プロジェクトベースのダウンロード

ハンズオン用のプロジェクトベースは以下のURLでダウンロードできます。ダウンロードしたら解凍してください。

https://github.com/hifivemania/pwa_handson

解凍後、そのフォルダの中に以下のコマンドを実行します。Node.jsを事前にインストールしてください（※バージョンは8以上を使ってください）。

```
npm i
```

これで必要なライブラリがインストールされます。

Webサーバを立ち上げる

ライブラリをインストールすると以下のコマンドでWebサーバが立ち上がります。

```
npm run dev
```

Webサーバは http://localhost:3000/ または http://127.0.0.1:3000/ でアクセスできます。

ファイル構成について

サンプルのプロジェクトは以下のような構成になっています（一部）。Monacaの場合は public を www と読み替えてください。

```
├── keygen.js（WebPush用のキーファイルを生成します）
├── public （Webブラウザからアクセスするファイルが入っています）
|   ├── icon.png（WebPush用のアイコンファイルです）
|   ├── index.html（TodoアプリのUIです）
|   ├── js（Webブラウザから読み込むJavaScriptファイル群です）
|   |   ├── app.js（Service Workerの処理が記述されています）
|   |   ├── app.push.js（WebPushに関する処理が記述されています）
|   |   └── todo.js（Todoアプリのコードです）
|   ├── manifest.json（アプリ用のマニフェストファイルです）
```

```
|   ├── sw.js（Service Workerです）
|   └── vendors（依存ライブラリです）
|       ├── bootstrap（UIフレームワークのBootstrapです）
|       |   ├── css
|       |   |   └── bootstrap.min.css
|       |   └── js
|       |       └── bootstrap.bundle.min.js
|       ├── hifive
|       |   ├── ejs-h5mod.js（テンプレートエンジンejsのファイルです）
|       |   ├── h5.css（hifive用のCSSです）
|       |   ├── h5.dev.js（HTML5用のMVCフレームワークです。こちらは開発用です）
|       |   ├── h5.js（HTML5用のMVCフレームワークです）
|       |   └── h5.js.map
|       └── jquery（jQueryです）
|           └── jquery.min.js
├── push.js（WebPushを配信します）
└── server.js （Webサーバを立てます）
```

ハンズオンの内容について

資料は大きく分けて3部構成になります。第1章 アプリの説明は共通です。

第1部 アプリ化を体験する

- 第2章 マニフェストファイルを作る
- 第3章 Service Workerのインストールと有効化
- 第4章 Service Workerを使った表示高速化、オフライン対応について

第2部　Todoアプリのオフライン化

- 第5章 Todoの表示処理をオフライン対応させる
- 第6章 Todoの投稿処理をオフライン対応

第3部 WebPush通知を体験する

- 第7章 Webリモートプッシュ通知を実装する

各部は独立していますので、第2部から体験も可能です。では第1章 アプリの説明に進みましょう。

第1章 アプリの説明

この章ではコーディングはしません。説明だけです

今回ハンズオンを体験するアプリの紹介です。アプリはシンプルなTodoアプリとなっています。特徴は以下の通りです。

TodoのCRUD操作

Todoは一覧、追加、削除ができます。

サーバとの通信

Todoはサーバに登録され、そこから読み込みと表示や削除を行います。通信はAjaxを用い、画面の再描画は行いません。

デザインはBootstrapベース

今回はデザインフレームワークとしてBootstrapを採用しています。アイコンはFont Awesomeを使っています。

サーバはGoogle Chrome機能拡張

ローカルの開発環境にGoogle Chrome機能拡張を利用しています。

Web Server for Chrome - Chrome ウェブストア

フレームワークにHifiveを採用

hifiveはWeb業務システムに特化したMVCフレームワークです。jQueryを用いており、テンプレートエンジンにEJSを採用しています。

追加する機能

ベースになるTodoアプリには上記の機能がすべて含まれています。次の章からこのWebアプリケーションをPWA化していきます。では次にマニフェストファイルを作るに進みます。

第2章 マニフェストファイルを作る

マニフェストファイルはアプリの仕様が書かれたJSONファイルです。多くの場合manifest.json という名前で作成します。これをHTMLの中で読み込みます。

元：

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

修正後：

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

> 今回はTodoアプリを作っている最中にインストールされてしまうと問題があるのでコメントアウトしています。アプリ化を体験する際にはコメントアウトを外してください。

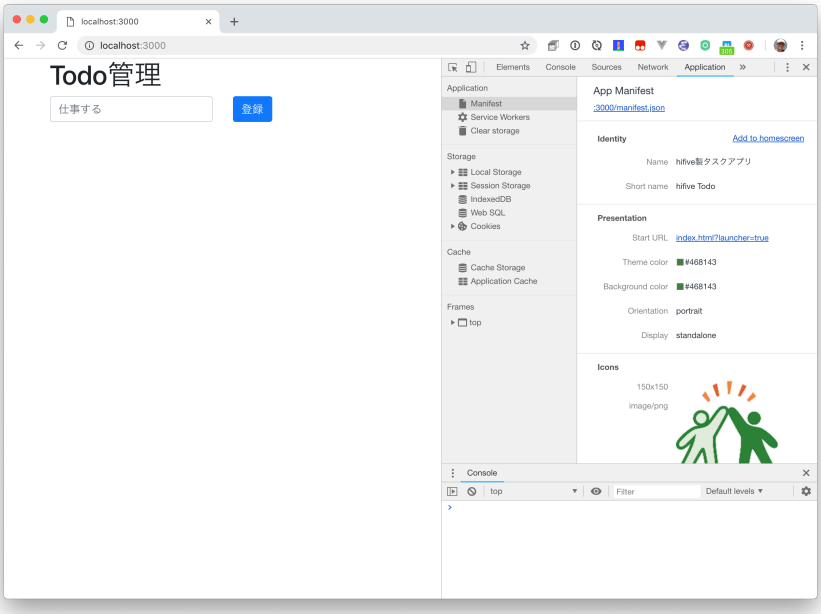
マニフェストファイルの内容

マニフェストファイルは以下のような内容になっています。

キー	内容
short_name	短いアプリ名
name	アプリ名
icons	アイコン。サイズに応じて複数指定可能
display	フルスクリーン、スタンドアローン、Webブラウザなど
background_color	アプリが立ち上がる際の背景色
theme_color	テーマカラー。ヘッダーバーの色の適用
orientation	回転方向
start_url	PWAを表示する際のURL

内容を確認する

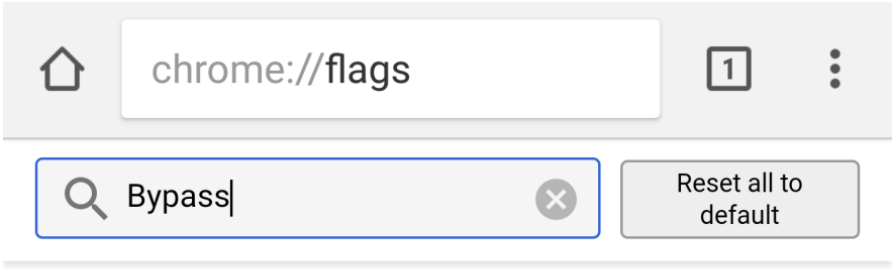
manifest.jsonがきちんと書かれているかどうかは Google Chromeで確認するのが一番簡単です。開発者ツールを開いて、Applicationタブに切り替えます。以下のようにmanifest.jsonファイルの内容が表示されます。

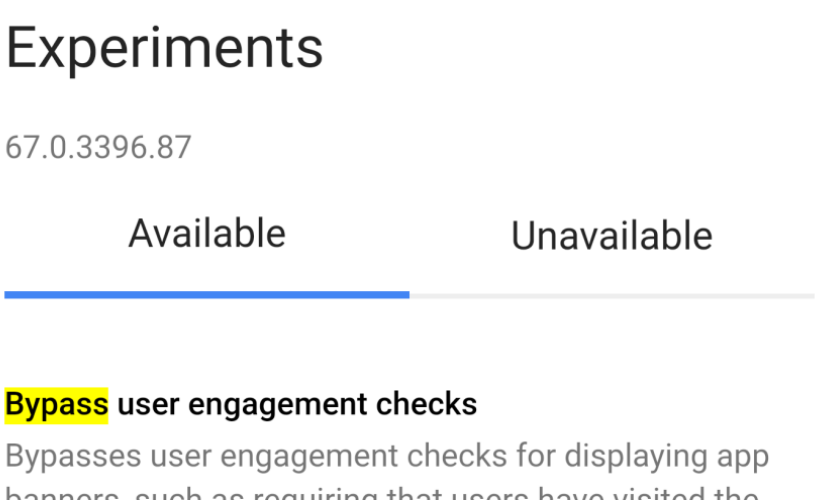


この内容を編集して、Webブラウザで再読込すると表示に反映されます。編集して確認してみましょう。

Androidで設定を変更する

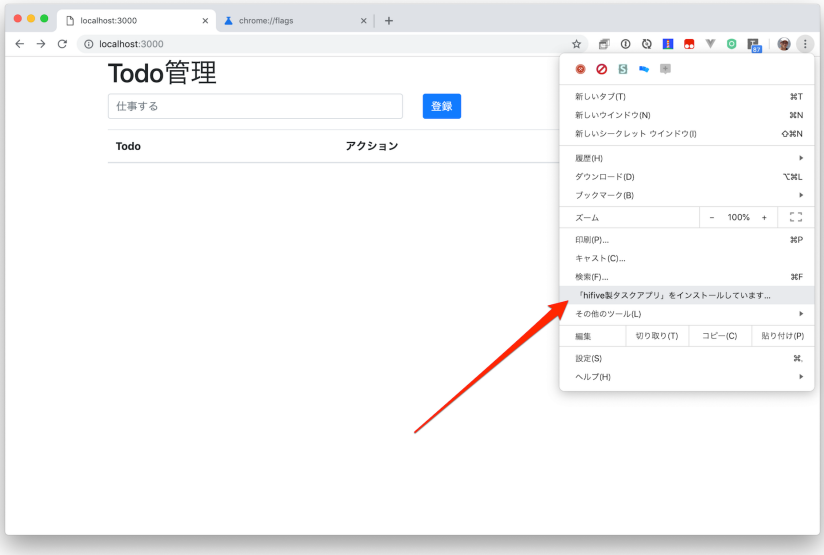
PWAとしてインストールするか確認するバナー（A2H = Add to Home Screen）は5分以上の時間をおいて、2回目以降のアクセスで表示されます。しかし開発中ではこの状態では不便なので、Google Chromeの設定変更をお勧めします。Androidで `chrome://flags` を開きます。そしてBypass user engagement checksと検索して有効にします。



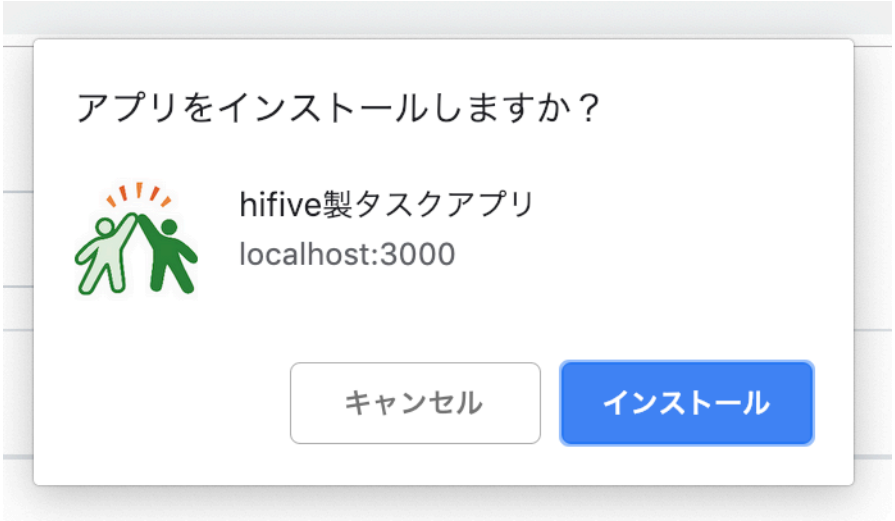


これで一度目のアクセスでインストールバナーが表示されるようになります。

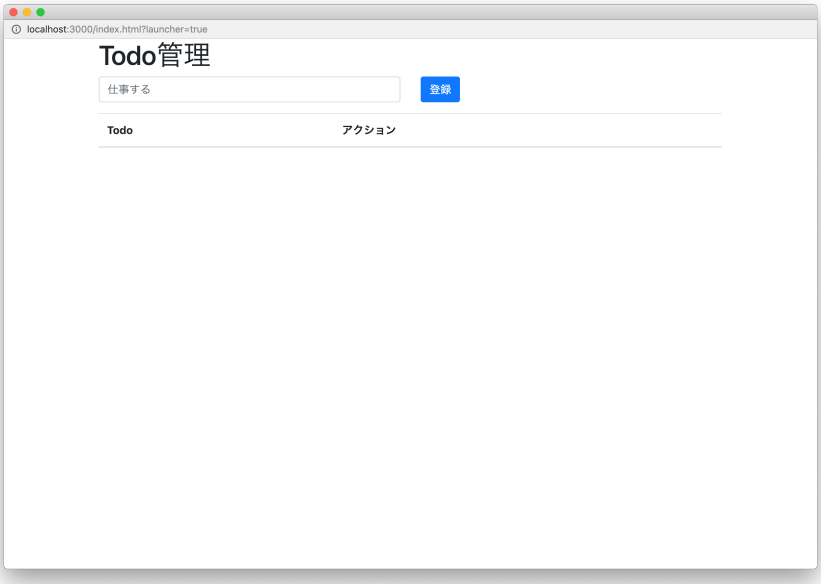
再起動後、メニュー（右上の縦型の三点リーダー）をクリックすると **「hifive製タスクアプリ」をインストール...** と表示が追加されています。これを選びます。



選ぶとインストールを行うダイアログが出ます。インストールを押せば、デスクトップアプリとしてインストールされます。



Windowsの場合はデスクトップに、macOSの場合は `~/Applications/Chrome`アプリの中にインストールされます。ここから立ち上げると、アドレスバーだけが表示され、シンプルなUIでウィンドウが開きます。



Service Workerについて

アプリ化はマニフェストファイルだけでは利用できません。次の第3章 Service Workerのインストールと有効化と第4章 Service Workerを使った表示高速化、オフライン対応についてを行うとアプリとしてインストールできるようになります。では第3章 Service Workerのインストールと有効化を行いましょう。

第3章 SERVICE WORKERのインストールと有効化

ここではTodoアプリにService Workerの組み込みと、その有効化を行います。この章のコードはすでに `www/js/app.js` に記述済みです。ファイルを見て、コードとその内容を確認してください。

Service Workerをインストールする

まずService Workerのインストール法についてです。インストールといってもService Workerは単なるJavaScriptファイルであり、それを別なJavaScriptファイルから読み込むだけです。

現在、 `www` 以下に sw.js というファイルがあります。このファイルがService Workerです。次に `www/js/app.js` からService WorkerのJavaScriptファイルを読み込みます。

Service Workerに対応しているかチェック

まずレガシーなWebブラウザ対策としてService Workerに対応しているかチェックします。対応していない場合は何もしません。

```
// app.jsに記述済み
if ('serviceWorker' in navigator) {
  // Service Worker対応
}
```

`navigator.serviceWorker.register` を使ってService Workerをインストールします。Promiseで返ってきますので、 `then` で処理を繋ぎます。
`registration.onupdatefound` はService Workerのファイルがアップデートされていると実行されるイベントです。ファイルの更新はWebブラウザが自動的に行ってくれますが、オンラインでないといけませんので注意してください。

```
// app.jsに記述済み
navigator.serviceWorker
  .register('/sw.js')
  .then(function(registration) {
    // 登録成功
    registration.onupdatefound = function() {
```

```
      console.log('アップデートがあります！');
    }
  })
  .catch(function(err) {
    // 登録失敗 :(
    console.log('ServiceWorker registration failed: ', err);
});
```

Service Workerがインストールされたらキャッシュ処理を実行

次に **sw.js** を見てみます。Service Workerでの役割の一つがキャッシュです。キャッシュというとWebブラウザ標準で用意されていたキャッシュと混同してしまいますが、CACHE APIは開発者が自由にコントロールできるキャッシュで、本来のURLにアクセスする前に呼ばれるものです。そのためCACHEの内容を改変したり、オンラインとオフラインの状態で表示内容を変えるといったことも自由にできます。

キャッシュする際に必要なのはキャッシュ名と、キャッシュしたいURLのリストです。それらは **sw.js** の一番上に指定してあります。

```
// すでに記述済み
// キャッシュ名（変更可）
const CACHE_NAME = 'YOUR_CACHE_NAME';
// キャッシュするURL
const urlsToCache = [
  '/',
  // : 省略
];
self.skipWaiting();
```

次にService Workerがインストールされると install イベントが実行されます。
event.waitUntil はその中の処理（今回はキャッシュのためのリクエストする処理）が完了するのを保証してくれます。全体として非同期処理で行われるので各処理が確実に終わるのを待つ必要があります。

```
// すでに記述済み
// Service Workerがインストールされた時に呼ばれる処理
self.addEventListener('install', event => {
  // Service Workerがバージョンアップしている時に、新しいものを有効にする処
```

```
  event.waitUntil(self.skipWaiting());
  // キャッシュ登録処理を完了するのを保証する
  event.waitUntil(
    // キャッシュを開く
    caches.open(CACHE_NAME)
      // 指定したURLをキャッシュに登録する
      .then(cache => {
        urlsToCache.map(url => {
          // アクセスして結果を受け取る
          fetch(new Request(url))
            .then(response => {
              // 結果をキャッシュに登録する
              return cache.put(url, response);
            }, err =>  console.log(err));
        });
      })
  );
});
```

注意点

ここで分かるかと思いますが、Service Workerの中でfetchを使ってリクエスト処理を行っています。つまり通常のWebブラウザの中で行うHTTPリクエスト（画像やJavaScriptファイル、HTMLなど）を行う処理とService Workerとのリクエストは「別物」ということです。

そのため、Webブラウザからアクセスする時には特別な情報（ヘッダーなど）を付与しているとService Workerでは別なコンテンツがキャッシュされてしまう可能性があります。

ここまででService Workerのインストールが完了しました。第4章 Service Workerを使った表示高速化、オフライン対応について解説します。

第4章 SERVICE WORKERを使った表示高速化、オフライン対応について

前回、Service Workerを使ってキャッシュ登録を行いました。今回はそのキャッシュを返す処理を作成します。キャッシュを返すことによって二つのメリットが生まれます。

- サーバにアクセスしないので表示が高速
- サーバにアクセスしないのでオフラインでも表示できる

CACHE APIはローカルプロキシと比喩されることがありますが、リモートサーバにアクセスする前にCACHE APIを呼び出すので、まさにプロキシとして利用できます。

JSON BOXにアクセスする

JSON BOXは無料、登録なしで使えるREST APIのデータベースサービスです。

https://jsonbox.io/

アクセスすると専用のURLが生成されますので、コピーして todo.jsの下記部分を書き換えてください。

```
var DOMAIN = 'JSON BoxのURL';
```

キャッシュ処理を確認する

二回目以降のWebブラウザでのアクセスはすべてService Workerの `fetch` イベントを通過します。 `event.respondWith` はHTTPレスポンスを受け取ってWebブラウザに返します。 `event.request` の中にリクエスト情報が入っています。sw.jsの中のfetchイベントを探して、下記のように変更してください。

元：

```
// ネットワークアクセス時に使われるfetchイベント
self.addEventListener('fetch', async (event) => {
});
```

修正後：

```
// ネットワークアクセス時に使われるfetchイベント
self.addEventListener('fetch', async (event) => {
  event.respondWith(
    fetch(event.request)
      .then(response => {
        return response;
      },
      error => {
        return caches.match(event.request);
      })
    )
});
```

ここで注意して欲しいのは fetch イベントはキャッシュに登録したURL以外の場合も呼ばれるということです。そこでキャッシュの中から該当リクエストがあるかどうかを確認します。そのレスポンスがあればそのまま返し、なければFetch APIを使ってHTTPリクエストを行います。

オフラインの場合どうなるか？

オフラインの場合、挙動はどうなるでしょうか。そのURLをキャッシュしている場合、レスポンスがあるのでキャッシュを返します。つまりオフライン対応できます。対してレスポンスがない（キャッシュしていない）場合は Fetch APIを使ってHTTPリクエストを行いますが、オフラインなので当然失敗します。そのためオフラインの場合にはキャッシュがあれば表示され、なければ表示できないという状態になります。当たり前ですが、オフライン時にどういう表示にするかは開発者に委ねられていることになります。

ここまででTodoアプリのオフライン化、表示高速化が実現しました。次にTodoの表示処理をオフライン対応させてみましょう。

第5章 TODOの表示処理をオフライン対応させる

TodoアプリをPWA化させる上で大事なのは以下の機能になるでしょう。

- Todoの表示、登録、削除をオフライン化する

Todoの表示をオフライン化する場合、CACHE APIを使えば簡単に実現できそうな気がします。確かに「一度だけの」キャッシュであれば簡単です。Service Workerはデフォルトではキャッシュの自動更新機能を備えていません。そのため、キャッシュの削除処理を実行しないとキャッシュが更新されません。

そこで今回はTodoの表示をオフライン化させてみます。その際に使えるのが動的なキャッシュ生成です。

動的なキャッシュ生成とは

通常Service Worker側でキャッシュを作成する場合、あらかじめURLを決めておいて、それをまとめて登録します。しかしこの方法ではTodoのように動的に内容が変わるものに対応できません。そこでメッセージ機能を使ってWebブラウザのJavaScript（メインスレッド）からService Worker（ワーカースレッド）を呼び出してキャッシュを更新します。

処理はTodo登録処理に追加する

ではいつタスクを更新するかと言えば、Todoを登録した際が一番良さそうです。現在のTodo一覧と新しく登録したTodoを追加してService Workerに送ります。 todo.js を開いて、 `// Service Workerのキャッシュを更新します（第5章）` と書かれている場所の下にコードを追加します。

Service Workerにメッセージを送る際には次のようにします。これで Service Workerの `message` というイベントが呼ばれます。

修正後：

```
// Service Workerのキャッシュを更新します（第5章）
// todo.jsの中です
const channel = new MessageChannel();
navigator.serviceWorker.controller
  .postMessage({
    url: DOMAIN,
```

```
      todos: me.__todos
    }, [channel.port2]);
```

Service Workerでキャッシュを更新する

キャッシュはURLとそのレスポンスのキー/バリューでしかありませんので、更新は簡単です。sw.jsから `// Todoの一覧を更新する処理（第5章）` と書かれている部分を探して追記します。

```
// Todoの一覧を更新する処理（第5章）
// sw.jsの中です
self.addEventListener('message', function(event) {
  // キャッシュを開く
  caches.open(CACHE_NAME)
    .then(cache => {
      // 新しいレスポンスを作る
      // レスポンスとともに、レスポンスヘッダーも指定
      const res = new Response(JSON.stringify(event.data.todos), {
        headers: {
          'Content-Type': 'application/json'
        }
      });
      // URLとレスポンスを紐付け
      cache.put(event.data.url, res);
    })
});
```

これでTodoを登録した際のキャッシュ変更が完了します。

Todo削除時にも同様の処理を行う

Todo削除の際にも `updateView` を呼び出しますので、この時にキャッシュを更新します。

キャッシュを動的に更新することでWeb APIを使ったような動的なコンテンツにも対応できます。では次はTodoの投稿処理をオフライン対応します。

第6章 TODOの投稿処理をオフライン対応

Todo表示処理をオフライン化しましたが、Todo投稿や削除処理はオフラインの状態では失敗してしまいます。これには二つの問題があります。

- オフライン時には投稿内容をキューに保存する
- オンラインになったタイミングでキューを実行する

オフライン時には投稿内容をキューに保存する

Webブラウザのオンライン、オフライン判定を行うのは `navigator.onLine` になります。trueの場合はオンライン、falseの場合はオフラインになります。キューをグローバルな変数として用意しておき、オフライン時にはその変数に保存しましょう。ただし変数に入れたままだとWebブラウザを再読込した時に消えてしまいます。そこでlocalStorageを使ってデータを恒久的に残しておきます。 `todo.js` で `// オフライン時の処理（第6章で追加）` を探して追記してください。

```
// オフライン時の処理（第6章で追加）
if (!navigator.onLine) {
  // オフライン時の処理
  var task = {
    _id: '_local_' + Math.random().toString(32).substring(2),
    todo: todo
  };
  queues.add.push(task);
  localStorage.setItem('addQueue', JSON.stringify(queues.add));
  // 表示を更新
  return res(task);
}
```

逆にWebブラウザを開いた時にはキューを復元します。これは todo.js の一番上に記載されています。

```
// すでに記載済み
// オフライン時に追加/削除するTodo入れておくキュー
var queues = {add: [], delete: []};
```

```
  for (var name of ['add', 'delete']) {
    var value = localStorage.getItem(name + 'Queue');
    if (value) {
      queues[name] = JSON.parse(value);
    }
  }
```

これでキューに保存、そして復元する処理が完了です。

削除処理も同様に

Todoを削除した際にも同様の処理が必要です（こちらはすでに記載済みです）。

```
  // オフライン時
  // すでに記載済み
  if (!navigator.onLine) {
    queues.delete.push(todo);
    localStorage.setItem('deleteQueue', JSON.stringify(queues.delete))
    return res({todo});
  }
```

これで削除するTodoもキューに追加されました。

オンラインになった時にキューを処理する

ではWebブラウザがオンラインに戻った時にキューを処理しましょう。これは `document.addEventListener` で実行できます。hifiveの場合は `'{window} online'` と'定義します。 `todo.js` で オンライン復帰時の処理（第6章用） を探して追記してください。

```
  // オンライン復帰時の処理（第6章用）
  '{window} online': function(context) {
    // 以下を追加
    this.executeQueue(queues);
  }
```

処理は `addTodo` と `deleteTodo` を呼ぶだけです。すでに記述済みです。

```
  // 以下はすでに追加済みです。
  // Todo追加、削除の実行
  executeTask: function(action, todo) {
    switch (action) {
    case 'add':
      this.addTodo(todo);
      break;
    case 'delete':
      this.deleteTodo(todo);
      break;
    }
  },
  // キューを処理する
  executeQueue: function(queues) {
    var me = this;
    for (var action of ['add', 'delete']) {
      for (var todo of queues[action]) {
        this.executeTask(action, todo);
      }
      queues[action] = [];
      localStorage.setItem(action + 'Queue', []);
    }
  },
```

Todoの削除も同様です。今回はTodo名しか使っていないので、同じラベルで複数作ってしまうとまとめて削除されてしまいますので注意してください。

これでオフライン対応も完了です。

ここまででPWAとしての基本であるオフライン対応が完了します。リソースの取得はService Workerがうまく行ってくれますが、データの追加/更新/削除については自分たちでうまく実装しなければなりません。また、Web APIの場合はデータが動的に変わるのでキャッシュを最適なタイミングで更新しないと古いデータが表示されてしまう可能性があるので注意してください。

では次回はこのTodoアプリにWebリモートプッシュ通知を実装してみましょう。

第7章 WEBリモートプッシュ通知を実装する

WebリモートプッシュはWebブラウザベンダーにロックインされたものと、そうでないVAPIDがあります。簡単に比較すると次のようになります。

	Google Chrome	Firefox & Edge	iOS Safari	macOS Safari
ベンダー実装	○	×	×	○
VAPID	○	○	×	×

iOSはいずれのWebリモートプッシュ通知にも対応していません。iOS13以降に期待になるでしょう。macOS SafariではSafari Developerとして登録して証明書を作成する必要があります。そうした手間を考えると、現状ではVAPIDプッシュ通知だけ対応しておけば十分でしょう。以下の操作はデスクトップのGoogle ChromeやFirefox、EdgeそしてAndroidのGoogle Chromeだけが利用できます。

ライブラリについて

Node.jsでWebPushを行う場合には `web-push` を使います。このライブラリはあらかじめインストールされています。

鍵の生成

プッシュ通知は公開鍵/非公開鍵を用いて行います。そのための鍵を生成する処理は以下のコマンドを実行します。

```
npm run keygen
```

これは keygen.js を実行しています。keygen.jsはVAPID用のキーを生成して、`application-server-keys.json` というファイルを作成します。

JavaScriptの修正

Webブラウザで読み込んでいた app.js を app.push.js に変更します（index.html内の記述）。これはWebPushに関する記述がすでにされています。内容についてはコメントを参考にしてください。

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

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

処理としては以下のようになっています。

- Service Workerのインストール
- プッシュ通知のサポート状況確認
- プッシュ通知が拒否されていないか確認
- 購読情報の取得
- 購読データの作成

App.push.jsの修正

application-server-key.jsonに書かれている **publicKey** の値を app.push.js の KEY の値に置き換えてください。例えば **application-server-key.json** の内容が以下であったとします。

```
{
  "publicKey": "BLr...bG-Gc",
  "privateKey":"di..5E"
}
```

この場合、 **BLr...bG-Gc** をコピーします。そして **app.push.js** を開きます。その中で下記の記載を探します。

```
// KEYを生成した公開鍵に書き換えます
const key = 'KEY';
```

この key の値を書き換えます。今回の例では以下のようになります。

```
// KEYを生成した公開鍵に書き換えます
const key = 'BLr...bG-Gc';
```

Service Workerで必要な記述（記述済み）

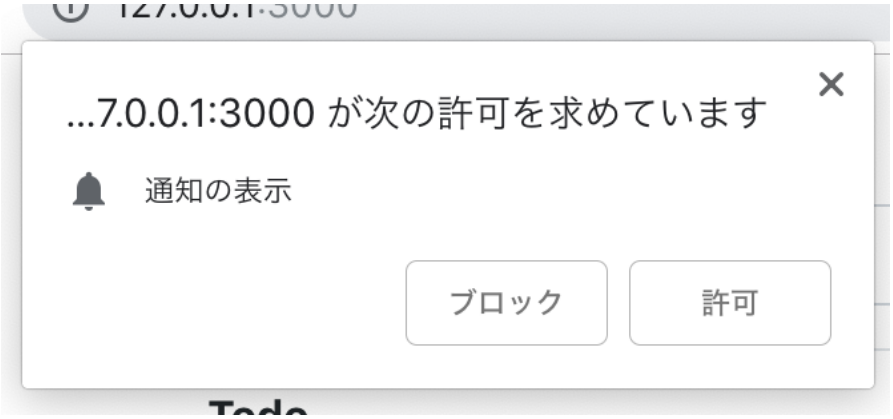
WebPush通知を受け取るとService Worker **sw.js** の **push** イベントが呼ばれます。
WebPushではペイロード（独自データ）が送信できます。そして

`self.registration.showNotification` でプッシュ通知を表示します。sw.js内に以下の処理を追加します。これは受け取ったWebプッシュ通知の内容を表示する処理です。

```
// 以下は記述済みです
// Webプッシュ通知の処理（第7章）
self.addEventListener('push', ev => {
  // payloadの取得
  const {title, msg, icon} = ev.data.json();
  self.registration.showNotification(title, {
    icon: icon,
    body: msg
  });
});
```

HTMLを読み込む

Webページをリロードするとプッシュ通知の購読確認が表示されますので、購読を開始してください。



そうすると購読データが表示されますので、これをクリップボードにコピーしてください（一度目はnullが出ますので再読込してください）。通常、このデータをデータベースに保存します。

プッシュ通知を送信する

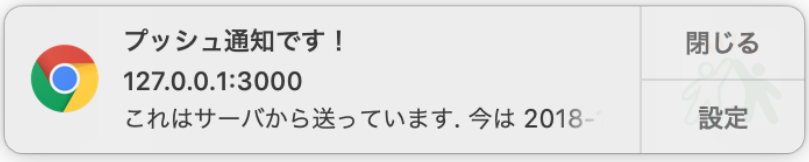
先ほどコピーした内容を `token.json` として保存してください。場所は server.js のあるディレクトリです。

```
touch token.json
```

保存したら以下のコマンドを実行します。

```
npm run push
```

うまくいけばプッシュ通知が表示されるはずです。このコマンドは push.js を実行しています。



ここまでの処理でWebPush通知の実装は完了です。

PWAを体験するハンズオンは以上で終了となります。

- アプリ化
- オフライン化
- Webプッシュ通知

の3つを体験してもらいました。他にも認証を容易にするWebAuthnであったり、決済を容易にするPayment Requestなどもあります。より使いやすいWebページのためにPWAを導入していきましょう。

WORKBOXを使った実践的CACHE APIの使い方

WorkboxはオープンソースのCACHE API用ライブラリです。細かく条件を指定してキャッシュを行えます。この章では、Workboxを使ってすでに設定されているキャッシュの設定を変更してみましょう。

Workboxのインストール

Workboxはsw.jsに次のように記述してインストールできます。次の内容をsw.jsの一番最初の行に記述してください。

```
importScripts('https://storage.googleapis.com/workbox-cdn/releases/4
```

Workboxを試す

Workboxでキャッシュを行うため、`install` イベントで実行されているキャッシュ処理を削除します。

```
// 空にします
self.addEventListener('install', event => {
});
```

例えば `http://127.0.0.1:8887` 以下のアクセスをすべてキャッシュする場合は次のように記述します。

```
workbox.routing.registerRoute(
  /127.0.0.1:8887/,
  workbox.strategies.staleWhileRevalidate()
);
```

キャッシュ戦略の違い

Workboxでは複数のキャッシュ戦略を用意しています。

- workbox.strategies.staleWhileRevalidate()
- workbox.strategies.CacheOnly()
- workbox.strategies.CacheFirst()
- workbox.strategies.NetworkFirst()
- workbox.strategies.NetworkOnly()

`workbox.strategies.staleWhileRevalidate()` は最初にキャッシュを返し、もしなければネットワークアクセスしてくれるモードです。一番よく使われる形式になります。

細かくカスタマイズする

例えばスタイルシートだけをキャッシュしたいならば、次のように記述します。

```
workbox.routing.registerRoute(
  /\.css$/,
  workbox.strategies.staleWhileRevalidate()
);
```

有効期限を指定する

キャッシュに有効期限を設けられます。以下は1週間のキャッシュになります。

```
workbox.routing.registerRoute(
  /\.js$/,
  new workbox.strategies.CacheFirst({
    cacheName: 'js-cache',
    plugins: [
      new workbox.expiration.Plugin({
        maxAgeSeconds: 7 * 24 * 60 * 60,
        maxEntries: 10,
      }),
    ]
  })
);
```