Don't miss out on the action at this year's Chrome Dev Summit, happening on Oct 23rd and 24th. Learn more (https://developer.chrome.com/devsummit/).

はじめてのプログレッシブ ウェブアプリ

By Pete LePage
(https://developers.google.com/web/resources/contributors#petelepage)
Pete is a Developer Advocate

プログレッシブ ウェブアプリ (https://developers.google.com/web/progressive-web-apps)はウェブとアプリの両方の利点を兼ね備えたアプリです。ブラウザのタブで表示してすぐに利用することができ、インストールの必要はありません。使い続けてユーザーとの関係性が構築されていくにつれ、より強力なアプリとなります。不安定なネットワークでも迅速に起動し、関連性の高いプッシュ通知を送信することができます。また、ホーム画面にアイコンを表示することができ、トップレベルの全画面表示で読み込むことができます。

プログレッシブ ウェブアプリとは

プログレッシブ ウェブアプリには以下の特徴があります。

- **段階的** - プログレッシブ・エンハンスメントを基本理念としたアプリであるため、 ブラウザに関係なく、すべてのユーザーに利用してもらえます。
- **レスポンシブ** - パソコンでもモバイルでもタブレットでも、次世代の端末でも、 あらゆるフォームファクタに適合します。
- **ネットワーク接続に依存しない** - Service Worker の活用により、オフラインでも、 ネットワーク環境が良くない場所でも動作します。
- **アプリ感覚** - App Shell モデルに基づいて作られているため、アプリ感覚で操作できます。
- **常に最新** - Service Worker の更新プロセスにより、常に最新の状態に保たれます。
- **安全** - 覗き見やコンテンツの改ざんを防ぐため、HTTPS 経由で配信されます。
- **発見しやすい** - W3C のマニフェストとService Worker の登録スコープにより、 「アプリケーション」として認識されつつ、検索エンジンからも発見することができます。

- **再エンゲージメント可能** - プッシュ通知のような機能を通じで容易に 再エンゲージメントを促すことができます。
- **インストール可能** - ユーザーが気に入ればアプリのリンクをホーム画面に残して おくことができ、アプリストアで探し回る必要はありません。
- **リンク可能** - URL を使って簡単に共有でき、複雑なインストールの必要はありません。

このスタートガイドでは、独自のプログレッシブ ウェブアプリを作成する方法について、設計時の考慮事項や、実装の詳細など、順を追って説明します。この説明に沿って作業することで、 プログレッシブ ウェブアプリの原則に基づいたアプリを作成することができます。

Note: さらに詳しくは、2015 年の Chrome Dev Summit で Alex Russell が行った、プログレッシブ ウェブアプリ (https://www.youtube.com/watch?v=MyQ8mtR9WxI)についての講演内容をご覧ください。

作成するもの

(https://weather-pwa-sample.firebaseapp.com/final/)
Try it
(https://weather-pwa-sample.firebaseapp.com/final/)

このコードラボでは、プログレッシブ ウェブアプリの技法を使って お天気ウェブアプリを作成します。

- **段階的** - 徐々に機能が強化されていくようにします。
- **レスポンシブ** - あらゆるフォームファクタに適合するようにします。
- **ネットワーク接続に依存しない** - Service Worker で App Shell をキャッシュします。
- **アプリ感覚** - アプリと同様の操作で、都市の追加やデータの更新を行えるようにします。
- **常に最新** - Service Worker で最新のデータをキャッシュします。
- **安全** - HTTPS 対応のホストにアプリを実装します。
- **発見しやすい** / インストール可能 - マニフェストを指定し、検索エンジンでアプリを簡単に見つけられるようにします。
- **リンク可能** - ウェブページとして活用できるようにします。

学習する内容

- 「App Shell」方式に基づいてアプリを設計し構築する方法
- アプリをオフラインで動作可能にする方法
- データを保存し、後からオフラインで使用できるようにする方法

必要なもの

- Chrome 47 以上
- HTML、CSS 、JavaScript の基本知識

このスタートガイドではプログレッシブ ウェブアプリに絞って説明するため、いくつかの概念については説明を省略しています。また、単にコピーして貼り付けるだけの コードブロック（スタイルや関連性のない JavaScript）を用意している箇所もあります。

App Shellを構築する

App Shell とは、プログレッシブ ウェブアプリのユーザー インターフェースが機能する ための最小限の HTML、CSS、JavaScript であり、高いパフォーマンスを発揮するために 必要な要素の 1 つです。最初の読み込みは高速で行われ、読み込み後すぐにキャッシュ されます。それ以降、毎回の読み込みは行われず、必要なコンテンツだけが取得されます。

App Shell のアーキテクチャでは、アプリケーションの核となるインフラストラクチャと ユーザー インターフェースを、データから切り離して扱います。ユーザー インターフェース とインフラストラクチャはすべて、Service Worker によりローカルにキャッシュされるので、以降の読み込み時にはすべてを読み込まなくても必要なデータだけを取得すればよい ことになります。

App Shell は、ネイティブ アプリの作成時にアプリストアに公開するコードバンドルの ようなもの、と言うこともできます。これはアプリを起動するために必要な基本の構成要素ですが、多くの場合データは含まれません。

App Shell アーキテクチャを使用する理由

App Shell アーキテクチャを採用すると、スピードを追及でき、プログレッシブ ウェブアプリにネイティブ アプリのような特性を持たせることができます。 つまり、アプリストア

を一切介することなく、瞬時の読み込みや定期的な更新が可能です。

App Shell を設計する

最初のステップでは、中心となる構成要素に細分化して設計を検討します。

次のことを考えてみてください。

- 画面に即座に表示しなければならないものは？
- その他、アプリに重要なユーザー インターフェース要素は？
- App Shell に必要なサポート リソースは？（例: 画像、JavaScript、スタイル）

これから最初のプログレッシブ ウェブアプリとして、お天気アプリを作ります。 主要な構成要素は次のとおりです。

- タイトル ヘッダー、追加 / 更新ボタン
- 予報カードのコンテナ
- 予報カードのテンプレート
- 都市の追加用ダイアログ ボックス
- 読み込みインジケータ

より複雑なアプリを設計する場合は、最初に読み込む必要のないコンテンツは後でリクエスト するようにできます。読み込んだコンテンツは、今後の使用に備えてキャッシュします。 たとえば、New City ダイアログの読み込みを、初回エクスペリエンスの表示が終わるまで 遅らせるとともに、アイドル時の処理を組み込みます。

App Shell を実装する

どのようなプロジェクトでも開始にはいくつかの方法がありますが、通常は Web Starter Kit の利用をおすすめしています。ただし今回は、プロジェクトをできるだけ 簡単なものにしてプログレッシブ ウェブアプリに集中できるように、必要なリソースを すべてご用意しました。

コードのダウンロード

簡単に使えるように、このプログレッシブ ウェブアプリガイドのすべてのコードをZIP ファイルとしてダウンロード (https://developers.google.com/web/fundamentals/getting-started/codelabs/your-first-pwapp/pwa-weather.zip)
することができます。各ステップで必要となるすべての リソースがZIPファイル内から利用可能です。

App Shell の HTML を作成する

可能な限りクリーンな状態で始めることを確認するために、ブランドの新しい**index.html**ファイルから始めて、App Shell を構築する (#step-01)で取り上げた中心的な構成要素を 追加しましょう。

今回の構成要素をもう一度挙げます。

- タイトル ヘッダー、追加 / 更新ボタン
- 予報カードのコンテナ
- 予報カードのテンプレート
- 都市の追加用ダイアログ ボックス
- 読み込みインジケータ

```
<!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 App</title>
  <!-- Insert link to styles.css here -->
</head>
<body>
  <header class="header">
    <h1 class="header__title">Weather App</h1>
    <button id="butRefresh" class="headerButton"></button>
    <button id="butAdd" class="headerButton"></button>
  </header>

  <main class="main" hidden>
    <!-- Insert forecast-card.html here -->
  </main>

  <div class="dialog-container">
    <!-- Insert add-new-city-dialog.html here -->
```

```
  </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>
```

mainコンテンツが**hidden**であり、ローダーがデフォルトで表示されることに注目 してください。ページが読み込まれるとすぐにローダーがユーザーの目に入り、これから コンテンツが読み込まれることがはっきりわかるようになっています。

次に、予報カードを追加し、そして New City ダイアログを追加しましょう。時間を節約するために、それらは**resources**ディレクトリの中で提供されていますので、対応する場所 にそれらを簡単にコピーアンドペーストすることができます。

主要な UI コンポーネントにスタイルを追加する

ここで、コアスタイルを追加します。ビルドやデプロイプロセスの一環として、ドキュメント ボディにそれらのコアスタイルをインラインで記述したいところですが、今回は分離された CSSファイルにそれらを置きましょう。

index.htmlファイル内で、**<!-- Insert link to styles here -->**を以下に 置き換えます。

```
<link rel="stylesheet" type="text/css" href="styles/inline.css">
```

時間を節約するために、あなたが使えるstylesheet (https://weather-pwa-sample.firebaseapp.com/styles/inline.css) をすでに作成してあります。それをレビューし、そしてあなた自身でそれをカスタマイズする ために数分使ってください。

Note: 個別に各アイコンを指定すると、画像のスプライトを使用する場合と比較して効率が悪く見えるかもしれませんが、アプリのシェルの一部として後でそれらをキャッシュし、ネットワーク要求をする必要なく常に利用可能であることを保証します。

実行と調整

今が実行する絶好の時です。これらがどのような見た目になるかを見て、そしてあなたが行いたい調整をしてください。`main`コンテナから`hidden`属性を削除し、そしてカードに幾つかの架空のデータを追加することによって、予報カードの描画をテストしてください。

Note: 時間短縮のため、また確実な材料を使って作業を始められるように、マークアップとスタイルをご用意しました。次のセクションでは自分でコードを書く機会もあります。

このアプリは現状ほぼレスポンシブですが、完全ではありません。レスポンシブ性を改善し、 異なるデバイスを横断して本当に光り輝かせるために、追加のスタイルを加えてみてください。 また、あなた自身でできることを考えてみてください。

主要な JavaScript ブートコードを追加する

ここまでで、ユーザー インターフェースの大半が揃いました。次はすべてが動作するように コードを組み合わせます。App Shell の他の部分と同様に、中心的なエクスペリエンスを 実現するのに重要なコードがどれで、後で読み込むことのできるコードがどれかを意識して 作業してください。

今回のブートコードには次の要素が含まれています。

- アプリに必要な基本情報を含んでいる`app`オブジェクト。
- すべてのボタンのイベント リスナー。ヘッダーのボタン（`add/refresh`）と、 都市の追加ダイアログのボタン（`add/cancel`）があります。
- 予報カードを追加または更新するためののメソッド（`app.updateForecastCard`）。
- Firebase Public Weather API から最新の天気予報データを取得するためのメソッド（`app.getForecast`）。
- 表示のテスト用の架空データ（`fakeForecast`）。

JavaScript コードを追加してください。

1. `resources`ディレクトリからあなたの`scripts`フォルダへ`step3-app.js`ファイルをコピーして、それを`app.js`に名前変更してください。
2. `index.html`ファイル内で、新しく作られた`app.js`へのリンクを追加してください。

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

テスト

基本の HTML、スタイル、JavaScript が揃ったので、アプリをテストしましょう。 この時点で行われる動作は限定的ですが、コンソールにエラーが書き込まれないことを確認 してください。

架空の天気データがどのように表示されるかを確認するには、**app.js**ファイルの末尾に 次の行を追加してください。

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

試す (https://weather-pwa-sample.firebaseapp.com/step-03/)

最初の読み込みを高速に行えるようにする

プログレッシブ ウェブアプリは、高速に起動してすぐに使えるものでなければなりません。現在の状態では、お天気アプリは高速に起動しますが、データがないため使えるものになっていません。AJAX リクエストを使ってデータを取得することもできますが、それではリクエストを余分に行うことになり、最初の読み込みに時間がかかってしまいます。そこで、最初の読み込みでは実際のデータを指定します。

天気予報データを挿入する

このコードラボでは天気予報の静的データをあらかじめ指定します。ただし本番のアプリでは、 ユーザーの IP アドレスから判定できる地域情報に基づいて、最新の天気予報データを サーバーから挿入することになります。

即時呼び出しの関数式の内部に以下のコードを追加します。

```
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}
    ]
  }
};
```

次に、前にテストのために作成した**fakeForecast**データはもう使うことはないので、削除します。

初回実行時と区別する

さて、前述の情報を表示するタイミングはどのように判断するのでしょうか。今後お天気アプリがキャッシュから取得されて読み込まれるとき、この情報の関連性は失われているかもしれません。ユーザーが次にアプリを読み込むときには都市が変わっている可能性もあります。そのため、これまでに確認された都市に限らず、該当する都市の情報を読み込む 必要があります。

ユーザーが登録した都市のリストのようなユーザー設定は、IndexedDB などの高速な ストレージ システムを利用してローカルに保存しておく必要があります。今回はできるだけ 簡単な例にするために localStorage (https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage) を使用しましたが、これは本番のアプリには適していません。**localStorage** では ブロッキングな同期の仕組みが使われており、端末によっては著しくスピードが低下する 可能性があるためです。

Note: 補習: **localstorage** の実装を idb (https://www.npmjs.com/package/idb) に置き替えてみましょう。

まず、**app.js** 内の即時呼び出しの関数式の最後に、ユーザー設定の保存に必要な コードを追加します。

```
// Save list of cities to localStorage, see note below about localStorage.
app.saveSelectedCities = function() {
  var selectedCities = JSON.stringify(app.selectedCities);
  // IMPORTANT: See notes about use of localStorage.
  localStorage.selectedCities = selectedCities;
};
```

次に、スタートアップ コードを追加します。このコードでは、ユーザーが登録している 都市があるか確認してその都市を読み込むか、サーバーからのデータを使用します。 **app.js** ファイル内の、先程追加したコードの後に次のコードを追加しましょう。

```
/*****************************************************************************
 *
 * Code required to start the app
 *
 * NOTE: To simplify this getting started guide, 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 (https://www.npmjs.
 *   SimpleDB (https://gist.github.com/inexorabletash/c8069c042b734519680c (htt
 *
 ****************************************************************************/

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();
}
```

最後に、ユーザーが新しい都市を追加したときには必ず都市のリストを保存することを 忘れないでください。それには、**butAddCity** ボタンのイベント ハンドラに **app.saveSelectedCities();** を追加します。

テスト

- 初回実行時には、**initialWeatherForecast** からの予報が即座に表示される必要があります。
- 右上の + アイコンをクリックして都市を追加し、2 つのカードが表示されることを確認します。
- ブラウザを更新して、アプリに両方の予報が読み込まれ最新情報が表示されることを確認します。

試す (https://weather-pwa-sample.firebaseapp.com/step-04/)

Service Worker を使って App Shell を事前キャッシュする

プログレッシブ ウェブアプリは、高速に動作し、かつインストール可能でなければなりません。 つまり、オンラインでも、オフラインでも、接続が不安定または遅い場所でも動作することが 求められます。これを実現するには、Service Worker を使用して App Shell をキャッシュし、常にすばやく利用できる状態を維持する必要があります。

Service Worker をよくご存知ない場合は、Service Worker の概要記事 (https://developers.google.com/web/fundamentals/getting-started/primers/service-workers)をご覧ください。 この記事では、Service Worker でできることや、Service Worker のライフサイクルなど、 基本事項を説明しています。

Service Worker を介して提供する機能は、プログレッシブ・エンハンスメントの 1 つとして 考えるべきで、こうした機能を追加するのはブラウザでサポートされている場合のみにする 必要があります。たとえばネットワークを使用できない状況では、Service Worker を使って App Shell とアプリのデータをキャッシュすることができます。一方 Service Worker がサポートされていない場合は、オフラインのコードを呼び出すことなく、最小限のユーザー エクスペリエンスのみを提供します。段階的な機能向上を提供するための機能検出に伴う オーバーヘッドはわずかで、機能をサポートしていない古いブラウザが使用されている場合、 問題が起こることはありません。

Note: Service Worker の機能は HTTPS 経由でアクセスしたページでのみ使用できます（テストを円滑に進められるように、`https://localhost` (https://localhost) またはこれに相当する URL でも動作するようになっています）。この制約が課せられる理由については、Chromium チームの投稿記事 Prefer Secure Origins For Powerful New Features (http://www.chromium.org/Home/chromium-security/prefer-secure-origins-for-powerful-new-features)（強力な新機能に対する「セキュア オリジン」の採用傾向について）をご覧ください。

Service Worker が利用可能な場合に登録する

オフラインでもアプリが動作するようにするには、まず Service Worker を登録します。Service Worker は、ウェブページを開いていなくても、またはユーザーの操作がなくても、 バックグラウンドで処理を進めることのできるスクリプトです。

登録の手順は次のとおりです。

1. Service Worker のコードを提供する JavaScript ファイルを作成します。

2. 作成した JavaScript ファイルを Service Worker として登録するようブラウザに指定します。

まず、アプリケーションのルートフォルダ（**your-first-pwapp-master/work**）に **service-worker.js** という空のファイルを作成します。このファイルは アプリケーションのルートに置く必要があります。このファイルが置かれている ディレクトリによって Service Worker のスコープが定義されるためです。

次に、ブラウザで Service Worker がサポートされているかどうかを確認し、サポート されている場合は Service Worker を登録します。方法は、 **app.js** ファイルに、 次のコードを追加します。

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

サイトのアセットをキャッシュする

Service Worker を登録すると、ユーザーがページに初めてアクセスしたときに **インストール** イベントが呼び出されます。このイベント ハンドラで、アプリケーションに 必要なすべてのアセットをキャッシュします。

Note: 下記のコードは本番環境では使用**しないでください**。これはごく基本的な用途に対応したコードで、本番環境で使用すると App Shell が更新されない状況に陥る可能性があります。後のセクションでこの実装に伴う危険性とその回避方法を説明していますので、必ずご確認ください。

Service Worker が呼び出されると、caches オブジェクトが開かれ、App Shell の 読み込みに必要なアセットが挿入されます。**service-worker.js** ファイルの末尾に次の コードを追加してください。

```
var cacheName = 'weatherPWA-step-5-1';
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);
    })
```

```
  );
});
```

まず、**caches.open()** を使用してキャッシュを開き、キャッシュに名前を付けます。 キャッシュに名前を付けることでファイルのバージョン管理が可能になります。また、 データと App Shell を切り離し、お互いに影響を与えることなく個別に更新できるように なります。

キャッシュが開いたら、**cache.addAll()** を呼び出します。これは URL のリストを取り、該当のファイルをサーバーから取得して応答をキャッシュに追加します。**cache.addAll()** は最小単位であるため、ファイルのうち 1 つでも取得できないものがあると、キャッシュの ステップそのものが失敗に終わります。

Service Worker に変更を加えるときには必ず **cacheName** を変更し、キャッシュから 最新版のファイルが取得されるようにします。使わないコンテンツやデータのキャッシュは 定期的に削除することが重要です。イベント リスナーを追加して、すべてのキャッシュキー の取得と使われていないキャッシュキーの削除を行う **activate** イベントを待機します。

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

最後に、App Shell に必要なファイルのリストを更新しましょう。画像、JavaScript、 スタイルシートなど、アプリに必要なすべてのファイルを配列に含めます。

```
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'
];
```

Note: ファイル名はパス全体を含めるようにしてください。たとえばアプリは `index.html` から配信されますが、リクエストには「`/`」も含まれています（サーバーはルートフォルダがリクエストされたときに `index.html` を送信します）。`fetch` メソッドでこの処理を行うこともできますが、大文字と小文字の組み合わせに注意が必要となり、かえって複雑になる可能性があります。

まだアプリはオフラインで動作しません。App Shell の構成要素のキャッシュは できましたが、ローカル キャッシュからの読み込みを行う必要があります。

キャッシュからApp Shell を配信する

Service Worker を使うと、プログレッシブ ウェブアプリから送信されたリクエストを 傍受して Service Worker 内部で処理することができます。つまり、リクエストの 処理方法を決めることができ、キャッシュした応答を配信することも可能です。

例:

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

`service-worker.js` file: それでは、キャッシュからApp Shell を配信しましょう。**`service-worker.js`** ファイルの末尾に次のコードを追加します。

```
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);
    })
  );
});
```

内側から外側に向かって説明すると、まず **caches.match()** を使用して、**fetch** イベントを呼び出したウェブ リクエストを評価し、キャッシュからのデータが利用可能か どうかを確認します。次に、キャッシュ データで応答するか、fetch を使用して ネットワークからコピーを取得します。そして、**e.respondWith()** を使用して ウェブページに **response** を返します。

Note: `[ServiceWorker]` がコンソールにログ出力されない場合は、`cacheName` を変更していることを確認してページを再度読み込んでください。これで解決できない場合は、「運用中の Service Worker のテストを行う際のヒント」の項をご覧ください。

特殊なケースに関する注意

何度も言いますが、このコードは**本番環境では使用しないでください。**このコードは、 多くの特殊ケースには対応していません。

変更のたびにキャッシュキーの更新が必要

たとえば、このキャッシュ方法では、コンテンツを変更するたびにキャッシュキーを更新する 必要があります。そうしないとキャッシュは更新されず、古いコンテンツが配信されることに なります。このため、プロジェクトでの作業中は、変更を行うたびにキャッシュキーを変更 するようにしてください。

変更のたびにキャッシュ全体の再ダウンロードが必要

もう 1 つの注意点は、ファイルを変更するとキャッシュ全体が無効になるため、 再ダウンロードが必要になるということです。つまり、1 文字のスペルミスを修正した だけでも、キャッシュが無効になり、もう一度全体をダウンロードしなければならなく なります。これはあまり効率的とは言えません。

ブラウザ キャッシュによってService Worker のキャッシュ更新が妨害される

さらにもう 1 つの注意点として、インストール処理中に行う HTTPS リクエストは ネットワークに直接送信し、ブラウザのキャッシュから応答が返されないようにすることが 重要です。そうしないと、キャッシュされた古い応答がブラウザから返され、その結果、Service Worker のキャッシュが更新されなくなります。

本番環境での「キャッシュ優先」戦略の使用

今回のアプリでは「キャッシュ優先」の戦略を使用します。この場合、キャッシュされたコンテンツのコピーがあれば、ネットワークに問い合わせを行わずにキャッシュのコピーを 返すことになります。「キャッシュ優先」の戦略は簡単に実装できる一方で、後から さまざまな課題を生む原因になることがあります。ホストページと Service Worker の登録内容のコピーがキャッシュされると、Service Worker の設定を変更することは 極めて困難です（設定は定義された場所に依存するため）。また、実装したサイトの更新も 非常に複雑になります。

特殊なケースを回避するには

こうした特殊なケースを回避するには、sw-precache (https://github.com/GoogleChrome/sw-precache) のようなライブラリを使用します。こうしたライブラリを使用するとデータの有効期限を 適切に管理することができます。また、リクエストがネットワークに直接送信されるように なるとともに、あらゆる煩雑な作業から解放されます。

運用中の Service Worker をテストする際のヒント

Service Worker のデバッグは困難な場合があります。さらに、キャッシュを使用する 場合に想定どおりにキャッシュが更新されないと、さらに解決に苦労することになります。 典型的な Service Worker のライフサイクルとコードのバグに挟まれて、身動きが とれなくなってしまうでしょう。しかし、こうした作業を容易にしてくれるツールが あります。

ヒント:

- Service Worker の登録が解除された後も、関連するブラウザ ウィンドウが閉じられる まで、Service Worker が表示されることがあります。
- アプリに対して複数のウィンドウが開いている場合、新しい Service Worker の動作が 有効になるのは、すべてのウィンドウが再読み込みされて最新の Service Worker に更新されてからとなります。
- Service Worker の登録を解除してもキャッシュは消去されません。このため、 キャッシュ名が変わっていないと古いデータが使用される可能性があります。
- Service Worker が既に存在する場合、新しい Service Worker を登録しても、 即時コントロール (https://github.com/GoogleChrome/samples/tree/gh-pages/service-worker/immediate-control) を指定していなければ新しい Service Worker は機能しません。ページを再読み込みして 初めて機能するようになります。

作業に役立つページ: chrome://serviceworker-internals

Chrome の Service Worker ページ（`chrome://serviceworker-internals`）を 利用すると、既存の Service Worker を停止し、登録を解除して、新たに開始するという 一連の操作を簡単に行うことができます。このページでは、Service Worker から デベロッパー ツールを起動して、Service Worker のコンソールにアクセスすることも できます。

テスト

- Chrome DevTools を開き、Service Worker が適切に登録され正しいリソースが キャッシュされていることを確認します。
- **`cacheName`** を変更してみて、キャッシュが適切に更新されることを確認します。

試す (https://weather-pwa-sample.firebaseapp.com/step-05/)

Service Worker を使ってアプリケーション データをキャッシュする

データに正しいキャッシュ戦略を選択することは重要であり、これはアプリで提供する データの種類によって決まります。たとえば、天気情報や株価など時間の経過とともに 変動するデータはできるだけ最新のものでなければなりませんが、アバターの画像や記事の コンテンツなどは更新の頻度が比較的少なくても問題はないと考えられます。

今回のアプリに適しているのは、**まずキャッシュ、次にネットワークという優先**順でデータを 取得する戦略です。この戦略では、画面にとにかく早くデータを表示し、その後ネットワーク から最新のデータが返された時点でデータの更新を行います。**キャッシュではなく ネットワークを優先**した場合、ネットワークからの fetch がタイムアウトになってからキャッシュ データが取得されることになり、待ち時間が発生してしまいます。キャッシュ優先の場合はこうした待ち時間がなくなります。

キャッシュ、ネットワークの順でデータを取得するには、キャッシュに 1 回、 ネットワークに 1 回、合計 2 回の非同期リクエストを送信する必要があります。 アプリのネットワーク リクエストにはそれほど変更を加える必要はありませんが、 Service Worker には、応答を返す前にキャッシュを行うよう変更を加える必要が あります。

以上の理由から、非同期リクエストを 2 回（キャッシュに 1 回、ネットワークに 1 回） 行う必要があります。通常は、キャッシュ データはほぼ瞬時に返され、最近のデータとしてアプリで利用可能になります。そしてネットワークのリクエストが返されると、ネットワーク からの最新データを基にアプリが更新されます。

ネットワーク リクエストを傍受して応答をキャッシュする

Service Worker に対し、Weather API へのリクエストを傍受するように、また後の アクセスを容易にするためその応答を Cache に格納するように変更を加えます。 **キャッシュ、ネットワークの順**にデータを取得する戦略では、ネットワークの応答を 「確実な情報源」として想定し、常に最新の情報を提供するものとして位置づけます。 ネットワークからデータを取得できない場合は、アプリで最新のキャッシュ データを取得 しているので、ネットワークで失敗しても問題はないということになります。

Service Worker に **dataCacheName** を追加し、アプリケーションのデータと App Shell を切り離せるように設定しましょう。こうすると、App Shell が更新されて 古いキャッシュが消去されても、データは変更されず高速な読み込みに対応できます。なお、 将来データ形式が変わった場合は、App Shell とコンテンツの同期を確保しつつ新しい 形式に対応する方法が必要になります。

service-worker.js ファイルの先頭に次の行を追加します。

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

次に、**fetch** イベント ハンドラに変更を加え、データ API へのリクエストを他の リクエストと別に処理できるようにする必要があります。

```
self.addEventListener('fetch', function(e) {
  console.log('[ServiceWorker] Fetch', e.request.url);
  var dataUrl = 'https://publicdata-weather.firebaseio.com/ (https://publicdata-weat
  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);
      })
    );
  }
});
```

このコードでは、リクエストを傍受し、URL の先頭が Weather API のアドレスかどうかを確認します。URL の先頭が Weather API のアドレスであれば、**fetch** を使用して リクエストを行います。応答が返されたらキャッシュを開き、応答をコピーして格納した後、 リクエストの送信元に応答を返します。

次に、コードの **// Put data handler code here** の部分を以下のコードに 置き換えます。

```
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;
      });
    })
);
```

このアプリはまだオフラインでは動作しません。App Shell のデータのキャッシュと取得を実装しましたが、データをキャッシュできてもまだネットワークに依存している状態です。

リクエストを行う

前に説明したとおり、アプリではキャッシュに 1 回、ネットワークに 1 回、合計 2 回の 非同期リクエストを送信する必要があります。アプリでは **window** で利用可能な **caches** オブジェクトを使ってキャッシュにアクセスし、最新のデータを取得します。 これはプログレッシブ・エンハンスメントを実装する場合の良い例です。すべてのブラウザで **caches** オブジェクトが利用可能とは限らず、**caches** オブジェクトが利用できない ときはネットワーク リクエストが引き続き動作可能でなければならないからです。

必要な手順は次のとおりです。

1. グローバルな **window** オブジェクトにおいて、**caches** オブジェクトが利用可能か どうかを確認します。
2. キャッシュにデータをリクエストします。
    a. サーバーへのリクエストでまだ応答がない場合は、キャッシュ データを使ってアプリを更新します。
3. サーバーにデータをリクエストします。
    a. データを保存し、後ですばやくアクセスできるようにします。
    b. サーバーからの最新データを使ってアプリを更新します。

まれに、キャッシュよりも先に XHR が応答することがあります。このような場合に キャッシュによってアプリが更新されないように、まずフラグを追加しましょう。**app** オブジェクトに **hasRequestPending: false** を追加します。

次に、**caches** オブジェクトが存在するかどうかを確認し、存在する場合はそこから 最新のデータをリクエストします。方法は、XHRが作られる前に、**app.getForecast** に 次のコードを追加します。

```
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);
        }
      });
    }
  });
}
```

最後に、**app.hasRequestPending** フラグを更新します。それには、XHR の作成の前に **app.hasRequestPending = true;** を追加し、XHR の応答ハンドラで **app.updateForecastCard(response)** の直前に **app.hasRequestPending = false;** と設定します。

これで、お天気アプリでは、キャッシュから 1 回、XHR を介して 1 回、合計 2 回の 非同期リクエストが行われるようになりました。キャッシュにデータが存在する場合は そのデータが返され、XHR からの応答がなければキャッシュ データが高速（10s/ms） に表示されてカードが更新されます。その後、XHR から応答があると、Weather API から直接取得した最新のデータを使ってカードが更新されます。

何らかの理由でキャッシュより早く XHR から応答があった場合は、**hasRequestPending** フラグにより、ネットワークの最新データにキャッシュ データが上書きされる事態が回避されます。

テスト

- コンソールで、更新のたびに 2 つのイベント（キャッシュからデータが取得されたことを 示すイベントと、ネットワークからデータが取得されたことを示すイベント）が表示される ことを確認します。
- この時点で、アプリは完全にオフラインで動作するようになっています。開発用のサーバー を停止し、ネットワークの接続を切断して、アプリを実行してみてください。App Shell とデータの両方がキャッシュから配信されるようになります。

試す (https://weather-pwa-sample.firebaseapp.com/step-07/)

Translated By:

By Yoichiro Tanaka
(https://developers.google.com/web/resources/contributors#yoichiro)
Yoichiro is a Google Developers Expert

Last updated 5月 18, 2017.

Chromium Blog
The latest news on the Chromium blog

GitHub
Fork our API code samples and other open-source projects.

Twitter
Connect with @ChromiumDev on Twitter

Videos
Check out the Web Developer Relations team's videos

Events
Attend a developer event and get hacking