Лабораторна робота

Тема: Основи створення прогресивних веб-застосунків

Мета: Ознайомитися з підходами та технологіями створення прогресивних веб-застосунків. Спробувати створення елементарного веб-застосунку

Методичні рекомендації до виконання

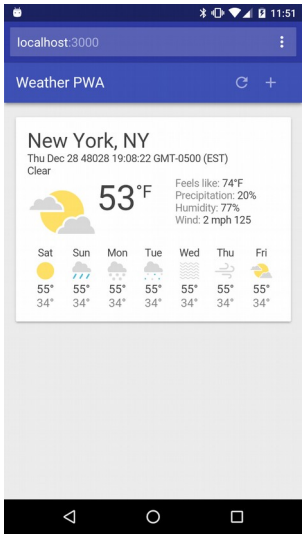
1 Вступ

Прогресивні веб-застосунки поєднують в собі найкраще зі світу веб та світу традиційних прикладних програм. Вони корисні для користувачів від найпершого відкриття сторінки в браузері, не вимагають установки. У міру того, як користувач поступово заглиблюється у роботу із застосунком, він стає все більш і більш корисним. Він швидко завантажується, навіть в нестабільних мережах, відправляє релевантні пуш-повідомлення, має іконку на домашньому екрані і дарує повноцінний повноекранний досвід.

Що ми будемо робити?

Ця лабораторна робота проведе вас по всьому процесу створення Progressive Web App, включаючи дизайн, і пояснить всі подробиці для того, щоб ви могли зробити застосунок, що відповідає всім ключовим принципам Progressive Web. Давайте розглянемо властивості Progressive Web App:

- **Прогресивне** - ми будемо використовувати прогресивне поліпшення в ході процесу.
- Адаптивне- ми переконаємося, що воно працює на будь-якому форм факторі.
- **Незалежне від з'єднання** - закешуємо оболонку за допомогою service worker-а.
- **Схоже на рідний застосунок**- будемо використовувати взаємодії як в застосунках для додавання міст і поновлення даних.
- **Нове** - будемо кешувати нові дані за допомогою service worker-а.
- **Безпечне** - розмістимо додаток на сервері з підтримкою HTTPS.
- **Індексоване та встановлюється** - додамо маніфест для простої індексації пошуковими системами.
- **З посиланням - це веб!**



Що вам знадобиться

- Chrome 70 або вище
- Встановлений Web Server for Chrome або використання власного веб сервера на ваш вибір.
- Тестовий код
- Текстовий редактор
- Пройдені попередні роботи з розумінням HTML, CSS і JavaScript

2 Налаштування

Ви можете завантажити весь код для цієї лабораторної з електронного курсу.

Розпакуйте викачаний zip-файл. Ви отримаєте кореневу теку (`your-first-pwapp-master`) і в ній є ще по одній директорії для кожного розділу цієї лабораторної, а також всі ресурси, які вам знадобляться.

Теки `step-NN` містять потрібний кінцевий стан для кожного кроку. Вони наведені в якості довідки. Всю роботу ми будемо робити в теці під назвою `work`.

Для роботи з простими веб-застосунками, які базуються лише на HTML, CSS та JavaScript можете встановити Спеціальне розширення браузера Web Server for Chrome. Самостійно проведіть його налаштування.

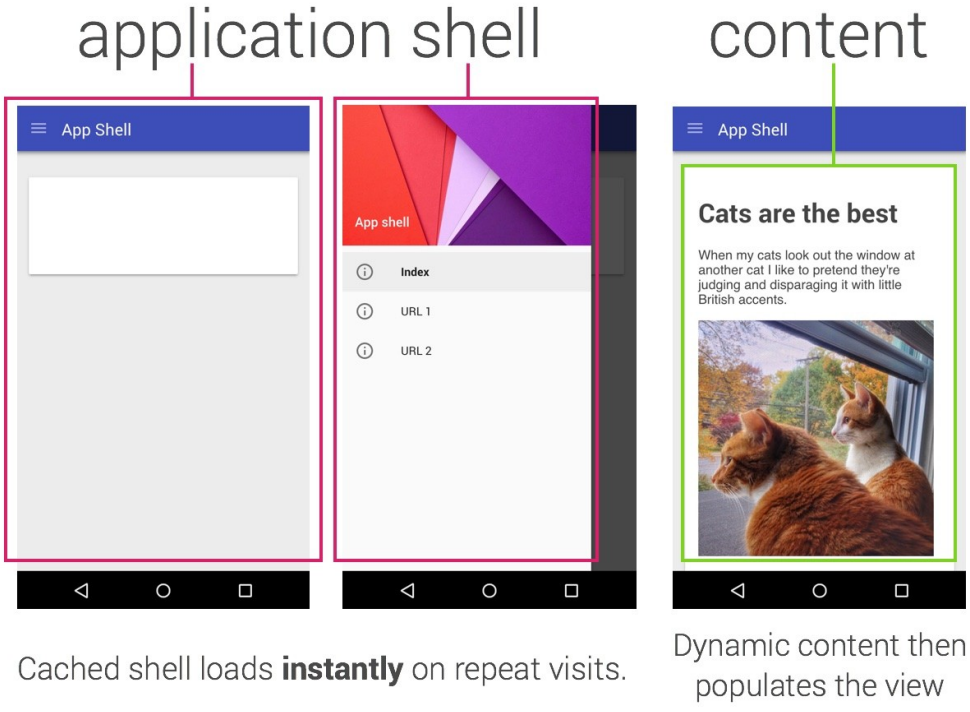
Або використовуйте сервер, на якому робили попередні лабораторні роботи.

3 Архітектура вашої оболонки у застосунку (App Shell)

Що таке app shell?

Оболонка програми - мінімальний набір HTML, CSS і JavaScript, який потрібно для роботи призначеного для користувача інтерфейсу прогресивного веб-застосунку і є одним з компонентів, що забезпечують хорошу продуктивність. Його перше завантаження повинно бути надзвичайно швидким і негайно кешуватися. "Кешуватися" означає те, що файли оболонки завантажуються один раз по мережі і потім зберігаються на локальному пристрої. Кожен наступний раз, коли користувач відкриває застосунок, файли оболонки завантажуються з кешу локального пристрою, що призводить до блискавичного запуску.

В архітектурі оболонки застосунку ядро інфраструктури і UI відокремлені від даних. Все, що стосується UI і інфраструктури, кешується локально за допомогою service worker-а, так що при наступних запусках Progressive Web App потрібно отримати тільки необхідні дані, а не завантажувати все знову.



Іншими словами, оболонка застосунка схожа на той код, що ви публікуєте в магазині застосунків, створюючи нативний застосунок. Це основні компоненти, необхідні для того, щоб зрушити ваш застосунок з мертвої точки, але, ймовірно, в них немає ніяких даних.

Чому треба використовувати архітектуру App Shell?

Використання оболонки програми дозволяє вам зосередитися на швидкості робить ваш прогресивний веб-застосунок схожим на нативний: він миттєво завантажується і регулярно оновлюється, і все це без звернення до магазину застосунків.

Створюємо App Shell

Перший крок - розкласти конструкцію на основні елементи.

Запитайте себе:

- Що відразу повинно бути на екрані?
- Які UI компоненти є ключовими для вашого застосування?
- Які ресурси необхідні для роботи оболонки? Наприклад, картинки, JavaScript, стилі і т.д.

Ми створимо Погодний застосунок в якості нашого першого Progressive Web App. Його ключові компоненти це:

- Тема з назвою і кнопками додавання та оновлення
- Контейнер з карткою прогнозу
- Шаблон картки прогнозу
- Вікно діалогу при додаванні міста
- Індикатор завантаження

При розробці більш складних додатків, контент, який не є необхідним при початковому завантаженні, може бути запитаний пізніше і потім закешований для подальшого використання. Наприклад, ми можемо відкласти завантаження Нью-Йорка на потім, коли буде простій, а спочатку відобразити перший запуск.

4 Реалізація вашого App Shell

Створюємо HTML для App Shell

Тепер ми додамо всі основні компоненти, які ми обговорювали в попередньому розділі.

Ваш файл `index.html` в директорії `work`повинен бути схожий на це (це лише частина всього контенту, не копіюйте весь код в ваш файл):

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Weather PWA</title>
```

```
    <link rel="stylesheet" type="text/css" href="styles/inline.css">
</head>
<body>
    <header class="header">
        <h1 class="header__title">Weather PWA</h1>
        <button id="butRefresh" class="headerButton"></button>
        <button id="butAdd" class="headerButton"></button>
    </header>

    <main class="main">
        <div class="card cardTemplate weather-forecast" hidden>
        . . .
        </div>
    </main>

    <div class="dialog-container">
    . . .
    </div>

    <div class="loader">
        <svg viewBox="0 0 32 32" width="32" height="32">
            <circle id="spinner" cx="16" cy="16" r="14" fill="none"></circle>
        </svg>
    </div>

    <!-- Insert link to app.js here -->
</body>
</html>
```

Зверніть увагу, що індикатор завантаження видно за замовчуванням. Це зроблено для того, щоб користувач побачив індикатор відразу при завантаженні сторінки - це дасть йому зрозуміти, що контент завантажується.

Щоб заощадити час, ви вже маєте таблицю стилів у каталозі work/styles/. Витратьте пару хвилин на її вивчення і налаштуйте на свій смак.

Додаємо ключовий початковий код JavaScript

Тепер, коли ми підготували велику частину UI, час звернутися до коду для того, щоб все закрутилося. Як і в усьому іншому, що стосується оболонки pfcnjceyre, вам треба вирішити, який код є ключовим для роботи, а який можна завантажити пізніше.

У початковий код (`app.js`), ми включили:

•Об'єкт `app`, який несе ключову інформацію, необхідну для роботи програми.

•Споглядач подій для всіх кнопок в заголовку (`add/refresh`) і діалогу додавання міста (`add/cancel`).

•Метод для додавання або оновлення карток прогнозів (`app.updateForecastCard`).

•Метод для отримання останніх прогнозів з Firebase Public Weather API (`app.getForecast`).

•Метод для ітерації поточних карток і виклику `app.getForecast` для отримання даних прогнозів (`app.updateForecasts`).

•Деякі фейковий дані (`fakeForecast`) для того, щоб можна було швидко протестувати рендеринг.

Додаємо JavaScript код

1.Створіть теку `scripts у` вашому каталозі `work`.

2.Скопіюйте `app.js` з директорії `resources/step4 у` вашу теку `scripts`.

3.У `index.html` додайте посилання на щойно створений `app.js,` замінивши `<!-- Insert link to app.js here -->` на таке:

```
<script src = "scripts / app.js" async> </ script>
```

Протестуємо

Тепер, коли у вас є основа у вигляді HTML, стилів і JavaScript, саме час протестувати застосунок. Хоча він поки багато не вміє, треба переконатися, що в ньому немає помилок.

Щоб побачити, як відображаються фейковий дані, приберіть коментар в цьому рядку у файлі `scripts/app.js`:

```
// app.updateForecastCard (fakeForecast);
```

В результаті ви повинні отримати гарну картку з прогнозом (хоч і неправильним). Щось типу такого:

Після того, як ви спробували застосунок і переконалися, що він працює, видаліть `fakeForecasti` звернення до `app.updateForecastCard(fakeForecast)`. Все це потрібно нам було тільки для тестування. В наступному кроці ми почнемо використовувати реальні дані.

5 Починаємо зі швидкого першого завантаження

Прогресивний веб-застосунок має запускатися швидко і бути готовим для використання негайно. У своєму поточному стані наш погодний застосунок стартує швидко, але він не працездатний. У ньому немає даних. Ми можемо зробити AJAX запит, щоб отримати ці дані, але це додатковий запит, а значить більш тривале завантаження. Замість цього давайте надамо реальні дані в перший же запуск.

Вставляємо дані погодного прогнозу

У цій лабораторній роботі ми просто вставимо прогноз погоди прямо в JavaScript, але в реальному застосунку останній прогноз сервера треба вставляти, ґрунтуючись на геолокації користувача, яку можна зробити за IP-адресою.

Додайте наступний код JavaScript прямо всередину функції, яка викликається відразу в `scripts/app.js`(після `'use strict';` на початку):

```
var initialWeatherForecast = {
      key: 'newyork',
      label: 'New York, NY',
      currently: {
        time: 1453489481,
        summary: 'Clear',
        icon: 'partly-cloudy-day',
        temperature: 52.74,
        apparentTemperature: 74.34,
        precipProbability: 0.20,
        humidity: 0.77,
```

```
      windBearing: 125,
      windSpeed: 1.52
    },
    daily: {
      data: [
        {icon: 'clear-day', temperatureMax: 55, temperatureMin: 34},
        {icon: 'rain', temperatureMax: 55, temperatureMin: 34},
        {icon: 'snow', temperatureMax: 55, temperatureMin: 34},
        {icon: 'sleet', temperatureMax: 55, temperatureMin: 34},
        {icon: 'fog', temperatureMax: 55, temperatureMin: 34},
        {icon: 'wind', temperatureMax: 55, temperatureMin: 34},
        {icon: 'partly-cloudy-day', temperatureMax: 55, temperatureMin: 34}
      ]
    }
};
```

Розрізняємо перший запуск

Але як ми бачимо, що вже не потрібно показувати цю інформацію (яка може бути вже застарілою) під час запуску програми з кешу? Коли користувач запускає застосунок наступного разу, можливо, він вже змінив місто, тому нам треба завантажити інформацію вже для цього міста, а зовсім не для того, з якого він починав.

Призначені для користувача налаштування, на зразок міст, для яких людина отримує прогнози, повинні зберігатися локально, в IndexedDB, або за допомогою іншого швидкого механізму запису. Щоб спростити цю лабораторну, ми використаємо `localStorage`, що не ідеально для готового застосунку, так як це блокований, синхронний механізм, який, потенційно, може працювати дуже повільно на деяких пристроях.

По-перше, давайте додамо код, який потрібно для збереження налаштувань користувача в кінець негайно викликається функції в `scripts/app.js`(до рядка `})();` в кінці файлу):

```
// Save list of cities to localStorage.
app.saveSelectedCities = function() {
  var selectedCities = JSON.stringify(app.selectedCities);
  localStorage.selectedCities = selectedCities;
};
```

Далі, давайте додамо код перевірки того, чи є у користувача збережені міста і показ їх, якщо є. В іншому випадку будемо виводити прописані на початку дані. Додайте цей код в ваш `scripts/app.js` (після того коду, який ви тільки що додали):

```
/*****************************************************************************
```

```
  *
  * Code required to start the app
  *
  * NOTE: To simplify this codelab, we've used localStorage.
  *   localStorage is a synchronous API and has serious performance
  *   implications. It should not be used in production applications!
  *   Instead, check out IDB (https://www.npmjs.com/package/idb) or
  *   SimpleDB (https://gist.github.com/inexorabletash/c8069c042b734519680c)
  ************************************************************************/

app.selectedCities = localStorage.selectedCities;
if (app.selectedCities) {
    app.selectedCities = JSON.parse(app.selectedCities);
    app.selectedCities.forEach(function(city) {
        app.getForecast(city.key, city.label);
    });
} else {
    app.updateForecastCard(initialWeatherForecast);
    app.selectedCities = [
        {key: initialWeatherForecast.key, label: initialWeatherForecast.label}
    ];
    app.saveSelectedCities();
}
```

Зберігаємо вибрані міста

Нарешті, не забудемо зберегти список міст тоді, коли користувач додає нове. Додамо: `app.saveSelectedCities();` в обробник натискання кнопки `butAddCity` (прямо перед `app.toggleAddDialog(false);`).

Тестуємо

- При першому запуску ваше застосування повинне негайно показати прогноз з `initialWeatherForecast`.
- Додайте нове місто (клацніть на іконку з + вгорі праворуч) і переконайтеся, що тепер показуються дві картки.
- Оновіть браузер і перевірте, що програма завантажує два прогнози з останньою інформацією.

6 Використовуємо service worker для попереднього кешування App Shell

Прогресивні веб-застосунки повинні бути швидкими і встановлюються, що означає їх роботу в онлайні, офлайні і не уриватися при повільних з'єднаннях. Щоб домогтися цього, нам потрібно кешувати оболонку застосунку за допомогою service worker-а, щоб вона була завжди доступна - швидко і надійно.

Якщо ви не знайомі з Service worker-ами, то початкове розуміння ви можете отримати зі статті Введення в Service Worker-и - там викладено, що вони можуть робити, як працює їхній життєвий цикл і так далі.

Порада: Використовуйте нове вікно в режимі інкогніто для тестування і налагодження service worker-ів. Коли інкогніто вікно закривається, Chrome видаляє всі кешовані дані і встановлені service worker-и, даючи вам можливість завжди почати з чистого аркуша.

Функції, що надаються Service worker-ами, слід розглядати як прогресивне поліпшення і додавати тільки якщо вони підтримуються браузером. Наприклад, за допомогою service worker ви можете кешувати оболонку програми та дані вашого застосування, так щоб воно було доступне навіть тоді, коли мережі немає. Якщо service worker не підтримується, то і викликів до офлайнового коду немає, а значить, користувач отримує тільки простий функціонал. Використання функції визначення для надання прогресивного поліпшення не доставляє будь-яких складнощів, а програма не "зламається" в старих браузерах, які не підтримують цей функціонал.

Пам'ятайте: Функції Service worker-а працюють тільки для сторінок, доступних через HTTPS (https: // localhost і аналоги також працюють для цілей тестування). Щоб дізнатися більше про причини такого обмеження, прочитайте статтю Віддаємо перевагу безпечній основі для потужних нових функцій від команди Chromium.

Реєструємо service worker якщо він доступний

Перший крок до того, щоб наш застосунок запрацював в офлайні - зареєструвати service worker, скрипт, який буде забезпечувати фонову роботу без відкритої веб-сторінки або взаємодії з користувачем.

Це робиться за два простих кроки:

1. Треба попросити браузер зареєструвати файл з JavaScript як service worker.
2. Створити файл з JavaScript, що містить service worker

По-перше, нам треба перевірити, чи підтримує браузер роботу service worker-ів, а якщо підтримує, зареєструвати service worker. Додайте наступний код в `scripts/app.js`file (до `})();` в кінці):

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker
           .register('./service-worker.js')
           .then(function() { console.log('Service Worker Registered'); });
}
```

Кешуємо ресурси сайту

Коли service worker зареєстрований, подія установки спрацьовує, коли користувач вперше відвідує сторінку. У обробнику цієї події ми закешуємо всі ресурси, які необхідні застосунку.

Коли service worker запускається, він повинен відкрити об'єкт `caches` і заповнити його ресурсами, необхідними для завантаження App Shell. Створіть файл `service-worker.js` в кореневій теці вашого проекту (це `your-first-pwapp-master/work`). Цей файл повинен знаходитися в корені застосунку, так як область діяльності service worker-а визначається тією директорією, в якій розташовується файл. Додайте цей код в ваш новий файл `service-worker.js`:

```
var cacheName = 'weatherPWA-step-6-1';
var filesToCache = [];

self.addEventListener('install', function(e) {
  console.log('[ServiceWorker] Install');
  e.waitUntil(
    caches.open(cacheName).then(function(cache) {
      console.log('[ServiceWorker] Caching app shell');
      return cache.addAll(filesToCache);
    })
  );
});
```

Перш за все, нам треба відкрити кеш за допомогою `caches.open()`і визначити ім'я кеша. Визначення імені кешу дозволить нам працювати з версіями файлів або відокремити дані від оболонки, так що ми зможемо легко оновлювати одне і не чіпати інше.

Відкривши кеш, ми можемо звернутися до `cache.addAll()`, який отримає список URL, потім запросить їх у сервера і додати отриману відповідь в кеш. На жаль, `cache.addAll()` атомарний, тобто, якщо будь-який файл видасть помилку, то весь кеш нічого не отримає!

Не забудьте змінювати змінну кожного разу, коли ви міняєте ваш service worker . Так ви завжди будете отримувати найостанніші версії файлів з кешу. Важливо також періодично чистити кеш від невикористаного контенту і даних. Додайте до подій слухача, який отримує все ключі кешу і видаляє непотрібні. Розмістіть цей код в кінець: **`cacheName`** `activateservice-worker.js`

```
self.addEventListener('activate', function(e) {
  console.log('[ServiceWorker] Activate');
  e.waitUntil(
    caches.keys().then(function(keyList) {
      return Promise.all(keyList.map(function(key) {
        if (key !== cacheName) {
          console.log('[ServiceWorker] Removing old cache', key);
```

```
                return caches.delete(key);
            }
        }));
    })
  );
});
```

Нарешті, давайте оновимо список файлів, які необхідні для оболонки програми. В цей масив нам треба включити всі файли, які потрібні нашому застосунку, включно із зображеннями, JavaScript, файлами стилів і т.д. На початку вашого `service-worker.js` замініть `var filesToCache = [];` на такий код:

```
var filesToCache = [
  '/',
  '/index.html',
  '/scripts/app.js',
  '/styles/inline.css',
  '/images/clear.png',
  '/images/cloudy-scattered-showers.png',
  '/images/cloudy.png',
  '/images/fog.png',
  '/images/ic_add_white_24px.svg',
  '/images/ic_refresh_white_24px.svg',
  '/images/partly-cloudy.png',
  '/images/rain.png',
  '/images/scattered-showers.png',
  '/images/sleet.png',
  '/images/snow.png',
  '/images/thunderstorm.png',
  '/images/wind.png'
];
```

Не забудьте включити всі можливі імена файлів. Наприклад, наш застосунок починається з `index.html`, але він також може бути запрошеним як `/`, так як сервер віддає `index.html`, коли запитується просто коренева директорія. Ви можете опрацювати це в методі `fetch`, але так це вимагало б спеціального перетворення регістра, що може стати складним.

Наш застосунок ще не до кінця працює в офлайні. Ми кешуємо компоненти оболонки, але нам все ще треба завантажувати їх з локального кешу.

Запускаємо app shell з кешу

Service worker дає можливість перехоплювати запити, зроблені з нашого Progressive Web App та обробляти їх усередині service worker-а. Це означає, що ми можемо визначити те, як ми хочемо обробляти запити і, потенційно, вручити користувачеві нашу кешовану відповідь.

наприклад:

```
self.addEventListener('fetch', function(event) {
  // Do something interesting with the fetch here
});
```

Давайте тепер завантажимо нашу оболонку з кешу. Додайте наступний код в кінець файлу `service-worker.js`:

```
self.addEventListener('fetch', function(e) {
  console.log('[ServiceWorker] Fetch', e.request.url);
  e.respondWith(
    caches.match(e.request).then(function(response) {
      return response || fetch(e.request);
    })
  );
});
```

Якщо дивитися зсередини, `caches.match()` оцінює веб запит, який ініціював подію `fetch`, і перевіряє його доступність в кеші. Він відповідає або кешованою версією, або використовує `fetch` для отримання копії з мережі. Метод `response` передає відповідь назад веб-сторінці за допомогою `e.respondWith()`.

Остерігайтеся граничних випадків

Як ми раніше говорили, цей код **не треба використовувати в робочих застосунках, оскільки** в ньому не оброблено безліч граничних випадків.

Кеш залежить від поновлення ключа кешу для кожної зміни

Наприклад, цей метод кешування вимагає від вас оновлювати ключ кешу кожного разу, коли змінюється контент, інакше кеш НЕ буде оновлений і буде показуватися старий контент. Так що вам обов'язково треба міняти ключ кешу з кожною зміною в ході роботи над проектом!

Вимога завантаження всього для кожного зміни

Зворотний бік - весь кеш стає нечинним та його треба завантажувати заново кожен раз, коли змінюється навіть один файл. Це означає, що виправлення одного символу в коді робить кеш неробочим і його треба завантажувати знову. Не дуже ефективно.

Кеш браузера може заважати service worker-у оновлювати кеш

Ось ще один важливий нюанс. Дуже важливо, щоб HTTPS запит в ході установки обробника йшов прямо до мережі і не повертав відповідь від кешу браузера. В іншому випадку браузер

може повернути стару, кешовану версію, що призведе до того, що кеш service worker-а насправді ніколи не оновиться!

Остерігайтеся cache-first стратегії в продакшині

Наш застосунок використовує cache-first стратегію - це значить, що копія будь-якого кешованого об'єкта повертається без звернення до мережі. Хоча таку стратегію просто реалізувати, вона може викликати проблеми в майбутньому. Коли копію хост сторінки і реєстрацію service worker-а закешовано, може бути надзвичайно складно змінити конфігурацію service worker-а (оскільки конфігурація залежить від того, де вона була визначена) і ви можете зіткнутися з тим, що розгорнуті сайти буде дуже важко відновити.

Як уникнути цих граничних випадків?

Так як боротися з цими проблемами? Використовуйте бібліотеку, типу sw-precache , вона дає точний контроль над терміном життя об'єктів, відправляючи запити прямо в мережу, і займається рештою важкої роботи для вас.

Поради по тестуванню працюють service worker-ів

Налагодження service worker-ів може бути складним завданням, а коли тут задіяний ще і кеш, все може перетворитися на справжній кошмар. Одна помилка - і все піде прахом. Але рано панікувати. Є інструменти, які можуть спростити ваше життя.

Деякі поради:

- Навіть якщо service worker зареєстровано, він може продовжувати працювати до тих пір, поки його вікно браузера не буде закрито.
- Chrome іноді показує помилку в консолі при спробі звернутися до service worker-у, її можна ігнорувати.
- Якщо відкрито кілька вікон з вашим додатком, новий service worker не запрацює до тих пір, поки всі вони не будуть перезавантажені і оновлені до останньої версії.
- Розреєстрація service worker-а не очищає кеш, так що імовірна ситуація коли ви будете отримувати старі дані до тих пір, поки ім'я кешу не зміниться.
- Якщо service worker існує і реєструється новий, новий не отримає управління до тих пір, поки сторінка не буде перезавантажена, якщо тільки ви не перехопили його .

Ваш новий кращий друг: chrome://serviceworker-internals

Внутрішня сторінка Chrome Service Worker Internals (`chrome://serviceworker-internals`) це справжній рятівник, вона дозволяє вам просто зупиняти і розреєстровувати існуючі service worker-и і запускати нові. Ви також можете використовувати цю сторінку для запуску Developer Tools для service worker-а, що дасть вам доступ в його консоль.

Протестируем

•Відкрийте Chrome DevTools або відкрийте нову вкладку і перевірте `chrome://serviceworker-internals` для того, щоб переконатися, що service worker правильно зареєстрований і потрібні ресурси закешовано.

•Спробуйте змінити cacheName і переконайтеся, що кеш правильно оновлюється.

7 Використовуємо service workers для кешування даних прогнозів

Вибір правильної стратегії кешування для ваших даних це життєво важливе питання, що залежить від типу даних, які представляє ваша програма. Наприклад, чутливі до часу дані, такі як погода або ціна акцій, повинні бути максимально свіжими, а ось аватари або контент статей можуть оновлюватися менш часто.

Стратегія спочатку-кеш-потім-мережу ідеальна для нашого застосування. Воно максимально швидко представляє на екрані дані, а потім оновлює їх після того, як мережа поверне найостаннішу інформацію. На відміну від спочатку-мережу-потім-кеш стратегії, користувачеві не потрібно чекати, поки fetch відвалиться через таймаут, щоб отримати кешовані дані.

Спочатку-кеш-потім-мережу означає, що нам треба зробити два асинхронних запити, один до кешу, другий до мережі. Наш мережевий запит не вимагає великих змін, але нам потрібно модифікувати service worker для кешування відповіді перед його виведенням.

У нормальних обставинах кешовані дані будуть повернуті практично миттєво, застосунок зможе використовувати останні з існуючих даних. Тоді, коли мережевий запит поверне нові дані, застосунок оновить їх для користувача.

Перехоплення мережевого запиту і кешування відповіді

Нам треба змінити service worker для перехоплення запитів до погодного API і збереження відповідей в кеші, щоб ми могли легко отримати до них доступ пізніше. У спочатку-кеш-потім-мережу стратегії ми очікуємо, що мережева відповідь буде давати нам найостаннішу інформацію. Якщо відповідь не буде отримана, то це теж нормально, так як у нас у застосунку вже є останні збережені дані.

У service worker давайте додамо `dataCacheName`, щоб ми могли відрізнити дані від оболонки програми. Якщо оболонка нашого застосування буде оновлена і старий кеш видалений, то дані залишаться недоторканими і готовими до супер швидкого завантаження. Пам'ятайте про те, що якщо формат ваших даних в майбутньому зміниться, то вам треба буде опрацювати цю ситуацію і переконатися, що оболонка і контент залишаються синхронізованими.

Додайте це в початок нашого `service-worker.js`:

```
var dataCacheName = 'weatherData-v1';
```

Далі нам треба змінити обробник події `fetch` для обробки запитів до API окремо від інших запитів.

```
self.addEventListener('fetch', function(e) {
  console.log('[ServiceWorker] Fetch', e.request.url);
  var dataUrl = 'https://publicdata-weather.firebaseio.com/';
  if (e.request.url.indexOf(dataUrl) === 0) {
    // Put data handler code here
  } else {
    e.respondWith(
```

```
        caches.match(e.request).then(function(response) {
          return response || fetch(e.request);
        })
      );
    }
});
```

Код перехоплює запити і перевіряє, чи починається URL з адреси погодного API. Якщо це так, то ми використовуємо `fetch` для запиту. Коли відповідь повернеться, наш код відкриває кеш, клонує відповідь, зберігає в кеші і, нарешті, повертає відповідь сервера початкового запиту.

Давайте замінимо `// Put data handler code here` на наведене нижче:

```
    e.respondWith(
      fetch(e.request)
        .then(function(response) {
          return caches.open(dataCacheName).then(function(cache) {
            cache.put(e.request.url, response.clone());
            console.log('[ServiceWorker] Fetched&Cached Data');
            return response;
          });
        })
    );
```

Наш застосунок все ще не працює в офлайні. Ми впровадили кешування і відновлення оболонки застосунку, але, не дивлячись на те, що ми кешуємо дані, ми як і раніше залежні від мережі.

Робимо запити

Як згадувалося раніше, із застосунком потрібно робити два асинхронних запиту - перший до кешу, другий до мережі. Застосунок використовує об'єкт `caches`, доступний в `window` для доступу до кешу і отримання останніх даних. Це прекрасний приклад прогресивного поліпшення, так як об'єкт `caches` може бути недоступний у всіх браузерах, і якщо його немає, то мережевий запит все одно буде працювати.

Щоб реалізувати це нам знадобиться:

1. Перевірити, чи доступний об'єкт `caches` в глобальному об'єкті `window`.

2. Запитати дані з кешу.

•Якщо серверний запит все ще не виконано, оновити застосунок кешованими даними.

3. Запитати дані з сервера.

•Зберегти дані для подальшого швидкого використання.

•Оновити застосунок новими даними, отриманими з сервера.

Відстеження очікуваних запитів

Для початку давайте додамо мітку, яку ми будемо використовувати для запобігання поновлення програми з кешу для тих рідкісних випадків, коли XHR відповідь прийде швидше кешу. Додайте `hasRequestPending: false,` (включаючи останню кому) наверх визначення об'єкта `app`у файлі `scripts/app.js`.

Отримання даних з кешу

Далі нам треба перевірити, чи існує об'єкт `caches` і запросити останні дані з нього. Додайте наступний код в `app.getForecast`, до коментаря `// Make the XHR to get the data, then update the card`:

```
if ('caches' in window) {
  caches.match(url).then(function(response) {
    if (response) {
      response.json().then(function(json) {
        // Only update if the XHR is still pending, otherwise the XHR
        // has already returned and provided the latest data.
        if (app.hasRequestPending) {
          console.log('updated from cache');
          json.key = key;
          json.label = label;
          app.updateForecastCard(json);
        }
      });
    }
  });
}
```

Оновлення мітки hasRequestPending

Нарешті, нам треба оновити пмітку `app.hasRequestPending`. Відразу після коментаря про здійснення XHR, додайте `app.hasRequestPending = true;`

Далі, в обробнику відповіді XHR `onreadystatechange`, прямо перед `app.updateForecastCard(response)`, встановіть `app.hasRequestPending = false;`

Наш погодний застосунок тепер робить два асинхронних запити для отримання даних, один з `cache` і один за допомогою XHR. Якщо є дані в кеші, вони повертаються і відрисовуються надзвичайно швидко (десятки мілісекунд), а оновлюють картку тільки якщо XHR не виконується. Потім, коли XHR відповідає, картка оновлюється новими даними прямо з API погоди.

Якщо з якихось причин XHR відповідає швидше кешу, мітка `hasRequestPending` запобігає перезапису даних, отриманих з мережі.

Тестуємо

•В консолі ви повинні бачити дві події кожного разу, коли ви оновлюєте сторінку, одне свідчить про те, що дані витягнуті з кешу, і одне про те, що дані отримані з мережі.

•Тепер застосунок повністю може працювати в офлайні. Спробуйте зупинити ваш сервер і відключити мережу, а потім запустити застосунок. Оболонка програми та дані повинні виходити з кешу.

8 Підтримка нативної інтеграції

Нікому не подобається набирати довгі адреси сайтів на мобільній клавіатурі. За допомогою функції Add To home screen ваші користувачі можуть додати посилання на ваш додаток на домашній екран, і воно буде доступно так само, наче було встановлено з магазину застосунків.

Банери установки веб-застосунку і додавання на домашній екран в Chrome для Android

Банери установки веб-застосунку дають вашим користувачам можливість швидко і просто додати веб-застосунок на домашній екран, що спрощує запуск і повернення в нього. Додавання такого банера є досить простим, а Chrome зробить всю роботу за вас. Нам просто треба додати маніфест веб-застосунку з інформацією про програму.

Chrome потім буде розумно використовувати набір критеріїв, включаючи використання service worker-а, наявність SSL і частоту візитів для визначення того, коли показувати банер користувачу. Крім того, людина сама може вручну додати застосунок собі через пункт "Add to Home Screen" в меню Chrome.

Оголошуємо маніфест застосунка за допомогою файлу `manifest.json`

Маніфест веб-застосунка це простий JSON файл, який дає вам, як розробнику, можливість управляти тим, як ваш застосунок представляється користувачеві там, де він очікує його побачити, (наприклад, на домашньому екрані), що може запускати користувач і, що більш важливо, як він може це робити.

Використовуючи маніфест веб-застосунку ви можете:

•Поліпшити показ на домашньому екрані Android

•Запускати застосунок в повноекранному режимі без рядка URL

•Управляти орієнтацією екрана для кращого перегляду

•Визначати "splash screen" і колірну тему сайту

•Відстежувати, запущено ваш застосунок з домашнього екрану чи з рядка браузера

Створіть файл з назвою `manifest.json` у вашій теці `work` і скопіюйте в нього такий вміст:

```
{
```

```
  "name": "Weather",
  "short_name": "Weather",
  "icons": [{
    "src": "images/icons/icon-128x128.png",
      "sizes": "128x128",
      "type": "image/png"
    }, {
      "src": "images/icons/icon-144x144.png",
      "sizes": "144x144",
      "type": "image/png"
    }, {
      "src": "images/icons/icon-152x152.png",
      "sizes": "152x152",
      "type": "image/png"
    }, {
      "src": "images/touch/icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    }, {
      "src": "images/touch/icon-256x256.png",
      "sizes": "256x256",
      "type": "image/png"
    }],
  "start_url": "/index.html",
  "display": "standalone",
  "background_color": "#3E4EB8",
  "theme_color": "#2F3BA2"
}
```

Маніфест підтримує масив іконок, призначених для екранів різних розмірів.

Простий спосіб відстежити, як застосунок запущено, це додати параметр query string під назвою `start_url` і потім використовувати аналітику для відстеження всіх варіантів. Якщо ви будете використовувати цей метод, то не забудьте оновити список файлів, які кешує App Shell для того, щоб і файл з параметром кешувався.

Сказати браузеру про маніфесті

Тепер давайте додамо такий рядок в кінець елемента `<head>`вашого `index.html`:

```
<link rel="manifest" href="/manifest.json">
```

Кращі практики

•Додавайте посилання на ваш маніфест до всіх сторінок, так щоб Chrome отримував його при першому візиті користувача - незалежно від того, до якої сторінки він звернувся.

•Параметр `short_name` має більше значення для Chrome і буде використовуватися саме він (якщо задано), а не поле з ім'ям.

•Використовуйте набір іконок для екранів різних піксельних щільностей. Chrome буде намагатися використовувати іконку, близьку до 48dp - наприклад, 96 точок на 2x пристрої або 144 точок на 3x пристрої.

•Не забудьте включити іконку з розміром, який підходить для сплеш скрін і не забудьте встановити `background_color`.

Ще можна почитати:

Використання банера установки додатка

Елемент Add to Homescreen для Safari в iOS

У вашому `index.html` додайте наступний код в кінець елемента `<head>`:

```
<!-- Add to home screen for Safari on iOS -->
<meta name="apple-mobile-web-app-capable" content="yes">
<meta name="apple-mobile-web-app-status-bar-style" content="black">
<meta name="apple-mobile-web-app-title" content="Weather PWA">
<link rel="apple-touch-icon" href="images/icons/icon-152x152.png">
```

Іконка плитки для Windows

У вашому `index.html` додайте наступний код в кінець елемента `<head>`:

```
<meta name="msapplication-TileImage" content="images/icons/icon-144x144.png">
<meta name="msapplication-TileColor" content="#2F3BA2">
```

Тестуємо

Щоб перевірити, чи працює сайт як прогресивний веб-застосунок, можна взяти Lighthouse. Lighthouse - це розширення для Chrome, яке покаже, наскільки PWA придатний і чи можна його поліпшити. Після установки відкрийте сайт і натисніть значок маяка в верхньому правому куті браузера, а потім «Generate Report». Через кілька секунд відкриється нова вкладка з інформацією про сайті: її можна прочитати цілком, а можна зосередитися на числах зверху і проігнорувати інше:

Ви повинні побачити щось на кшталт

Progressive Web App

Performance

Accessibility

Best Practices