НЕТОЛОГИЯ

WEB WORKERS, SERVICE WORKERS

МИХАИЛ КУЗНЕЦОВ

МИХАИЛ КУЗНЕЦОВ

Разработчик в ING Bank

@mkuznetcov

ПЛАН ЗАНЯТИЯ

1. Web Workers
2. Service Workers

ЗАДАЧА

Перед нами поставили следующую задачу: реализовать загрузку достаточно большого файла формата CSV с расходами в определённых категориях, который нужно обработать и найти сумму расходов.

В качестве примера возьмём сэмпл от IBM.

СЕРВЕР

Можно, конечно, отправлять всё на сервер и там всё рассчитывать, но для этого нужно:

1. Написать сервер;
2. Где-то его разместить.

Нельзя ли просто построить пользовательское приложение, которое потребует только статичных ресурсов и никуда ничего не будет отправлять?

РЕАЛИЗАЦИЯ

```
<button>freeze</button>
<input type="file" data-id="file-input" accept="text/csv">
<p data-id="result-text"></p>
```

```
function readFile(file) {
  return new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.addEventListener('load', (evt) => {
      resolve(evt.target.result);
    });
    reader.addEventListener('error', (evt) => {
      reject(evt.target.error);
    });

    reader.readAsText(file);
  });
}
```

ЧТЕНИЕ CSV

Остался вопрос с чтением CSV. Несмотря на то, что CSV это обычный текстовый формат, мы вас предостерегаем от написания вручную парсеров любых форматов данных (только если в этом не заключается ваша задача).

Воспользуемся для этого готовым пакетом csv-parse:

```
npm install csv-parse
```

ЧТЕНИЕ CSV

```
const fileEl = document.querySelector('[data-id=file-input]');
const resultEl = document.querySelector('[data-id=result-text]');
fileEl.addEventListener('change', async (evt) => {
  const content = await readFile(evt.currentTarget.files[0]);
  const records = parse(content, {
    columns: true,
    skip_empty_lines: true,
    skip_lines_with_empty_values: true,
  });
  // С целью сокращения кода посчитаем просто кол-во строк
  resultEl.textContent = records.length;
  console.log(records.length);
});
```

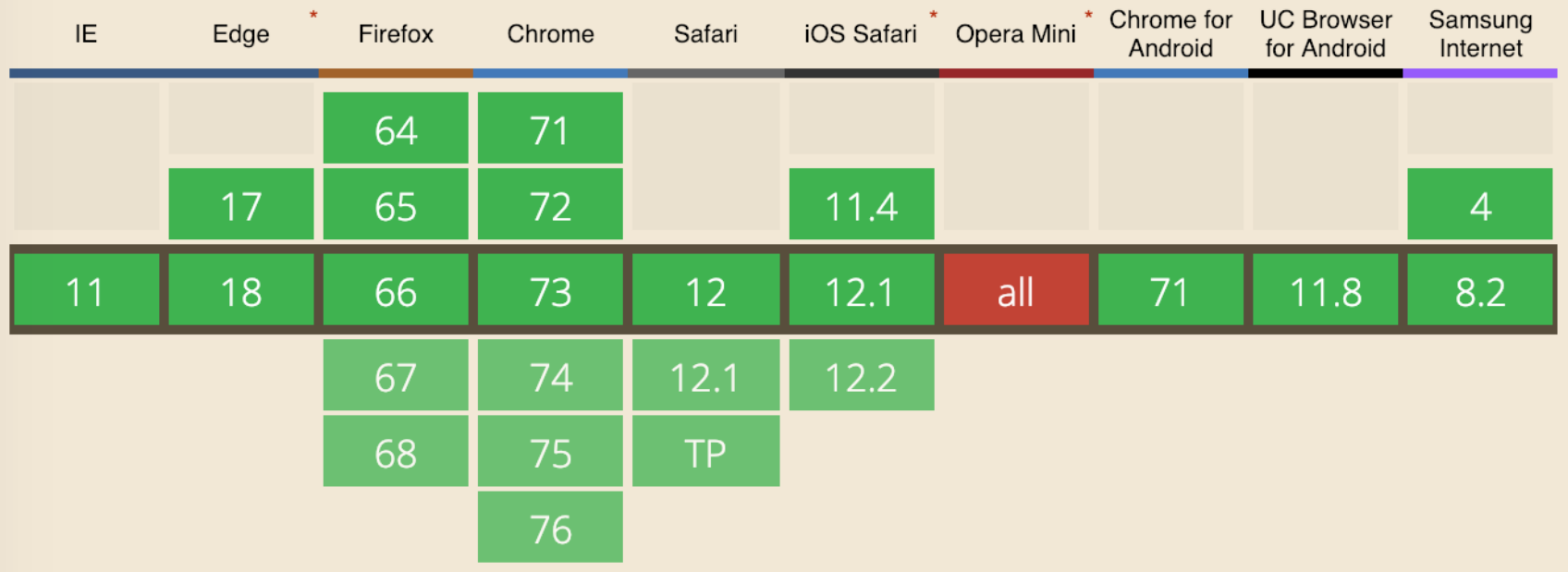
Видно, что на время «обработки файла» интерфейс «замораживается» и кнопка «Freeze» не реагирует на нажатия.

WEB WORKERS

WEB WORKERS

Web Workers - API, позволяющие исполнять скрипты в фоновом режиме независимо от скриптов, работающих с пользовательским интерфейсом.

ПОДДЕРЖКА


IE
Edge *
Firefox
Chrome
Safari
iOS Safari *
Opera Mini *
Chrome for Android
UC Browser for Android
Samsung Internet
64
71
17
65
72
11.4
4
11
18
66
73
12
12.1
all
71
11.8
8.2
67
74
12.1
12.2
68
75
TP
76

WEB WORKERS

Ключевая область применения - выполнение ресурсоёмких/длительных задач, которые сложно выполнять в основном потоке без влияния на скорость отрисовки пользовательского интерфейса, например:

- обработка изображений;
- сортировка/обработка/получение данных.

ОГРАНИЧЕНИЯ

Web Worker’ы не имеют доступа к `window`, `document` и DOM, но при этом имеют доступ к:

- таймаутам и интервалам;
- XHR/fetch;
- возможности создавать другие Worker’ы.

ПРИМЕР WORKER'А

`index.html`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
  </head>
  <body>
    <script src="js/app.js"></script>
  </body>
</html>
```

ПРИМЕР WORKER'А

`app.js`:

```
1  const worker = new Worker('web.worker.js');
2  worker.addEventListener('message', (event) => {
3    console.log(event);
4  });
5  worker.addEventListener('error', (event) => {
6    console.error(event);
7  });
8  worker.postMessage('hello from main');
```

`web.worker.js`:

```
1  self.addEventListener('message', (event) => {
2    console.log(event);
3  })
4
5  setInterval(() => {
6    self.postMessage('worker is alive');
7  }, 1000);
```

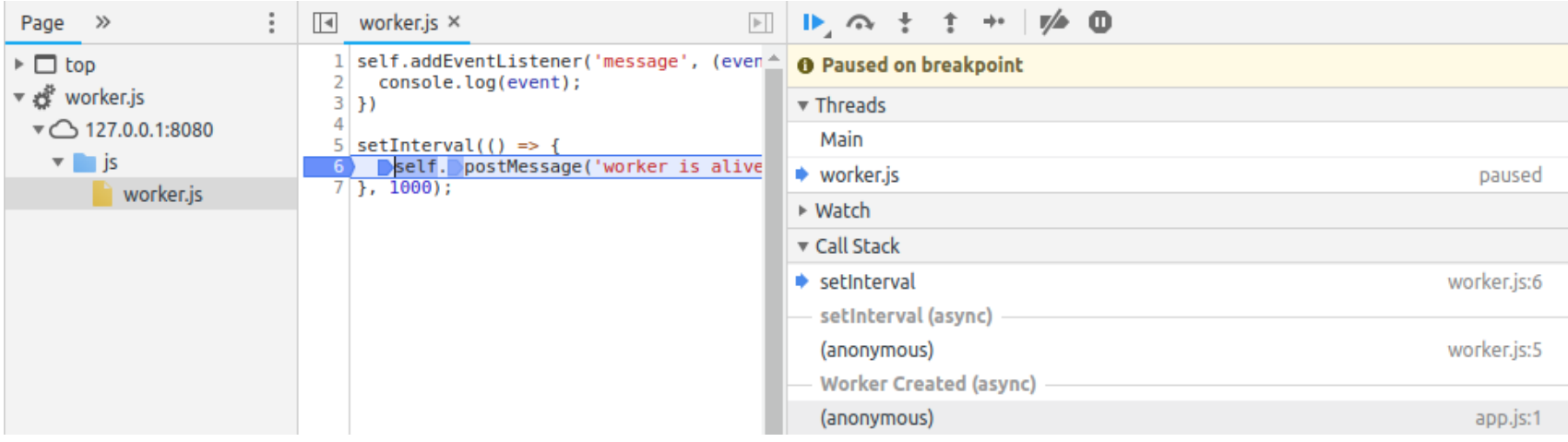
MODULES

Если вы не используете бандлеры, а используете ESM, то Worker можно интерпретироваь как модуль:

```
const worker = new Worker('web.worker.mjs', { type: "module" });
```

ПРИМЕР РАБОТЫ WORKER'A

Мы увидим, что будут отображаться два потока: `Main` и `worker.js`:



ЗАГРУЗКА ДРУГИХ СКРИПТОВ

Для загрузки других скриптов нам предоставляется метод `importScripts`, который загружает и исполняет скрипты в контексте Web Worker'а (либо можно просто воспользоваться бандлером).

GLOBAL

`globalThis` в Worker'е выглядит вот так:

```
[Exposed=Worker] // <- обратите внимание
interface WorkerGlobalScope : EventTarget {
  readonly attribute WorkerGlobalScope self;
  readonly attribute WorkerLocation location;
  readonly attribute WorkerNavigator navigator;
  void importScripts(USVString... urls);

  attribute OnErrorEventHandler onerror;
  attribute EventHandler onlanguagechange;
  attribute EventHandler onoffline;
  attribute EventHandler ononline;
  attribute EventHandler onrejectionhandled;
  attribute EventHandler onunhandledrejection;
};
```

globalThis

```
console.log(globalThis);
// DedicatedWorkerGlobalScope
```

`self`

`self` - это поле в `globalThis`, которое позволяет отправлять сообщения, подписываться на события, и завершать собственную работу с помощью метода `close`.

DedicatedWorkerGlobalScope

```
interface DedicatedWorkerGlobalScope : WorkerGlobalScope {
  [Replaceable] readonly attribute DOMString name;

  void postMessage(any message, sequence<object> transfer);
  void postMessage(any message, optional PostMessageOptions options);

  void close();

  attribute EventHandler onmessage;
  attribute EventHandler onmessageerror;
};
```

ИНФРАСТРУКТУРА

Но у нас же ESLint, Babel и, самое главное, Webpack, который собирает всё в один бандл.

Как интегрировать Web Worker’ы в нашу инфраструктуру?

ESLINT

Для работы с ESLint нужно поставить env:

```
"env": {
  "es6": true,
  "browser": true,
  "jest": true,
  "worker": true
},
```

`self`

К сожалению, `self` до сих пор находится в списке `restricted-globals`, поэтому нам придётся вручную его удалить.

```
npm install --save-dev eslint-restricted-globals
```

self

Для этого конфиг-файл переименуем в `.eslintrc.js` и настроим:

```
const standardRestrictedGlobals = require('eslint-restricted-globals');
const noRestrictedGlobals = ["error", "isNaN", "isFinite"].concat(
  standardRestrictedGlobals
);
const noRestrictedGlobalsWorker = noRestrictedGlobals.filter(o => o !== 'self');

module.exports = {
  ...
  "rules": {
    ...
    "no-restricted-globals": noRestrictedGlobals,
  },
  "overrides": [
    {
      "files": ["*.worker.js"],
      "rules": {
        "no-restricted-globals": noRestrictedGlobalsWorker
      }
    }
  ],
```

WEBPACK

```
npm install --save-dev worker-loader
```

```
{
  test: /web.worker\.js$/,
  use: { loader: 'worker-loader' }
}
```

`app.js`:

```
import Worker from './web.worker';
```

КАК РАБОТАЕТ WORKER

Синхронно исполняется сверху вниз и:

- завершает исполнение, если не зарегистрировано `onmessage` или eventListener'а на `message`;
- если хотя бы один зарегистрирован, то ожидает получения сообщений до вызова `terminate`.

УПРАВЛЕНИЕ WORKER'ОМ

Worker автоматически стартует при создании и продолжает своё выполнение до тех пор пока внутри себя не выполнит все инструкции (см. предыдущий слайд), либо не получит от своего создателя команду `terminate`.

Внутри себя Worker может завершить собственное выполнение, вызвав `self.close()`.

ПЕРЕДАЧА ДАННЫХ

Данные передаются в качестве аргумента метода `postMessage` в обе стороны.

РЕАЛИЗАЦИЯ

`app.js`:

```
import Worker from './web.worker';

function readFile(file) {
  ...
}

const fileEl = document.querySelector('[data-id=file-input]');
const resultEl = document.querySelector('[data-id=result-text]');
fileEl.addEventListener('change', async (evt) => {
  const content = await readFile(evt.currentTarget.files[0]);

  const worker = new Worker();
  worker.addEventListener('message', ({ data: result }) => {
    resultEl.textContent = result;
    console.log(result);
    worker.terminate();
  });
  worker.postMessage(content);
});
```

РЕАЛИЗАЦИЯ

`web.worker.js`:

```
1  import parse from 'csv-parse/lib/sync';
2
3  self.addEventListener('message', (evt) => {
4    const records = parse(evt.data, {
5      columns: true,
6      skip_empty_lines: true,
7      skip_lines_with_empty_values: true
8    });
9    self.postMessage(records.length);
10 });
```

Видно, что теперь интерфейс не «замораживается», а обработка происходит в параллельном потоке.

STRUCTURED CLONING

При передаче данных используется алгоритм «Structured Clone», который хоть и является достаточно быстрым, всё равно по факту является созданием копии.

При этот поддерживается копирование не всех объектов (MDN), но для самых нужных поддерживается:

- Все примитивы;
- `String`;
- `Blob`;
- `File`;
- `FileList`;
- `ArrayBuffer`;
- `ArrayBufferView`;
- `ImageData`;
- `Array`;
- `Object`;
- `Map`;
- `Set`.

РЕАЛИЗАЦИЯ

`app.js`:

```
fileEl.addEventListener('change', async (evt) => {
  const file = evt.currentTarget.files[0];
  const worker = new Worker();
  worker.addEventListener('message', ({ data: result }) => {
    resultEl.textContent = result;
    worker.terminate();
  });
  worker.postMessage(file);
});
```

`web.worker.js`:

```
self.addEventListener('message', async ({ data: file }) => {
  const content = await readFile(file);
  const records = parse(content, { columns: true, skip_empty_lines: true, skip_lines_with_empty_values: true });
  self.postMessage(records.length);
});
```

ПЕРЕДАЧА ДАННЫХ

По умолчанию, все данные, передаваемые в Worker, клонируются. Это очень неэффективно при передаче `ArrayBuffer`, поэтому при передаче через postMessage, можно вторым аргументом указать `buffer`, который должен передаваться, а не клонироваться:

```
worker.postMessage({
  value: buffer // an ArrayBuffer object
}, [buffer]);

// without object:
worker.postMessage(buffer, [buffer]);
```

Точно так же при отправке обратно.

TRANSFERABLE OBJECTS

Важно отметить следующую деталь: при такой отправке объект становится недоступным в том контексте, из которого его отправили. Т.е. буквально происходит перенос объекта, а не его клонирование, что оказывается гораздо эффективнее.

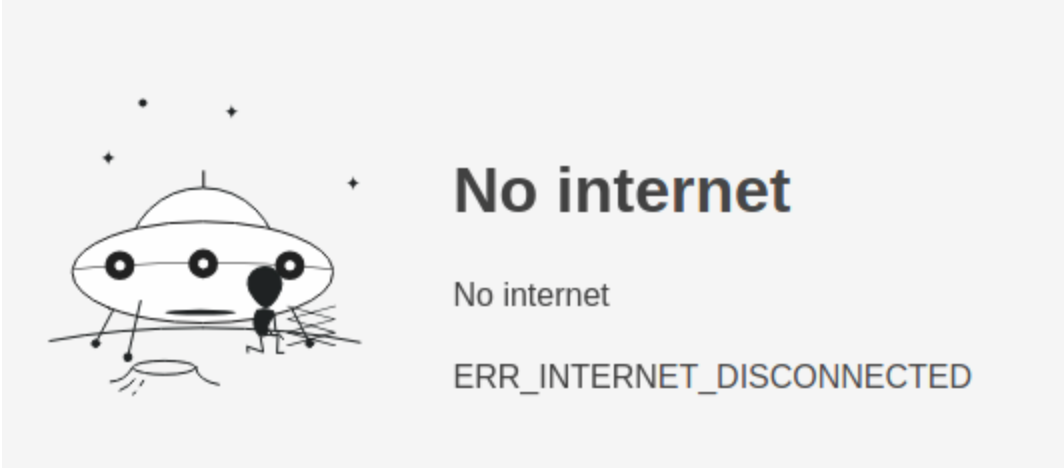
Важно: работает это только для `ArrayBuffer`, т.е. нам придётся прочитать файл в `ArrayBuffer`, отправить его в Worker, потом перевести в строку (т.к. csv-parser работает только со строками).

Поэтому здесь стоит искать компромисс между удобством и скоростью/производительностью.

SERVICE WORKERS

ЗАДАЧА

Нам поставили задачу сделать приложение, которое работает в оффлайн режиме, например, так, как умеет Gmail - при перезагрузке страницы мы видим сообщение о пропавшем интернете:



На базе этого можно разработать приложение, позволяющее (пусть даже в ограниченном режиме):

1. Письма (личные сообщения доступны), но не все, а только последние;
2. Поиск по ним (по последним сообщениям) работает.

LOCALSTORAGE

Когда мы обсуждали работу с формами, мы говорили, что вообще говоря, можно «кэшировать» данные в LocalStorage и это будет работать до тех пор, пока пользователь не нажмёт на кнопку «Обновить» (мы, конечно, можем зарегистрировать eventListener'а на событие beforeunload и попросить пользователя не уходить, но это так себе решение).

Посмотрим на альтернативы.

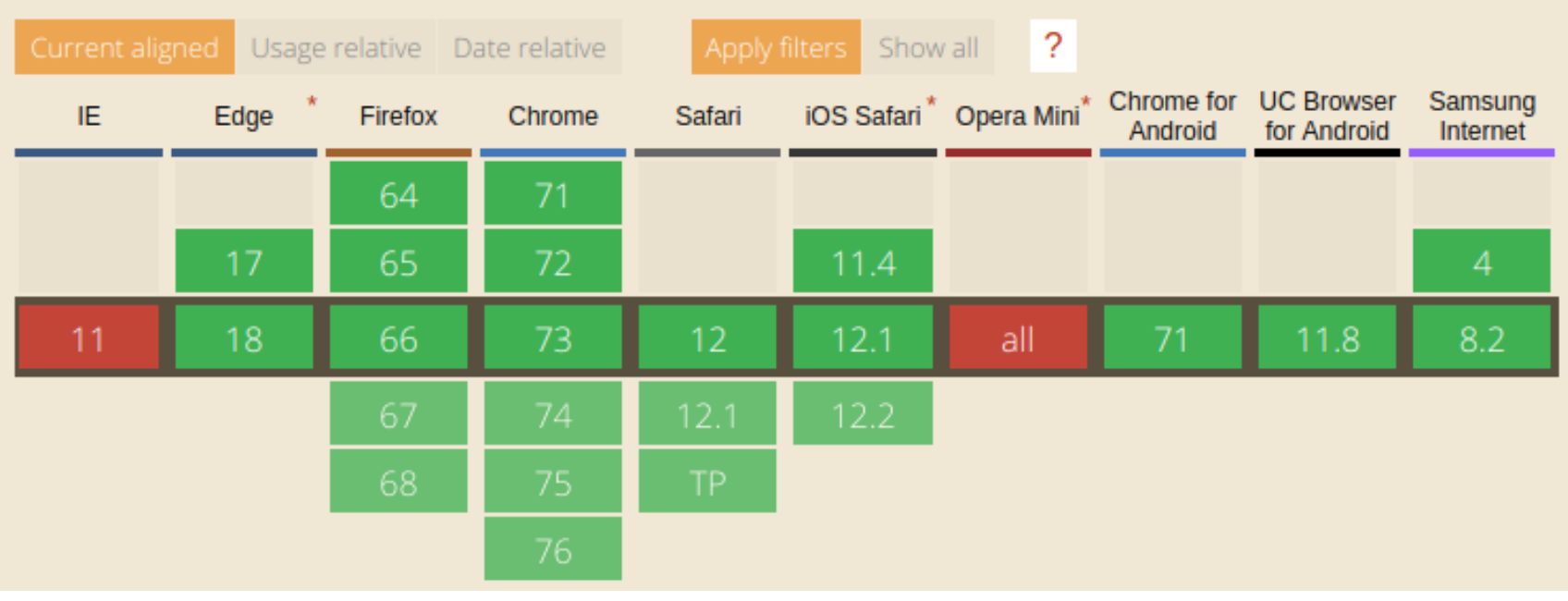
SERVICE WORKERS

Service Worker - event-driven Web Worker, который может отвечать на события, генерируемые из документа и других источников.

Если говорить упрощённо, то Service Worker чаще всего выступает неким прокси между приложением и сетью (кэшем), позволяя приложению получать ответы на запросы даже в случае отсутствия сети (прозрачно для самого приложения).

Примечание*: на самом деле умеют они не только это, но мы сосредоточимся именно на этом.

ПОДДЕРЖКА


Current aligned
Usage relative
Date relative
Apply filters
Show all
?
IE
Edge *
Firefox
Chrome
Safari
iOS Safari *
Opera Mini *
Chrome for Android
UC Browser for Android
Samsung Internet
64
71
17
65
72
11.4
4
11
18
66
73
12
12.1
all
71
11.8
8.2
67
74
12.1
12.2
68
75
TP
76

ESLINT

Для работы с ESLint нужно поставить env:

```
"env": {
  "es6": true,
  "browser": true,
  "jest": true,
  "worker": true,
  "serviceworker": true
},
```

HTTPS / LOCALHOST

Service Worker'ы работают только по HTTPS. По HTTP они могут работать только с `localhost` (для целей разработки и отладки).

СОЗДАНИЕ SERVICE WORKER'A

Начнём с самого простого, зарегистрируем Service Worker, который будет кэшировать статичные файлы и отдавать их на запросы браузера даже в случае отсутствия соединения. Пока поучимся работать без Webpack'а, а потом «докрутим» и Webpack.

Разметка:

```
<div>
  <img data-id="image">
</div>
<script src="js/app.js"></script>
```

РЕГИСТРАЦИЯ

Так же, как и Web Worker, Service Worker должен располагаться в отдельном файле* и мы должны его зарегистрировать:

```
if (navigator.serviceWorker) {
  window.addEventListener('load', async () => {
    try {
      await navigator.serviceWorker.register(
        '/service.worker.js', { scope: './' }
      );
    } catch (e) {
      console.log(e);
    }
  });
}
```

Важная деталь: Service Worker - это Worker, но не Web Worker.

navigator.serviceWorker

```
interface ServiceWorkerContainer : EventTarget {
  readonly attribute ServiceWorker? controller;
  readonly attribute Promise<ServiceWorkerRegistration> ready;

  [NewObject] Promise<ServiceWorkerRegistration> register(
    USVString scriptURL,
    optional RegistrationOptions options
  );
  ...
};

dictionary RegistrationOptions {
  USVString scope;
  WorkerType type = "classic";  // module (как в Web Workers)
  ServiceWorkerUpdateViaCache updateViaCache = "imports";
};
```

РЕГИСТРАЦИЯ

Регистрация производится методом `register` с указанием скрипта, из которого создаётся Service Worker и `scope`.

`scope` позволяет указать, с каких путей мы можем перехватывать запросы. По умолчанию устанавливается путь, по которому расположен сам Service Worker (в нашем случае - `/`).

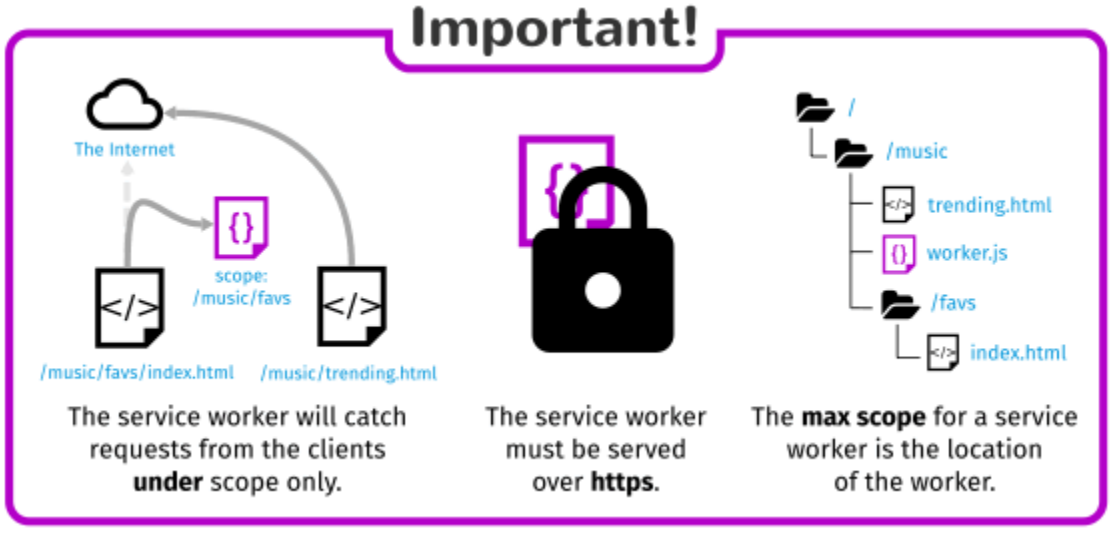
`scope`

Именно работа со `scope` вызывает больше всего проблем:

1. Вы можете перехватывать только запросы от клиентов у которых `scope` такой же, либо ниже;
2. Максимальный доступный `scope` - это расположение файла Service Worker.

Под клиентом понимается конкретная HTML-страница, Worker и т.д., но мы для упрощения будем рассматривать только HTML-страницы.

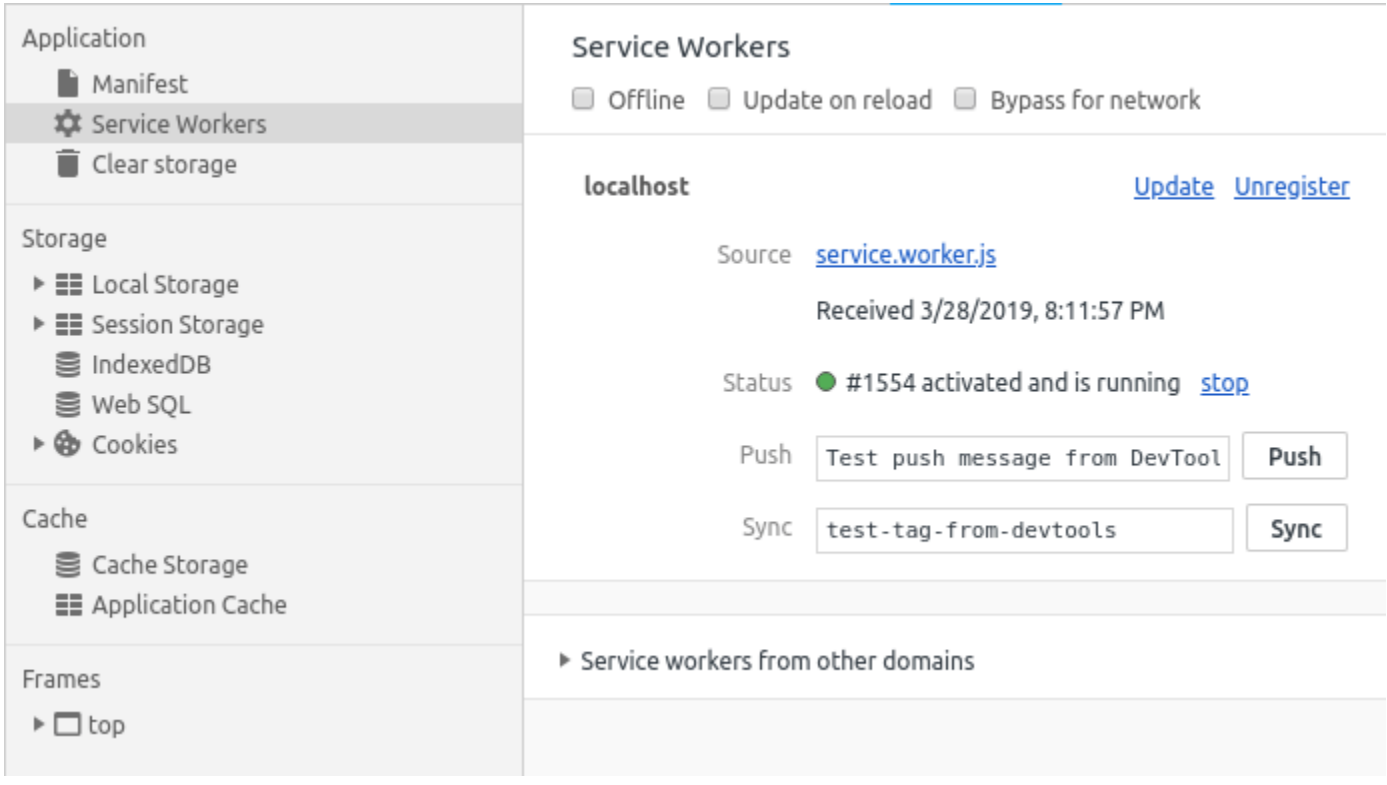
SCOPE MDN



Copyright MDN

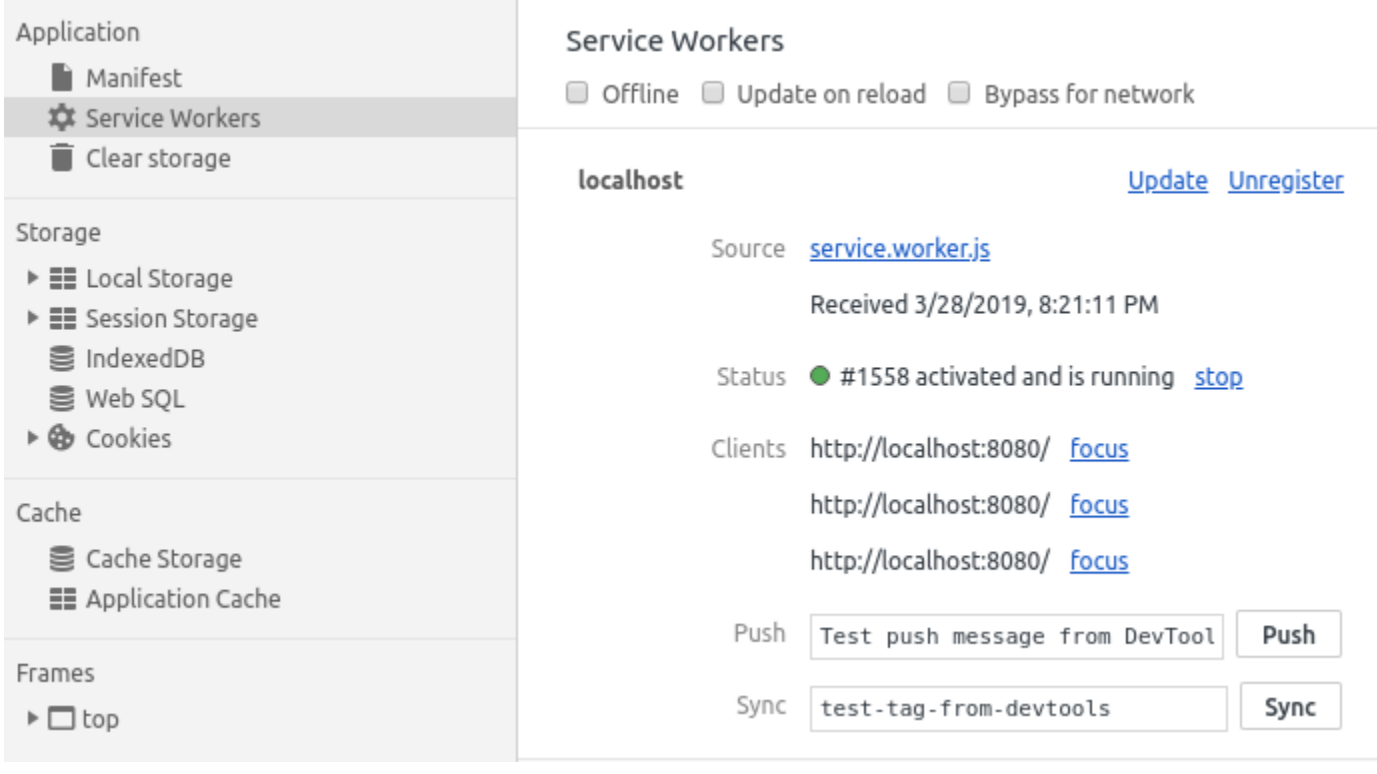
В большинстве случаев, чтобы избежать сложностей со `scope`, файл Service Worker'а кладут в корень, таким образом, получая в `scope` всех клиентов.

SERVICE WORKER



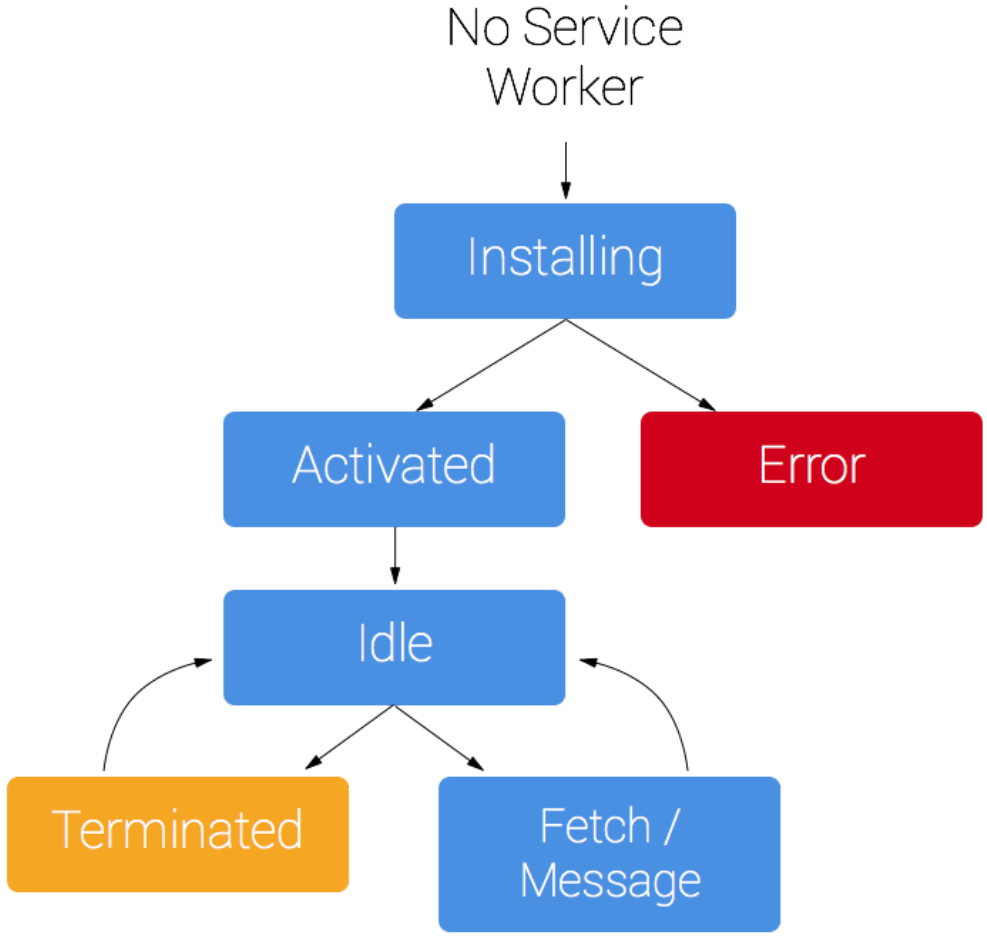
CLIENTS

Увидеть подключенных клиентов можно в том же окне (если их больше одного):



ЖИЗНЕННЫЙ ЦИКЛ

У Service Worker достаточно интересный жизненный цикл:



ЖИЗНЕННЫЙ ЦИКЛ

По факту жизненный цикл состоит из следующих этапов:

1. Регистрация (из клиента, скачивание + обработка скрипта);
2. Установка (точка, в которой можно сделать `skitWaiting` и кэшировать нужные URL'ы);
3. Активация (точка, в которой можно передать контроль текущему Service Worker у всех клиентов и удалить старые кэши);
4. Terminated (браузер периодически будет уничтожать наш Service Worker);
5. Fetch (обработка событий `fetch`).

РЕГИСТРАЦИЯ

Этап регистрации мы уже разобрали, достаточно вызвать:

```
await navigator.serviceWorker.register(
  '/service.worker.js', { scope: './' }
);
```

УСТАНОВКА

Установка достаточно сложный этап, происходит он следующим образом: браузер вызовает eventListener на событие `install`, но вызывает он его один раз. Т.е. сколько бы мы не обновляли страницу, `install` вызовется только один раз.

Но при этом, если мы изменим сам файл Service Worker'а (хотя бы байт), браузер заново загрузит и сделает `install` для Service Worker'а.

АКТИВАЦИЯ

Активация происходит после установки.

Но есть важный нюанс: каждый раз, как вы вносите в код Service Worker'а изменения и обновляете страницу (либо она сама обновляется через Hot/Live Reload), новая версия автоматически не активируется, а находится в режиме `waiting to activate` и не заменит предыдущую версию до тех пор, пока предыдущая версия обслуживает хотя бы одного клиента.

ВСЁ СЛОЖНО

На самом деле, мы можем отменить это поведение по умолчанию, в `install` вызвав `self.skipWaiting()`, а `activate` - `self.clients.claim()`:

```
self.addEventListener('install', (evt) => {
  console.log(evt);
  evt.waitUntil(self.skipWaiting());
});


self.addEventListener('activate', (evt) => {
  console.log(evt);
  evt.waitUntil(clients.claim());
});
```

`evt.waitUntil` - вызов, ждущий разрешения `Promise` для использования его в качестве результата установки или активации. Если `Promise` перейдёт в состояние `fullfiled`, то установка/активация считается успешной.

ServiceWorkerGlobalScope

```
[Global=(Worker,ServiceWorker), Exposed=ServiceWorker]
interface ServiceWorkerGlobalScope : WorkerGlobalScope {
  [SameObject] readonly attribute Clients clients;
  [SameObject] readonly attribute ServiceWorkerRegistration registration;

  [NewObject] Promise<void> skipWaiting();

  attribute EventHandler oninstall;
  attribute EventHandler onactivate;
  attribute EventHandler onfetch;

  // event
  attribute EventHandler onmessage; // event.source of the message events is Client object

  attribute EventHandler onmessageerror;
  };
```

`skipWaiting` - попытка активации клиента.

clients

Где `clients` - интерфейс, содержащий информацию о клиентах:

```
[Exposed=ServiceWorker]
interface Clients {
  // The objects returned will be new instances every time
  [NewObject] Promise<any> get(DOMString id);
  [NewObject] Promise<FrozenArray>Client>> matchAll(optional ClientQueryOptions options);
  [NewObject] Promise<WindowClient?> openWindow(USVString url);
  [NewObject] Promise<void> claim();
};
```

`claim` - «обновление» Service Worker'а для всех клиентов.

CACHE API

Cache API позволяет вам использовать хранилице CacheStorage для хранения объектов. Помещать объекты и извлекать их из хранилища необходимо автоматически.

`self.caches`:

```
partial interface mixin WindowOrWorkerGlobalScope {
  [SecureContext, SameObject] readonly attribute CacheStorage caches;
};
```

CacheStorage

```
[SecureContext, Exposed=(Window,Worker)]
interface CacheStorage {
  [NewObject] Promise<any> match(RequestInfo request, optional MultiCacheQueryOptions options);
  [NewObject] Promise<boolean> has(DOMString cacheName);
  [NewObject] Promise<Cache> open(DOMString cacheName);
  [NewObject] Promise<boolean> delete(DOMString cacheName);
  [NewObject] Promise<sequence<DOMString>> keys();
};
```

Ключевые методы:

- `open` - открываем кэш для дальнейшей работы;
- `keys` - получаем имена существующих кэшей;
- `delete` - удаляет кэш по имени.

Cache

```
[SecureContext, Exposed=(Window,Worker)]
interface Cache {
  [NewObject] Promise<any> match(RequestInfo request, optional CacheQueryOptions options);
  [NewObject] Promise<FrozenArray<Response>> matchAll(optional RequestInfo request, optional CacheQueryOptions options);
  [NewObject] Promise<void> add(RequestInfo request);
  [NewObject] Promise<void> addAll(sequence<RequestInfo> requests);
  [NewObject] Promise<void> put(RequestInfo request, Response response);
  [NewObject] Promise<boolean> delete(RequestInfo request, optional CacheQueryOptions options);
  [NewObject] Promise<FrozenArray<Request>> keys(optional RequestInfo request, optional CacheQueryOptions options);
};
```

Ключевые методы:

- `addAll` - добавляем объекты `Response`, соответствующие каждому `RequestInfo` (браузер сам делает HTTP-запросы);
- `match` - ищем объект, соответствующий `RequestInfo`;
- `put` - кладём в кэш объект Response, соответствующий `RequestInfo`;
- `delete` - удаляем объект, соответствующий `RequestInfo`.

СОБИРАЕМ ВСЁ ВМЕСТЕ

```
const version = 'v1';
const cacheName = `ahj-${version}`;

const files = [
  '/',
  '/img/js.png',
  '/js/app.js',
];

async function putFilesToCache(files) {
  const cache = await caches.open(cacheName);
  await cache.addAll(files);
}

async function removeOldCache(retain) {
  const keys = await caches.keys();
  return Promise.all(
      keys.filter(key => !retain.includes(key))
      .map(key => caches.delete(key))
  );
}
```

СОБИРАЕМ ВСЁ ВМЕСТЕ

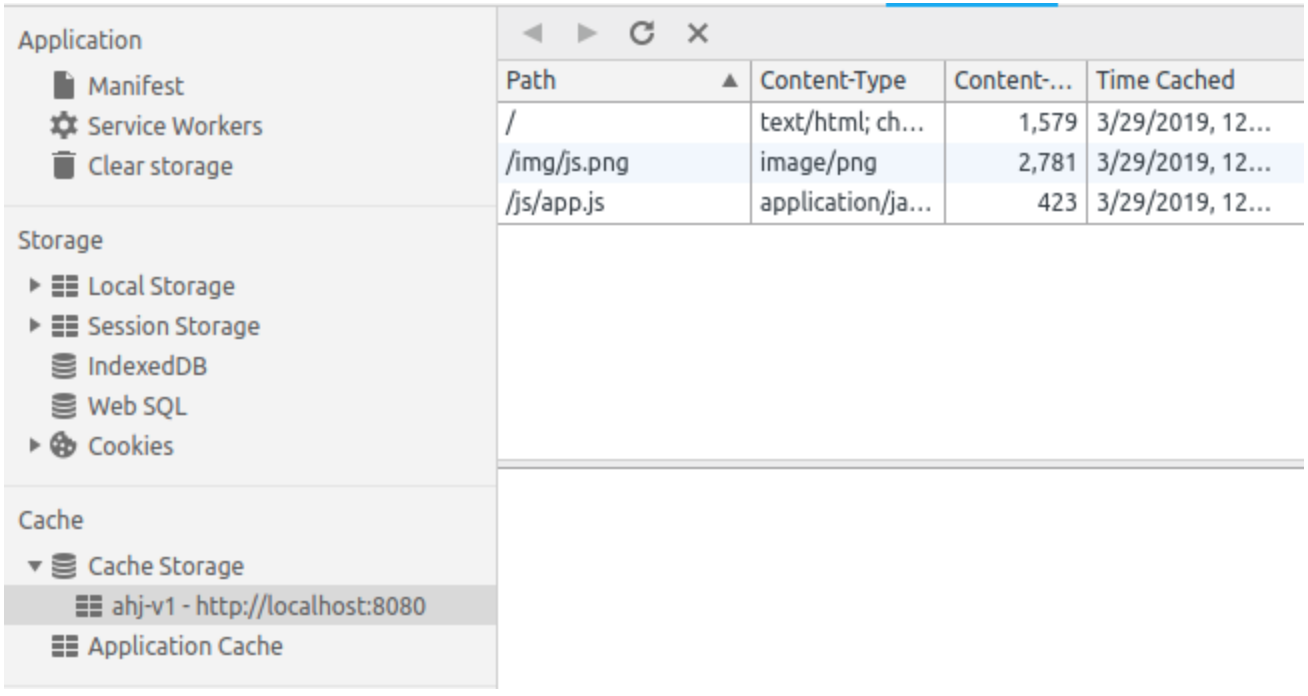
```
self.addEventListener('install', (evt) => {
  console.log(evt);
  evt.waitUntil((async () => {
    await putFilesToCache(files);
    await self.skipWaiting();
  })());
});


self.addEventListener('activate', (evt) => {
  console.log(evt);
  evt.waitUntil((async () => {
    await removeOldCache([cacheName])
    await self.clients.claim();
  })());
});
```

`skipWaiting`

Со `skipWaiting` нужно быть достаточно осторожным, т.к. использование этого метода приводит к тому, что «старый клиент» обрабатывался одним Service Worker'ом, а сейчас стал обрабатываться новым, который не учитывает некоторых «старых запросов».

CACHE STORAGE

В итоге получим:



Важно: обязательно удостоверьтесь, что при обновлении Service Worker'а, старый кэш удаляется (попробуйте подставить `const version = 'v2'`).

CACHE STORAGE LIMIT

Лимиты на Cache Storage находятся в районе 50 Мб, в том числе для мобильных браузеров.

fetch

Остался последний этап - перехватывать запрос и отдавать ответ из кэша:

```
self.addEventListener('fetch', (evt) => {
  evt.respondWith((async () => {
    const cache = await caches.open(cacheName);
    const cachedResponse = await cache.match(evt.request);

    if (cachedResponse) {
      return cachedResponse;
    }

    return fetch(evt.request);
  })());
});
```

`evt.respondWith` - «подменяем» ответ: если он есть в кэше, то отдаём из кэша, если его нет, то делаем запрос: `fetch(evt.request)`.

FETCH EVENT

```
[Constructor(DOMString type, FetchEventInit eventInitDict), Exposed=ServiceWorker]
interface FetchEvent : ExtendableEvent {
  [SameObject] readonly attribute Request request;
  readonly attribute Promise<any> preloadResponse;
  readonly attribute DOMString clientId;
  readonly attribute DOMString resultingClientId;
  readonly attribute DOMString replacesClientId;

  void respondWith(Promise<Response> r);
};
```

OFFLINE

Теперь, даже если мы поставим флажок `Offline`, мы всё равно после перезагрузки будем получать страницу и все ресурсы, прописанные в нашем кэше:

```
if (navigator.serviceWorker) {
  window.addEventListener('load', async () => {
    try {
      await navigator.serviceWorker.register(
        '/service.worker.js', { scope: './' }
      );
    } catch (e) {
      console.log(e);
    }

    setTimeout(() => {
      document.querySelector('[data-id=image]').src = 'img/js.png';
    }, 5000);
  });
}
```

Но остался следующий вопрос: со статичными ресурсами всё хорошо, как кэшировать динамические запросы? Ведь мы заранее не знаем URL ресурса.

FETCH VS SCRIPT

Мы можем добавлять их в кэш как в Service Worker, так и в наших скриптах, т.к. Cache API доступно и там:

```
[SecureContext, Exposed=(Window,Worker)]
interface CacheStorage { ... }
```

FETCH VS SCRIPT

Рассмотрим вариант с Service Worker:

```
self.addEventListener('fetch', (evt) => {
  const requestUrl = new URL(evt.request.url);

  if (!requestUrl.pathname.startsWith('/api')) {
    return;
  }

  evt.respondWith((async () => {
    const cache = await caches.open(cacheName);

    try {
      const response = await fetch(evt.request);
      evt.waitUntil(cache.put(evt.request, response.clone()));
      return response;
    } catch (e) {
      const cachedResponse = await cache.match(evt.request);
      if (cachedResponse) {
        return cachedResponse;
      }
    }
    throw new Error('no cached data');
  })());
});
```

А дальше уже логика приложения - как получать список последних сообщений/писем и складывать их в кэш.

`clone`

```
evt.waitUntil(cache.put(evt.request, response.clone()));
```

`clone` - позволяет создать копию объекта `Response`, т.к. Response является потоком и его можно прочитать только один раз.

STRATEGIES

Мы можем придумать достаточно много различных стратегий, комбинируя Cache API и fetch, включая динамическую подгрузку тех же самых сообщений.

Поскольку Service Worker - это Worker, то мы можем отправлять ему в том числе и сообщения через `postMessage`, для этого у нас есть `navigator.serviceWorker.controller` - активный Service Worker, контролирующий данного клиента.

```
if (navigator.serviceWorker.controller) {
  navigator.serviceWorker.controller.postMessage(...);
}
```

EVENTS

Ключевая вещь, которую нам нужно знать про Service Worker: это event-driven Worker. Он «живёт» обработкой событий и браузер сам решает, когда «уничтожить» Worker.

Поэтому ни в коем случае не должны ориентироваться на данные вне обработчиков событий и хранилищ, предоставляемых браузером (Cache API).

WEBPACK

Отлично, мы рассмотрели очень кратко часть функциональности Service Worker. Давайте посмотрим, как это привязать к Webpack.

Самый простой способ - использовать file-loader, который просто переместит наш файл в каталог `dist`:

```
{
  test: /service.worker\.js$/,
  loaders: [
    {
      loader: 'file-loader',
      options: {
        name: '[name].[ext]',
      },
    }
  ]
},
```

Но давайте посмотрим другие опции.

WORKBOX PLUGIN

Workbox Plugin предоставляет не только API для «облегчённой» регистрации Service Worker'а, но и готовые стратегии, интеграцию с Google Analytics и т.д.

```
npm install --save-dev workbox-webpack-plugin
```

У данного плагина существует два режима:

1. `GenerateSW` - для простых сценариев (кэш ресурсов);
2. `InjectManifest` - для более сложных, когда нужна собственная логика.

WORKBOX PLUGIN

Задача первая - обработка статики, генерируемой Webpack в результате сборки. Нам достаточно будет `GenerateSW`:

```
const WorkboxPlugin = require('workbox-webpack-plugin');

plugins: [
  ...,
  new WorkboxPlugin.GenerateSW({
    clientsClaim: true,
    skipWaiting: true,
    cleanupOutdatedCaches: true,
  }),
]
```

При этом писать самостоятельно никакого Service Worker'а не нужно.

РЕГИСТРАЦИЯ

Но регистрация всё равно потребуется (файл `app.js`):

```
if (navigator.serviceWorker) {
  window.addEventListener('load', async () => {
    try {
      if (navigator.serviceWorker) {
        await navigator.serviceWorker.register(
          '/service.worker.js'
        );
        console.log('sw registered');
      }
      // await registration.unregister();
    } catch (e) {
      console.log(e);
    }
  });
}
```

WEBPACK

В результате сборки мы получим файлы:

- `precache-manifest<checksum>.js` - список файлов для кэширования;
- `service.worker.js` - файл, подключающий библиотеку workbox и осуществляющий кэширование.

CUSTOM WORKER

Для данного варианта режим `GenerateSW` уже не подойдёт, нужно использовать `InjectManifest`:

```
const WorkboxPlugin = require('workbox-webpack-plugin');

plugins: [
  ...,
  new WorkboxPlugin.InjectManifest({
    swSrc: './src/service.worker.js',
    swDest: 'service.worker.js',
  }),
]
```

CUSTOM WORKER

```
workbox.core.skipWaiting();
workbox.core.clientsClaim();

const { strategies } = workbox;

self.addEventListener('fetch', (evt) => {
  const url = new URL(evt.request.url);
  if (url.pathname.startsWith('/api')) {
    const cacheFirst = new strategies.NetworkFirst();
    evt.respondWith(cacheFirst.makeRequest({ request: evt.request }));
  }
});

workbox.precaching.precacheAndRoute(self.__precacheManifest);
```

Workbox Plugin сам подставит список статичных ресурсов и ссылку на дистрибутив Workbox.

К сожалению, придётся отключить ESLint для нашего Service Worker'а.

ИТОГИ

Сегодня у нас была достаточно большая лекция:

- мы обсудили работу с Web Worker’ами;
- очень кратко рассмотрели Service Worker’ы.

НЕТОЛОГИЯ

Ваши вопросы?

МИХАИЛ КУЗНЕЦОВ

@mkuznetcov