

PWA - przyszłość web developmentu(?)

Mateusz Nejdrowski 18733

1. Czym jest PWA?

PWA (Progressive Web Application) to nowoczesna technologia tworzenia aplikacji, które są dostępne poprzez przeglądarkę internetową. Charakteryzują się one responsywnym designem i funkcjonalnościami podobnymi do tradycyjnych aplikacji mobilnych, takimi jak możliwość pracy w trybie offline, powiadomienia push. PWA można zainstalować na urządzeniach mobilnych jako ikony na ekranie głównym, co umożliwia szybki i łatwy dostęp do aplikacji bez konieczności pobierania ich z App Store lub Google Play. Dzięki wykorzystaniu technologii Service Worker, PWA oferują szybkie ładowanie i płynne działanie, niezależnie od jakości połączenia internetowego. To innowacyjne podejście łączy zalety stron internetowych i aplikacji mobilnych, tworząc wygodne i elastyczne rozwiązanie dla użytkowników.

2. Wymogi PWA

Aby strona internetowa mogła zostać uznana za PWA, musi spełniać określone wymagania:

1. Responsywność: Strona PWA powinna być responsywna, czyli dostosowywać się do różnych rozmiarów ekranów i urządzeń. Dzięki temu użytkownicy mogą korzystać z aplikacji na smartfonach, tabletach i komputerach, zapewniając optymalne doświadczenie użytkownika na każdym urządzeniu.
2. Wielopoziomowa obsługa: PWA powinna być w stanie obsłużyć różne poziomy połączenia internetowego, w tym słabe lub niestabilne połączenia. Dzięki temu użytkownicy mogą korzystać z aplikacji nawet w miejscach o słabym zasięgu lub w sytuacjach, gdy połączenie jest tymczasowo niedostępne.
3. Manifest aplikacji: PWA powinna posiadać plik manifestu, zwykle o nazwie "manifest.json", który zawiera metadane aplikacji, takie jak nazwa, ikona, opis i konfiguracja wyglądu. Ten manifest pozwala przeglądarce na dodanie aplikacji na ekran główny urządzenia, umożliwiając użytkownikom łatwy dostęp do strony.

4. Service Worker: PWA powinna wykorzystywać Service Worker, który to jest skryptem JavaScript, działającym w tle, niezależnie od strony internetowej. Service Worker umożliwia przechwytywanie i obsługę żądań sieciowych, co pozwala na działanie strony w trybie offline, buforowanie zasobów oraz obsługę powiadomień push.
5. Bezpieczne połączenie HTTPS: Strona PWA musi być dostępna poprzez protokół HTTPS, co zapewnia szyfrowane połączenie między przeglądarką a serwerem, zwiększa bezpieczeństwo przesyłanych danych oraz umożliwia zainstalowania ww. service worker'a. Pamiętajmy, że PWA działa również na localhost, co oznacza, że możemy testować naszą aplikację lokalnie na naszym komputerze przed jej udostępnieniem publicznie.

3. Jak stworzyć aplikację PWA?

Stworzenie nowej aplikacji PWA, bądź też dodanie tej technologii do już istniejącej aplikacji jest bardzo proste i zrobimy to razem!

Stworzyłem przykładową stronę z jednym przyciskiem, który po jego kliknięciu zmienia kolor tła witryny:

```
1  <html>
2    <head> </head>
3    <body>
4      <script>
5        var kolor = 0;
6        function klik() {
7          kolor = 1 - kolor;
8          document.body.style.backgroundColor = kolor == 0 ? "red" : "green";
9        }
10     </script>
11     <button id="przycisk" onclick="klik()">KLIK</button>
12   </body>
13 </html>
```

By teraz nasza strona stała się aplikacją PWA potrzebujemy dwóch plików - **manifest.json** oraz **service-worker.js**.

Manifest.json odpowiada za dostarczanie informacji o aplikacji PWA, takich jak jej nazwa, ikona, opis, wersja, autor i inne metadane. Jest to plik w formacie JSON, który określa wygląd i zachowanie aplikacji na urządzeniach użytkowników. Przygotowany przez nas plik musimy zdefiniować w sekcji **<head>** naszej aplikacji:

```
1 <html>
2   <head>
3     <link rel="manifest" href="./manifest.json" />
4   </head>
5   <body>
6     <script>
7       var kolor = 0;
8       function klik() {
9         kolor = 1 - kolor;
10        document.body.style.backgroundColor = kolor == 0 ? "red" : "green";
11      }
12    </script>
13    <button id="przycisk" onclick="klik()">KLIK</button>
14  </body>
15 </html>
```

<link rel="manifest" href="ścieżka/do/manifest.json">

Przeglądarka odczytuje jego zawartość i interpretuje dostarczone metadane aplikacji. Dzięki temu może zidentyfikować stronę jako aplikację PWA i podjąć odpowiednie działania, uwzględnienie preferencji użytkownika dotyczących wyglądu i zachowania aplikacji oraz inne dostosowania interfejsu użytkownika.

Przykładowy plik manifest.json:

```
{
  "theme_color": "#f69435",
  "background_color": "#f69435",
  "display": "standalone",
  "scope": "/",
  "start_url": "/",
  "name": "test",
  "short_name": "test",
  "description": "test",
  "icons": [
    {
      "src": "/icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "/icon-256x256.png",
      "sizes": "256x256",
      "type": "image/png"
    },
    {
      "src": "/icon-384x384.png",
      "sizes": "384x384",
      "type": "image/png"
    },
    {
      "src": "/icon-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}
```

Ten manifest.json ma ustawiony kolor motywu (**theme_color**) i kolor tła (**background_color**) na **#f69435**.

Właściwość **display** jest ustawiona na **standalone**, co oznacza, że aplikacja będzie otwierana w osobnym oknie.

Scope i **start_url** są ustawione na **/**, co oznacza, że aplikacja będzie działać w zakresie głównego katalogu i jej strona startowa to również **/**. Nazwa aplikacji (**name**) i krótka nazwa (**short_name**) są ustawione na "test". Opis aplikacji (**description**) również ma wartość "test".

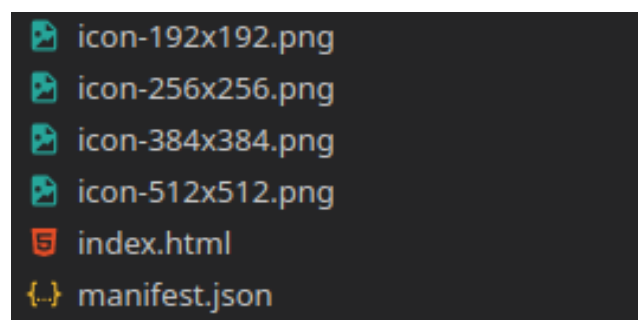
Istnieje również tablica ikon (**icons**) z różnymi rozmiarami i typami plików PNG.

Pamiętaj, że do projektu musisz dodać ikony w rozmiarach:

192x192, 256x256, 384x384, 512x512

Przeglądarka automatycznie wybierze właściwy rozmiar ikony w zależności od urządzenia, na którym zostanie zainstalowana aplikacja PWA. Dzięki temu ikony będą wyświetlane optymalnie i dostosowane do ekranu użytkownika.

Po dodaniu wybranych ikon oraz dodaniu pliku manifest.json nasz projekt powinien wyglądać tak:



O innych ustawieniach jakie możemy wpisać do naszego manifestu możemy poczytać na tej stronie:

<https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json>

Do stworzenia pliku manifest możemy również użyć takich witryn jak <https://app-manifest.firebaseapp.com/> lub też <https://www.simicart.com/manifest-generator.html/>.

Service Worker to skrypt JavaScript działający w tle przeglądarki, niezależnie od strony internetowej, która go zarejestrowała. Jego głównym zadaniem jest obsługa sieci i buforowanie zasobów, co umożliwia aplikacji PWA działanie w trybie offline. Service Worker może przechwytywać i manipulować żądaniami sieciowymi, co pozwala na szybsze ładowanie stron oraz dostęp do treści, nawet przy słabym lub braku połączenia internetowego. Ponadto, Service Worker może obsługiwać funkcje takie jak powiadomienia push, synchronizacja w tle oraz zarządzanie pamięcią podręczną, co sprawia, że jest kluczowym elementem w tworzeniu zaawansowanych aplikacji internetowych. Służy jako 'proxy' naszej strony internetowej - każdy request wykonywany przez aplikację przechodzi właśnie przez niego.



Stwórzmy zatem service worker'a dla naszej aplikacji!

Najpierw dodajmy plik '**service-worker.js**' w głównym katalogu naszej aplikacji:

```
icon-192x192.png
icon-256x256.png
icon-384x384.png
icon-512x512.png
index.html
manifest.json
service.worker.js
```

Teraz dodajmy dwie podstawowe funkcję service worker'a, czyli **install** i **fetch**:

```
1  self.addEventListener("install", (event) => {
2    event.waitUntil(
3      caches.open("my-cache").then((cache) => {
4        return cache.addAll(["/", "/index.html"]);
5      })
6    );
7  });
8
9  self.addEventListener("fetch", (event) => {
10   event.respondWith(
11     caches.match(event.request).then((response) => {
12       return response || fetch(event.request);
13     })
14   );
15 });
```

Pierwszy blok kodu rejestruje zdarzenie **"install"** i definiuje funkcję, która zostanie wywołana przy instalacji service workera.

W tym przypadku, funkcja ta otwiera nowe cache o nazwie "my-cache" za pomocą metody **caches.open()**. Następnie dodaje do tego cache pliki, które chcemy zapamiętać, używając metody **cache.addAll()**.

Na przykładzie naszej aplikacji, dodajemy dwa pliki - "/" (główny katalog) i "/index.html" (strona główna).

Funkcja **event.waitUntil()** zapewnia, że instalacja service workera nie zostanie uznana za zakończoną, dopóki nie zostaną wykonane wszystkie operacje wewnątrz niej.

Drugi blok kodu rejestruje zdarzenie **"fetch"** i definiuje funkcję, która zostanie wywołana przy każdym żądaniu zasobu (np. pliku HTML, CSS, obrazka) w przeglądarce.

Ta funkcja używa metody **caches.match()** do sprawdzenia, czy żądany zasób znajduje się już w pamięci podręcznej (**cache**).

Jeśli jest dostępny w cache, funkcja zwraca ten zasób jako odpowiedź (**response**). W przeciwnym razie, używa metody **fetch()**, aby pobrać żądany zasób z sieci.

W ten sposób, service worker sprawdza najpierw w cache, czy może dostarczyć zasób, zanim przejdzie do sieci.

By nasz service worker mógł rozpocząć swoją pracę, musimy go jeszcze zarejestrować na naszej stronie, co zrobimy w sekcji `<script>` naszego pliku `index.html`:

```
if ("serviceWorker" in navigator) {
  navigator.serviceWorker
    .register("/service-worker.js")
    .then((registration) => {
      console.log(
        "Service Worker registered with scope:",
        registration.scope
      );
    })
    .catch((error) => {
      console.error("Service Worker registration failed:", error);
    });
}
```

Ten kod sprawdza, czy przeglądarka obsługuje service workery przy użyciu warunku `if ("serviceWorker" in navigator)`. Jeśli tak, to wykonuje dalsze instrukcje.

navigator.serviceWorker: Jest to interfejs dostępny w obiekcie `navigator`, który zapewnia dostęp do funkcjonalności service workera.

.register("/service-worker.js"): Metoda **register()** interfejsu **navigator.serviceWorker** służy do zarejestrowania service workera. Przekazujemy do niej ścieżkę do pliku service workera, w tym przypadku `/service-worker.js`. Jeśli plik service workera znajduje się w innym miejscu, należy odpowiednio dostosować ścieżkę.

.then((registration) => { ... }): Ta część kodu to obsługa obietnicy (Promise), która jest zwracana przez metodę **register()**. Jeśli rejestracja service workera zostanie wykonana pomyślnie, wywoływana jest funkcja przekazana do **.then()**.

W tym przypadku, wyświetlamy informację w konsoli, że **"Service Worker został zarejestrowany z zakresem:"** oraz zakresem rejestracji (**registration.scope**).

.catch((error) => { ... }): Ta część kodu to obsługa błędów, jeśli rejestracja service workera nie powiedzie się. Jeśli wystąpi błąd podczas rejestracji, wywoływana jest funkcja przekazana do **.catch()**. W tym przypadku, wyświetlamy informację o błędzie rejestracji service workera w konsoli (**console.error()**).

Więcej informacji o service worker znajdziemy na stronie:

https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

Nasz projekt jest gotowy, by stać się pełnoprawną aplikacją PWA. Ostatnią rzeczą, którą musimy zrobić jest udostępnienie naszego projektu na serwerze - jest to konieczne, gdyż tak jak wspominałem o wymogach PWA - **strona musi być serwowana przez połączenie HTTPS, bądź też localhost**. Rejestracja naszego service worker'a nie powiedzie się w przypadku włączenia pojedynczego pliku html w przeglądarce.

Do przetestowania działania naszej aplikacji jako PWA możemy w naszym projekcie zainstalować bibliotekę 'http-server' - do tego potrzebny jest nam także środowisko **Node.js**, które zawiera menedżer pakietów npm (node package manager), który jest potrzebny do instalacji bibliotek. Na stronie <https://nodejs.dev/en/learn/how-to-install-nodejs/> możemy przeczytać jak zainstalować ww środowisko.

Po instalacji **Node.js** (lub też jeśli już posiadamy zainstalowane środowisko na swoim urządzeniu) wystarczy otworzyć terminal w katalogu naszego projektu, w który wpisujemy komendę:

npm install -g http-server

Spowoduje to globalną instalację biblioteki `http-server` na Twoim komputerze oraz doda do naszego projektu plik **package.json** oraz **package-lock.json**. Więcej o tych plikach możemy przeczytać na stronie <https://docs.npmjs.com/cli/v9/configuring-npm/package-json> .

Ostatnim krokiem jest uruchomienie serwera `http` komendą:

http-server

Teraz możemy przetestować naszą aplikację PWA lokalnie na jednym z wskazanych przez bibliotekę 'http-server' adresów.

By zainstalować aplikację na telefonie, możemy skorzystać z zainstalowanej wcześniej biblioteki. Do uruchomienia aplikacji na smartfonie konieczne jest połączenie obu urządzeń(urządzenie z aplikacją oraz telefon) w tej samej sieci WI-FI.

Po uruchomieniu lokalnego serwera, sprawdź adres IP urządzenia - wpisz w terminal komendę **ipconfig**. Następnie w przeglądarce telefonu wpisz sprawdzony adres i dopisz port na jakim uruchomiona jest aplikacja, np. 8080(domyślny port `http-server`). To jeden ze sposobów, możesz także przesłać aplikację na faktyczny serwer.

4. Korzyści PWA dla użytkowników

Jedną z kluczowych cech PWA jest możliwość dostępu offline. Dzięki mechanizmom buforowania treści, użytkownicy mogą korzystać z aplikacji i przeglądać treści, nawet gdy nie mają połączenia z internetem. PWA automatycznie przechowują zasoby w pamięci podręcznej, umożliwiając użytkownikom kontynuowanie korzystania z aplikacji bez przerw, co jest szczególnie korzystne w podróży, obszarach o słabym zasięgu lub przy ograniczonym dostępie do sieci.

Podczas pierwszego uruchomienia aplikacji PWA, Service Worker jest instalowany i rejestrowany przez przeglądarkę. Następnie, podczas kolejnych odwiedzin aplikacji, Service Worker jest aktywowany i może przechwytywać żądania sieciowe. Gdy użytkownik próbuje uzyskać dostęp do zasobu, Service Worker najpierw sprawdza, czy zasób jest

dostępny w pamięci podręcznej. Jeśli tak, zasób jest zwracany bez potrzeby wysyłania żądania do serwera. Dzięki temu, nawet w przypadku braku połączenia z internetem, aplikacja może dostarczać wcześniej pobrane zasoby i umożliwiać użytkownikom przeglądanie treści.

W przypadku braku zasobu w pamięci podręcznej, Service Worker wysyła żądanie do serwera w celu pobrania brakującego zasobu. Po pobraniu zasobu, Service Worker zapisuje go w pamięci podręcznej, co umożliwia dostęp offline w przyszłości. Ponadto, aplikacje PWA mogą wykorzystywać różne strategie buforowania, takie jak buforowanie podstawowe (**cache-first**) lub buforowanie sieciowe (**network-first**), aby zoptymalizować wydajność i dostępność zasobów w trybie offline.

PWA wyróżniają się również szybkością i responsywnością.

Zastosowanie zaawansowanych optymalizacji, takich jak buforowanie, minimalizacja i kompresja zasobów, oraz odpowiednie wykorzystanie mechanizmów przeglądarki, pozwala na szybkie ładowanie stron i płynne, interaktywne doświadczenie użytkownika. W rezultacie korzystanie z aplikacji PWA jest przyjemne i efektywne.

Aplikacje PWA wykorzystują również różne techniki, takie jak **lazy loading** (opóźnione ładowanie), aby przyspieszyć wczytywanie stron. Lazy loading polega na ładowaniu zasobów dopiero w momencie, gdy są one potrzebne, co umożliwia szybsze ładowanie strony początkowej.

Dzięki funkcji instalacji na ekranie głównym, użytkownicy mają możliwość zainstalowania strony PWA jako ikony na swoim urządzeniu. Ta funkcjonalność tworzy wrażenie tradycyjnej aplikacji mobilnej, zapewniając łatwy i szybki dostęp do aplikacji bez konieczności otwierania przeglądarki i wpisywania adresu URL.

Powiadomienia push w kontekście Progressive Web Apps (PWA) stanowią istotny aspekt, który umożliwia aplikacjom PWA dostarczanie użytkownikom istotnych informacji, aktualizacji i powiadomień o nowych treściach. Dzięki powiadomieniom push, aplikacje PWA mogą tworzyć spersonalizowane doświadczenia dla użytkowników, zwiększając zaangażowanie i retencję.

Powiadomienia te działają niezależnie od tego, czy aplikacja jest otwarta na ekranie użytkownika, umożliwiając natychmiastowe dostarczanie istotnych treści i powiadomień.

Dzięki automatycznym aktualizacjom, strony PWA zawsze działają na najnowszej wersji, bez konieczności ręcznej instalacji aktualizacji. To eliminuje trudności związane z zarządzaniem aktualizacjami i zapewnia użytkownikom dostęp do najnowszych funkcji, poprawek i zabezpieczeń.

5. Korzyści PWA dla deweloperów

Aplikacje PWA zapewniają deweloperom jedną platformę, na której mogą tworzyć aplikacje dla różnych urządzeń i systemów operacyjnych. To oznacza mniejsze nakłady na rozwój i utrzymanie, ponieważ nie trzeba tworzyć oddzielnych aplikacji dla każdej platformy.

Dzięki możliwości aktualizacji bezpośrednio na serwerze, deweloperzy mogą szybko wprowadzać zmiany, poprawiać błędy i dostarczać nowe funkcje użytkownikom. Aktualizacje są automatycznie stosowane przy kolejnym uruchomieniu aplikacji przez użytkownika, eliminując potrzebę pobierania i instalacji aktualizacji.

Aplikacje PWA wymagają korzystania z protokołu HTTPS, co zapewnia bezpieczne połączenie między użytkownikiem a aplikacją. Deweloperzy mają kontrolę nad aktualizacjami i mogą szybko reagować na nowe zagrożenia, dostarczając łatki bezpieczeństwa i aktualizacje, co przyczynia się do zapewnienia bezpieczeństwa i ochrony danych użytkowników.

6. Sukcesy i zastosowania PWA

Strony PWA odniosły wiele sukcesów i przyczyniły się do poprawy doświadczenia użytkowników oraz wyników biznesowych. Oto kilka przykładów sukcesów, jakie osiągnięto dzięki aplikacjom PWA:

1. Twitter stworzył wersję PWA swojej platformy o nazwie Twitter Lite, która zajmuje mniej miejsca, zużywa mniej danych i działa szybciej niż tradycyjna aplikacja mobilna. Dzięki temu udało się dotrzeć do użytkowników o słabym połączeniu internetowym i zwiększyć zaangażowanie. Twitter Lite zanotował 75% zwiększenie liczby tweetów, a czas retencji użytkowników wzrósł o 65%.
2. Platforma społecznościowa Pinterest zastosowała PWA, aby zwiększyć zaangażowanie użytkowników i zminimalizować czas ładowania strony. Dzięki temu liczba zapisanych elementów wzrosła o 60%, a liczba nowych użytkowników o 44%. PWA pomogło również zmniejszyć zużycie danych mobilnych, co było istotne dla użytkowników o ograniczonym planie danych.
3. Spotify, popularna platforma do strumieniowania muzyki, wprowadziła PWA, aby umożliwić użytkownikom dostęp do swojej biblioteki muzycznej bez konieczności pobierania aplikacji, dzięki czemu liczba dziennych, aktywnych użytkowników urosła dwukrotnie.
4. Strona BMW po wdrożeniu PWA zyskała 50% więcej użytkowników mobilnych oraz cztery razy więcej użytkowników odwiedzało podstronę sprzedaży.

7. Wady PWA

Aplikacje PWA mają ograniczony dostęp do funkcji urządzenia w porównaniu do tradycyjnych aplikacji mobilnych. Nie mają pełnej integracji z systemowymi funkcjami, takimi jak dostęp do czujników urządzenia (np. GPS, akcelerometr), systemowych powiadomień czy płatności. Oznacza to, że niektóre zaawansowane funkcje mogą być ograniczone w przypadku aplikacji PWA.

Mimo coraz większego wsparcia przeglądarek dla PWA, starsze przeglądarki mogą nie oferować pełnego wsparcia dla funkcji PWA. To oznacza, że niektórzy użytkownicy korzystający z przeglądarek niewspierających PWA mogą mieć ograniczony dostęp do pełnych możliwości aplikacji. Wymaga to od deweloperów uwzględnienia kompatybilności i zastosowania alternatywnych rozwiązań dla starszych przeglądarek.

Warto również wspomnieć, że technologia PWA jest nadal stosunkowo nowa i rozwija się dynamicznie. Niektóre z wad i ograniczeń mogą zostać rozwiązane w przyszłości, a funkcjonalności PWA mogą być stale udoskonalane i rozbudowywane.

8. Źródła

<https://greenparrot.pl/blog/plusy-i-minusy-progressive-web-apps-w-punktach-i-case-study-sukcesow-pwa/>

<https://trojanczyk.pl/wszystko-co-warto-wiedziec-na-temat-pwa/>

<https://auroracreation.pl/blog/co-to-pwa-i-jak-dziala/>

<https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json>

[https://developer.mozilla.org/en-US/docs/Web/API/Service Worker API](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API)