obsługującym \*\*HTTPS\*\*, co jest warunkiem koniecznym do działania Service Workerów.

\* \*\*Wybór platformy:\*\* Możliwe darmowe opcje to Netlify, GitHub Pages, Firebase Hosting.

\* \*\*Proces wdrożenia (ogólny):\*\*

1. Przygotowanie kodu produkcyjnego (upewnienie się, że wszystkie ścieżki są poprawne).

2. Utworzenie konta na wybranej platformie.

3. Wgranie plików aplikacji na serwer (np. przez Git push dla Netlify/GitHub Pages, lub przez CLI dla Firebase).

4. Konfiguracja domeny (jeśli dotyczy) i upewnienie się, że HTTPS jest aktywne.

5. Testowanie wdrożonej aplikacji.

\* \*\*Potencjalne błędy:\*\* Problemy ze ścieżkami zasobów, konfiguracją serwera (np. typy MIME), działaniem SW w środowisku produkcyjnym.

## 4. Opis Trudności i Ich Rozwiązań

Podczas realizacji projektu napotkano kilka typowych trudności związanych z tworzeniem PWA:

1. \*\*Problem: Cykl życia Service Workera i aktualizacje.\*\*

\* \*\*Opis:\*\* Service Worker po zarejestrowaniu i aktywacji może nie odświeżać się natychmiast po wprowadzeniu zmian w jego kodzie lub w buforowanych plikach. Użytkownik może widzieć starą, zbuforowaną wersję aplikacji.

\* \*\*Rozwiązanie:\*\*

\* Zmiana nazwy pamięci podręcznej (`CACHE\_NAME` w `service-worker.js`) przy każdej istotnej aktualizacji zasobów. W evencie `activate`, stary cache jest usuwany.

\* Użycie `self.skipWaiting()` w evencie `install` oraz `self.clients.claim()` w evencie `activate` aby nowy Service Worker przejął kontrolę szybciej.

\* Podczas developmentu, w narzędziach deweloperskich przeglądarki, można wymusić aktualizację ("Update on reload") i odznaczyć opcję "Bypass for network" dla Service Workera.

\* \*\*Czego się nauczyłem:\*\* Zrozumienie cyklu życia Service Workera jest kluczowe. Mechanizmy aktualizacji muszą być starannie zaplanowane, aby użytkownicy otrzymywali najnowsze wersje aplikacji bezproblemowo.

2. \*\*Problem: Zarządzanie IndexedDB i operacje asynchroniczne.\*\*

\* \*\*Opis:\*\* IndexedDB API jest w pełni asynchroniczne i oparte na zdarzeniach/requestach, co może być skomplikowane w zarządzaniu, szczególnie przy wielu operacjach (CRUD). Debugowanie i obsługa błędów wymaga uwagi.

\* \*\*Rozwiązanie:\*\*

\* Strukturyzacja kodu obsługującego IndexedDB w dedykowane funkcje (np. `initDB`, `addTaskToDB`, `getAllTasksFromDB`).

\* Użycie `Promise` do opakowania operacji IndexedDB, co ułatwia pracę z kodem asynchronicznym (np. użycie `async/await` w miejscach wywołania tych funkcji).

\* Dokładna obsługa zdarzeń `onsuccess`, `onerror`, `onupgradeneeded` dla każdej transakcji i requestu.

\* Korzystanie z narzędzi deweloperskich przeglądarki (zakładka Application -> IndexedDB) do inspekcji zawartości bazy.

\* \*\*Czego się nauczyłem:\*\* Praca z IndexedDB wymaga cierpliwości i dobrego zrozumienia jej asynchronicznej natury. Obietnice (Promises) znacznie upraszczają kod i zarządzanie przepływem sterowania.

3. \*\*Problem: Logika buforowania dla dynamicznych danych API.\*\*

\* \*\*Opis:\*\* Standardowa strategia \*Cache First\* (dobra dla statycznych zasobów aplikacji) nie jest optymalna dla danych z API, które mogą się często zmieniać. Trzeba było zaimplementować bardziej elastyczne podejście.

\* \*\*Rozwiązanie:\*\*

\* Dla danych API zaimplementowano w `js/api.js` strategię, która najpierw próbuje pobrać dane z sieci.

\* Jeśli pobieranie sieciowe się powiedzie, odpowiedź jest zapisywana do dedykowanej pamięci podręcznej (`api-data-cache-v1`) i wyświetlana użytkownikowi.

\* Jeśli pobieranie sieciowe zawiedzie (brak połączenia, błąd serwera), aplikacja próbuje załadować ostatnio zapisaną wersję z pamięci podręcznej.

\* Dodano przycisk "Pobierz Nowe Dane", aby użytkownik mógł manualnie wymusić próbę odświeżenia danych z sieci.

\* \*\*Czego się nauczyłem:\*\* Różne typy zasobów wymagają różnych strategii buforowania. Elastyczność Cache API pozwala na implementację niestandardowych rozwiązań dostosowanych do potrzeb. Ważne jest informowanie użytkownika o stanie danych (np. "załadowano z cache", "błąd sieci").

4. \*\*(Opcjonalnie) Problem: Debugowanie Service Workera w trybie offline.\*\*

\* \*\*Opis:\*\* Testowanie zachowania Service Workera, zwłaszcza strategii buforowania i fallbacków w trybie offline, może być trudne. Czasem trudno jest ustalić, dlaczego dany zasób jest (lub nie jest) serwowany z cache.

\* \*\*Rozwiązanie:\*\*

\* Intensywne korzystanie z zakładki "Application" w narzędziach deweloperskich:

\* "Service Workers": sprawdzanie statusu SW, wymuszanie aktualizacji, symulacja offline.

\* "Cache Storage": inspekcja zawartości pamięci podręcznych, manualne czyszczenie cache.

\* "Network": obserwowanie, które żądania są przechwytywane przez SW i skąd pochodzą odpowiedzi (z sieci, z cache SW).

\* Dodawanie wielu logów (`console.log`) w kodzie Service Workera oraz w skryptach aplikacji, aby śledzić przepływ sterowania i stan zmiennych.

\* \*\*Czego się nauczyłem:\*\* Dobre narzędzia deweloperskie są nieocenione. Systematyczne logowanie i testowanie różnych scenariuszy (online, offline, wolne połączenie) jest kluczowe dla zapewnienia niezawodności PWA.

## 5. Podsumowanie

### 5.1. Co działa dobrze

\* Aplikacja spełnia wszystkie podstawowe wymagania PWA: jest instalowalna, działa offline, posiada manifest i Service Workera.

\* Podstawowe funkcje listy zadań (dodawanie, usuwanie, oznaczanie jako ukończone) działają poprawnie i są zintegrowane z IndexedDB.

\* Integracja z zewnętrznym API oraz buforowanie jego odpowiedzi działa zgodnie z założeniami, umożliwiając dostęp do danych w trybie offline.

\* Nawigacja między trzema widokami jest płynna.

\* Interfejs użytkownika jest prosty i responsywny.

### 5.2. Co wymaga poprawy

\* \*\*Obsługa błędów:\*\* Chociaż podstawowa obsługa błędów istnieje, można ją rozbudować, aby dostarczać użytkownikowi bardziej szczegółowych informacji i sugestii.

\* \*\*Interfejs użytkownika (UI/UX):\*\* Wygląd aplikacji jest bardzo podstawowy. Można by go ulepszyć, dodając bardziej zaawansowane style, animacje i lepsze wrażenia użytkownika.

\* \*\*Synchronizacja w tle:\*\* Obecnie dane są zapisywane tylko lokalnie. W bardziej zaawansowanej wersji można by rozważyć synchronizację danych z serwerem w tle, gdy aplikacja odzyska połączenie.

\* \*\*Optymalizacja wydajności:\*\* Dla większej liczby zadań, operacje na IndexedDB i renderowanie listy mogłyby zostać zoptymalizowane.

\* \*\*Testy jednostkowe i integracyjne:\*\* Brak formalnych testów. W większym projekcie byłyby one niezbędne.

### 5.3. Czego się nauczyłem

Realizacja tego projektu była cennym doświadczeniem w zakresie tworzenia Progresywnych Aplikacji Internetowych od podstaw, bez użycia frameworków. Kluczowe nabyte umiejętności i wiedza to:

\* Praktyczne zrozumienie i implementacja \*\*Service Workerów\*\*, ich cyklu życia, strategii buforowania oraz przechwytywania żądań.

\* Konfiguracja i wykorzystanie \*\*Pliku Manifestu Aplikacji Webowej\*\*.

\* Praca z \*\*IndexedDB\*\* do przechowywania danych po stronie klienta, w tym obsługa transakcji i operacji asynchronicznych.

\* Implementacja mechanizmów zapewniających \*\*działanie aplikacji w trybie offline\*\*.

\* Integracja z \*\*zewnętrznym API\*\* i zarządzanie odpowiedziami (w tym buforowanie).

\* Znaczenie \*\*HTTPS\*\* dla PWA.

\* Rozwiązywanie problemów specyficznych dla PWA, takich jak aktualizacje SW czy debugowanie w trybie offline.

\* Docenienie wartości \*\*czystego JavaScriptu\*\* i zrozumienie podstawowych mechanizmów przeglądarki, które są często abstrahowane przez frameworki.

## 6. Załączniki

\* \*\*Link do działającej aplikacji:\*\* [W tym miejscu umieść link do hostowanej aplikacji]

\* \*\*Link do repozytorium GitHub z kodem źródłowym:\*\* [W tym miejscu umieść link do repozytorium GitHub]

\* \*\*Zrzuty ekranu pokazujące aplikację w trybie offline i online:\*\*

\* [Miejsce na zrzut ekranu - tryb online]

\* [Miejsce na zrzut ekranu - tryb offline]

\* [Miejsce na zrzut ekranu - lista zadań]

\* [Miejsce na zrzut ekranu - formularz]

\* [Miejsce na zrzut ekranu - dane z API]

\* \*\*Screenshoty z Lighthouse (wynik PWA >90 punktów wymagany):\*\*

\* [Miejsce na zrzut ekranu z audytu Lighthouse - kategoria PWA]