

Join MDN and developers like you at Mozilla's View Source conference, 12-14 September in Berlin, Germany. Learn more at <https://viewsourceconf.org>

使用 Service Workers

 这篇翻译不完整。请帮忙从英语[翻译这篇文章](#)。

 这是一个实验中的功能


此功能某些浏览器尚在开发中，请参考[浏览器兼容性表格](#)以得到在不同浏览器中适合使用的前缀。由于该功能对应的标准文档可能被重新修订，所以在未来版本的浏览器中该功能的语法和行为可能随之改变。

本文提供了使用 **service workers** 所需要的相关知识。包括它的基本结构、注册一个 **service worker**、一个新的 **service worker** 的安装和激活流程、更新你的 **service worker**、缓存管理和自定义响应内容。所有这些功能点都是基于一个场景：离线APP。

Service Workers出现的背景

有一个困扰web用户多年的难题：网络不可连接（离线）。即使是世界上最好的web app，如果你下载不了它，用户体验基本是毁了。已经有很多种技术尝试，来解决这一问题。随着[离线](#)页面的出现，一些问题已经得到了解决。但是，最重要的问题是，仍然没有一个好的统筹机制，来对缓存和网络请求进行控制。

之前的尝试 — APPCache — 看起来是个不错的方法，因为它可以很容易地指定需要离线缓存的资源。但是，这个方法假定了你使用时会遵循很多规则，如果你不严格遵循这些规则，它会把你的APP搞得一团糟。关于APPCache的更多详情，请看Jake Archibald的文章：[Application Cache is a Douchebag](#)。

 注意：从Firefox44起，当使用AppCache来提供离线页面支持时，会提示一个警告消息，来建议开发者使用 [Service workers](#) 来实现离线页面。([bug 1204581](#).)

Service workers应该最终解决了这些问题。Service Worker的语法比AppCache更加复杂，但带来的效果是你可以使用JavaScript，更加灵活和细粒度地控制你的应用的缓存资源。有了它，你可以解决目前离线应用的问题，同时也可以做更多的事。使用Service Worker可以使你的应用先访问本地缓存，所以在离线状态时，在没有通过网络接收到更多的数据前，仍可以提供基本的功能体验（一般称之为[Offline First](#)）。这是原生APP本来就支持的功能，这也是相比于 web app，原生app更受青睐主要原因。

使用Service Workers前的设置

在已经支持service workers的浏览器的较新版本中，很多service workers的特性默认没有开启支持。如果你发现示例代码在代当前版本的浏览器中怎么样都无法正常运行，你可能需要开启一下浏览器的相关配置：

- **Firefox Nightly:** 访问 `about:config` 并设置 `dom.serviceWorkers.enabled` 的值为 `true`; 重启浏览器;
- **Chrome Canary:** 访问 `chrome://flags` 并开启 `experimental-web-platform-features`; 重启浏览器 (注意: 有些特性在Chrome中没有默认开房支持);
- **Opera:** 访问 `opera://flags` 并开启 `ServiceWorker` 的支持; 重启浏览器。

另外, 你需要通过HTTPS来访问你的页面代码 — 出于安全原因, `Service Workers`严格要求要在HTTPS下才能运行。Github是个用来测试的好地方, 因为它就支持HTTPS。

基本架构


使用service workers, 通常遵循以下基本步骤:



1. service worker, 通过`serviceWorkerContainer.register()`来加载和注册 (一个脚本URL)。
2. 如果注册成功, service worker 在`ServiceWorkerGlobalScope`环境中运行; 这是一个特殊的worker上下文运行环境, 与主脚本的运行线程相独立, 同时也没有访问DOM的能力。
3. service worker现在可以处理事件了。
4. 受service worker控制的页面打开后, service worker尝试安装。最先发送给service worker的事件, 是安装事件(`install event` 在这个事件里, 可以开始IndexDB和Cache的相关操作流程)。这个流程同原生APP或者Firefox OS APP是一样的 — 让所有资源可离线访问。
5. 当`oninstall`事件的处理流程执行完毕后, 可以认为service worker安装完成了。
6. 下一步是激活。当service worker安装完成后, 会接收到一个激活事件(`activate event`)。激活事件的处理函数中, 主要操作是清理旧版本的service worker脚本中使用资源。
7. Service Worker 现在可以控制页面了, 但是只针对在成功注册(`register()`)了service worker后打开的页面。也就是说, 页面打开时有没有service worker, 决定了接下来页面的生命周期内受不受service worker控制。所以, 只有当页面刷新后, 之前不受service worker控制的页面才有可能被控制起来。

Worker lifecycle

INSTALLING

This stage marks the beginning of registration. It's intended to allow to setup worker-specific resources such as offline caches.

 **install**

-  Use **event.waitUntil()** passing a promise to extend the installing stage until the promise is resolved.
-  Use **self.skipWaiting()** anytime before activation to skip installed stage and directly jump to activating stage without waiting for currently controlled clients to close.

REGISTRATION



Installing



Waiting



Active

INSTALLED

The service worker has finished its setup and it's waiting for clients using other service workers to be closed.

REGISTRATION



Installing



Waiting





Active

ACTIVATING

There are no clients controlled by other workers. This stage is intended to allow the worker to finish the setup or clean other worker's related resources like removing old caches.

 **activate**

-  Use **event.waitUntil()** passing a promise to extend the activating stage until the promise is resolved.
-  Use **self.clients.claim()** in the activate handler to start controlling all open clients without reloading them.

REGISTRATION



Installing



Waiting



Active

ACTIVATED

The service worker can now handle functional events.

REGISTRATION



Installing



Waiting



Active

REDUNDANT

This service worker is being replaced by another one.

REGISTRATION



Installing

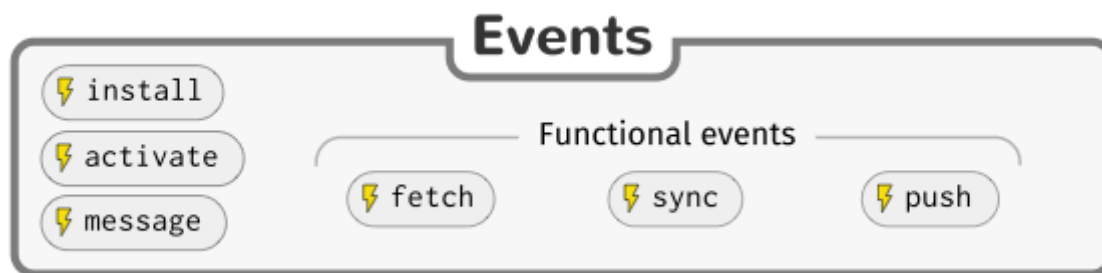


Waiting



Active

下图表示了service worker所有支持的事件：



Promises

Promises 是一种非常适用于异步操作的机制，一个操作依赖于另一个操作的成功执行。这是service worker的核心工作机制。

Promises 可以做很多事情。但现在，你只需要知道，如果有什么返回了一个promise，你可以在后面加上 `.then()` 来传入成功和失败的回调函数。或者，你可以在后面加上 `.catch()` 如果你想添加一个操作失败的回调函数。

接下来，让我们对比一下传统的同步回调结构，和异步promise结构，两者在功能上是等效的：

同步

```
1 try {  
2   var value = myFunction();  
3   console.log(value);  
4 } catch(err) {  
5   console.log(err);  
6 }
```

异步

```
1 myFunction().then(function(value) {  
2   console.log(value);  
3 }).catch(function(err) {  
4   console.log(err);  
5 });
```

在上面第一个例子中，我们必须等待 `myFunction()` 执行完成，并返回 `value` 值，在此之前，后续其它的代码无法执行。在第二个例子中，`myFunction()` 返回一个promise对象，下面的代码可以继续执行。当promise成功resolves后，`then()` 中的函数会异步地执行。

现在来举下实际的例子 — 如果我们想动态地加载图片，而且要在图片下载完成后再展示到页面上，要怎么实现呢？这是一个比较常见的场景，但是实现起来会有点麻烦。我们可以使用 `.onload` 事件处理程序，来实现图片的加载完成后展示。但是如果图片的 `onload` 事件发生在我们监听这个事件之前呢？我们可以使用 `.complete` 来解决这个问题，但是仍然不够简洁，如果是多个图片该怎么处理呢？并且，这种方法仍然是同步的操作，会阻塞主线程。

相比于以上方法，我们可以使用 promise 来实现。(可以看我们的 [Promises test](#) 示例源码, 也可以看这个在线示例: [look at it running live.](#))

 **Note:** service worker在实际使用中，会使用 `caching` 和 `onfetch` 等异步操作，而不是使用老旧的 `XMLHttpRequest` API。这里的例子使用 `XMLHttpRequest` API只是为了让你能将注意力集中于理解 `Promise` 上。

```
1 function imgLoad(url) {
2   return new Promise(function(resolve, reject) {
3     var request = new XMLHttpRequest();
4     request.open('GET', url);
5     request.responseType = 'blob';
6
7     request.onload = function() {
8       if (request.status == 200) {
9         resolve(request.response);
10      } else {
11        reject(Error('Image didn\'t load successfully; error code:' + request.status));
12      }
13    };
14
15    request.onerror = function() {
16      reject(Error('There was a network error.'));
17    };
18
19    request.send();
20  });
21 }
```

我们使用 `Promise()` 构造函数返回了一个新的`promise`对象，构造函数接收一个回调函数作为参数。这个回调函数包含两个参数，第一个为成功执行(`resolve`)的回调函数，第二个为执行失败(`reject`)的回调函数。我们将这两个回调函数在对应的时机执行。在这个例子中，`resolve`会在请求返回状态码200的时候执行，`reject`会在请求返回码为非200的时候执行。上面代码的其余部分基本都是XHR的相关操作，现在不需要过多关注。

当我们调用 `imgLoad()` 函数时，传入要加载的图片url作为参数。然后，后面的代码与同步方式会有点不同：

```
1 var body = document.querySelector('body');
2 var myImage = new Image();
3
4 imgLoad('myLittleVader.jpg').then(function(response) {
5   var imageURL = window.URL.createObjectURL(response);
6   myImage.src = imageURL;
7   body.appendChild(myImage);
8 }, function(Error) {
9   console.log(Error);
10 });
```

在函数调用后面，我们串联了 `promise` 的 `then()` 方法。`then()` 接受两个函数——第一个函数在 `promise` 成功执行的情况下执行，而第二个函数则在 `promise` 执行失败情况下执行。当执行成功时，在 `myImage` 中显示图片，并追加

到 `body` 里面(它的参数就是传递给 `promies` 的 `resolve` 方法的 `request.response`)；当执行失败是，在控制台返回一个错误。

On to the end of the function call, we chain the promise `then()` method, which contains two functions — the first one is executed when the promise successfully resolves, and the second is called when the promise is rejected. In the resolved case, we display the image inside `myImage` and append it to the body (it's argument is the `request.response` contained inside the promise's `resolve` method); in the rejected case we return an error to the console.

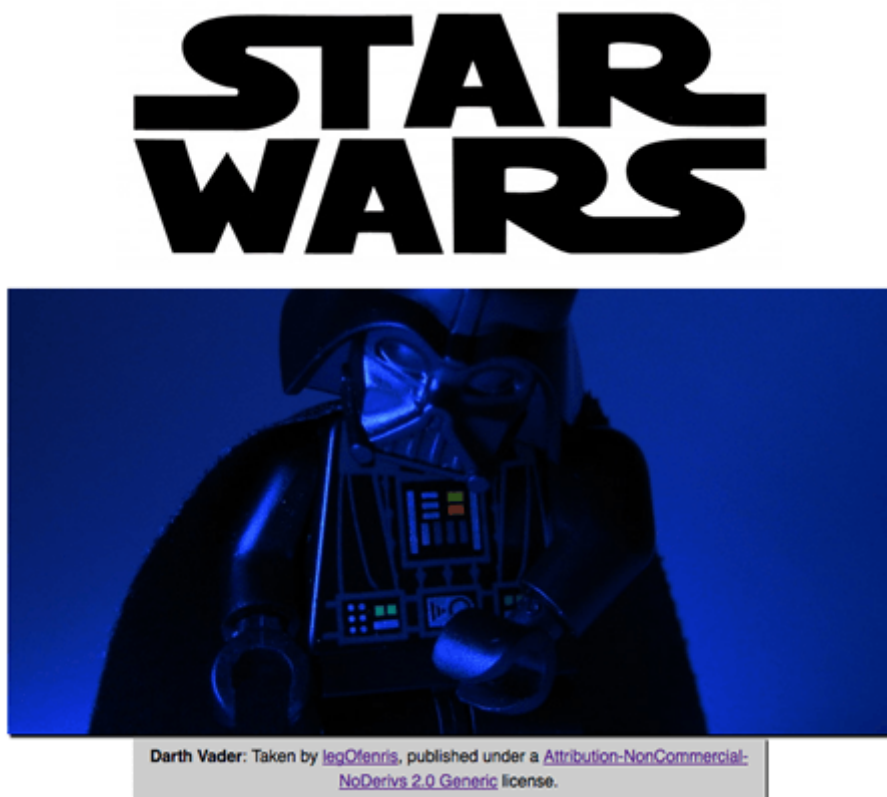
This all happens asynchronously.

Note: You can also chain promise calls together, for example:
`myPromise().then(success, failure).then(success).catch(failure);`

Note: You can find a lot more out about promises by reading Jake Archibald's excellent [JavaScript Promises: there and back again](#).

Service workers demo

To demonstrate just the very basics of registering and installing a service worker, we have created a simple demo called [sw-test](#), which is a simple Star wars Lego image gallery. It uses a promise-powered function to read image data from a JSON object and load the images using Ajax, before displaying the images in a line down the page. We've kept things static and simple for now. It also registers, installs, and activates a service worker, and when more of the spec is supported by browsers it will cache all the files required so it will work offline!



You can see the [source code on GitHub](#), and [view the example live](#). The one bit we'll call out here is the promise (see [app.js lines 22-47](#)), which is a modified version of what you read about above, in the [Promises test demo](#). It is different in the following ways:

1. In the original, we only passed in a URL to an image we wanted to load. In this version, we pass in a JSON fragment containing all the data for a single image (see what they look like in [image-list.js](#)). This is because all the data for each promise resolve has to be passed in with the promise, as it is asynchronous. If you just passed in the url, and then tried to access the other items in the JSON separately when the `for()` loop is being iterated through later on, it wouldn't work, as the promise wouldn't resolve at the same time as the iterations are being done (that is a synchronous process.)
2. We actually resolve the promise with an array, as we want to make the loaded image blob available to the resolving function later on in the code, but also the image name, credits and alt text (see [app.js lines 31-34](#)). Promises will only resolve with a single argument, so if you want to resolve with multiple values, you need to use an array/object.
3. To access the resolved promise values, we then access this function as you'd then expect (see [app.js lines 60-64](#)). This may seem a bit odd at first, but this is the way promises work.

Enter Service workers

Now let's get on to service workers!

Registering your worker

The first block of code in our app's JavaScript file — `app.js` — is as follows. This is our entry point into using service workers.

```
1  if ('serviceWorker' in navigator) {
2    navigator.serviceWorker.register('/sw-test/sw.js', { scope: '/sw-test/' }).then(function
3      // registration worked
4      console.log('Registration succeeded. Scope is ' + reg.scope);
5    }).catch(function(error) {
6      // registration failed
7      console.log('Registration failed with ' + error);
8    });
9  }
```

1. The outer block performs a feature detection test to make sure service workers are supported before trying to register one.
2. Next, we use the `ServiceWorkerContainer.register()` function to register the service worker for this site, which is just a JavaScript file residing inside our app (note this is the file's URL relative to the origin, not the JS file that references it.)
3. The `scope` parameter is optional, and can be used to specify the subset of your content that you want the service worker to control. In this case, we have specified `'/sw-test/'`, which means all content under the app's origin. If you leave it out, it will default to this value anyway, but we specified it here for illustration purposes.

4. The `.then()` promise function is used to chain a success case onto our promise structure. When the promise resolves successfully, the code inside it executes.
5. Finally, we chain a `.catch()` function onto the end that will run if the promise is rejected.

This registers a service worker, which runs in a worker context, and therefore has no DOM access. You then run code in the service worker outside of your normal pages to control their loading.

A single service worker can control many pages. Each time a page within your scope is loaded, the service worker is installed against that page and operates on it. Bear in mind therefore that you need to be careful with global variables in the service worker script: each page doesn't get its own unique worker.

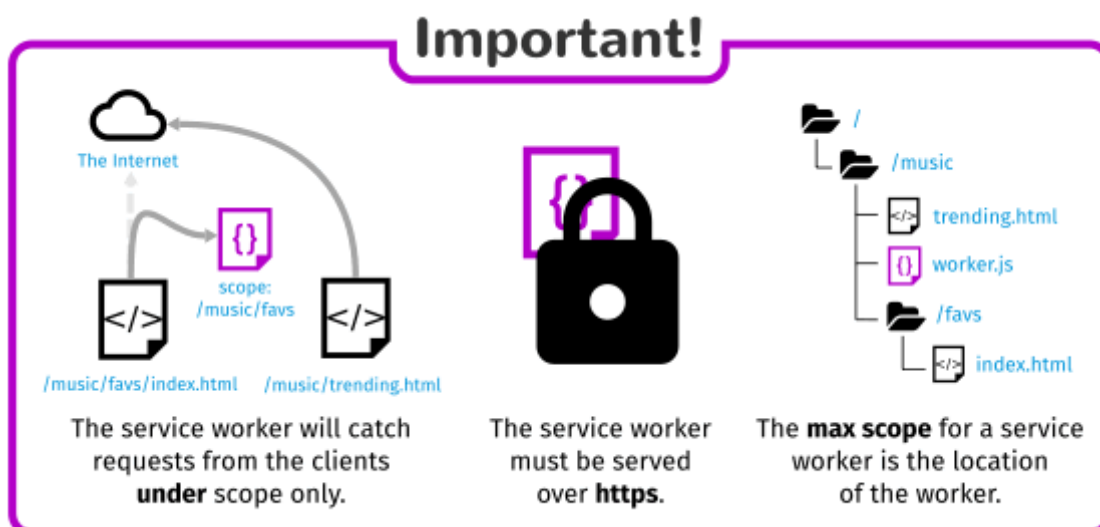
Note: Your service worker functions like a proxy server, allowing you to modify requests and responses, replace them with items from its own cache, and more.

Note: One great thing about service workers is that if you use feature detection like we've shown above, browsers that don't support service workers can just use your app online in the normal expected fashion. Furthermore, if you use AppCache and SW on a page, browsers that don't support SW but do support AppCache will use that, and browsers that support both will ignore the AppCache and let SW take over.

Why is my service worker failing to register?

This could be for the following reasons:

1. You are not running your application through HTTPS.
2. The path to your service worker file is not written correctly — it needs to be written relative to the origin, not your app's root directory. In our example, the worker is at `https://mdn.github.io/sw-test/sw.js`, and the app's root is `https://mdn.github.io/sw-test/`. But the path needs to be written as `/sw-test/sw.js`, not `/sw.js`.
3. The service worker being pointed to is on a different origin to that of your app. This is also not allowed.



Also note:


- The service worker will only catch requests from clients under the service worker's scope.

- The max scope for a service worker is the location of the worker.
- If your server worker is active on a client being served with the Service-Worker-Allowed header, you can specify a list of max scopes for that worker.
- In Firefox, Service Worker APIs are hidden and cannot be used when the user is in [private browsing mode](#).

Install and activate: populating your cache

After your service worker is registered, the browser will attempt to install then activate the service worker for your page/site.

The install event is fired when an install is successfully completed. The install event is generally used to populate your browser's offline caching capabilities with the assets you need to run your app offline. To do this, we use Service Worker's brand new storage API — **cache** — a global on the service worker that allows us to store assets delivered by responses, and keyed by their requests. This API works in a similar way to the browser's standard cache, but it is specific to your domain. It persists until you tell it not to — again, you have full control.

 **Note:** The Cache API is not supported in every browser. (See the [Browser support](#) section for more information.) If you want to use this now, you could consider using a polyfill like the one available in [Google's Topeka demo](#), or perhaps store your assets in [IndexedDB](#).

Let's start this section by looking at a code sample — this is the [first block you'll find in our service worker](#):

```
1  this.addEventListener('install', function(event) {
2    event.waitUntil(
3      caches.open('v1').then(function(cache) {
4        return cache.addAll([
5          '/sw-test/',
6          '/sw-test/index.html',
7          '/sw-test/style.css',
8          '/sw-test/app.js',
9          '/sw-test/image-list.js',
10         '/sw-test/star-wars-logo.jpg',
11         '/sw-test/gallery/',
12         '/sw-test/gallery/bountyHunters.jpg',
13         '/sw-test/gallery/myLittleVader.jpg',
14         '/sw-test/gallery/snowTroopers.jpg'
15       ]);
16     })
17   );
18 });
```

1. Here we add an `install` event listener to the service worker (hence `this`), and then chain a `ExtendableEvent.waitUntil()` method onto the event — this ensures that the Service Worker will not install until the code inside `waitUntil()` has successfully occurred.

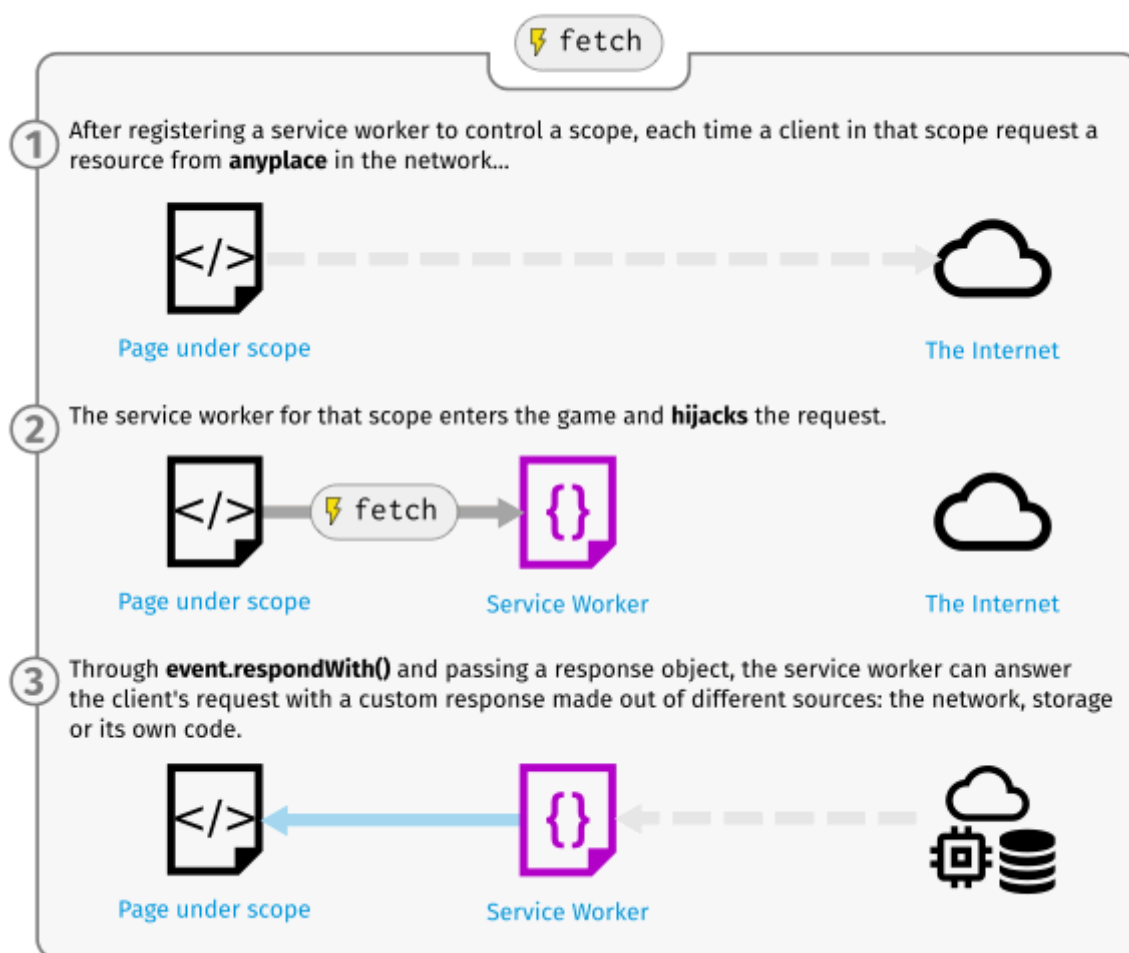
2. Inside `waitUntil()` we use the `caches.open()` method to create a new cache called `v1`, which will be version 1 of our site resources cache. This returns a promise for a created cache; once resolved, we then call a function that calls `addAll()` on the created cache, which for its parameter takes an array of origin-relative URLs to all the resources you want to cache.
3. If the promise is rejected, the install fails, and the worker won't do anything. This is ok, as you can fix your code and then try again the next time registration occurs.
4. After a successful installation, the service worker activates. This doesn't have much of a distinct use the first time your service worker is installed/activated, but it means more when the service worker is updated (see the [Updating your service worker](#) section later on.)

Note: `localStorage` works in a similar way to service worker cache, but it is synchronous, so not allowed in service workers.

Note: `IndexedDB` can be used inside a service worker for data storage if you require it.

Custom responses to requests

Now you've got your site assets cached, you need to tell service workers to do something with the cached content. This is easily done with the `fetch` event.



A `fetch` event fires every time any resource controlled by a service worker is fetched, which includes the documents inside the specified scope, and any resources referenced in those documents (for example if `index.html` makes a cross origin request to embed an image, that still goes through its service worker.)

You can attach a fetch event listener to the service worker, then call the `respondWith()` method on the event to hijack our HTTP responses and update them with your own magic.

```
1 | this.addEventListener('fetch', function(event) {  
2 |     event.respondWith(  
3 |         // magic goes here  
4 |     );  
5 | });
```

We could start by simply responding with the resource whose url matches that of the network request, in each case:

```
1 | this.addEventListener('fetch', function(event) {  
2 |     event.respondWith(  
3 |         caches.match(event.request);  
4 |     );  
5 | });
```

`caches.match(event.request)` allows us to match each resource requested from the network with the equivalent resource available in the cache, if there is a matching one available. The matching is done via url and vary headers, just like with normal HTTP requests.

Let's look at a few other options we have when defining our magic (see our [Fetch API documentation](#) for more information about [Request](#) and [Response](#) objects.)

1. The [Response\(\)](#) constructor allows you to create a custom response. In this case, we are just returning a simple text string:

```
1 | new Response('Hello from your friendly neighbourhood service worker!');
```

2. This more complex Response below shows that you can optionally pass a set of headers in with your response, emulating standard HTTP response headers. Here we are just telling the browser what the content type of our synthetic response is:

```
1 | new Response('<p>Hello from your friendly neighbourhood service worker!</p>', {  
2 |     headers: { 'Content-Type': 'text/html' }  
3 | })
```

3. If a match wasn't found in the cache, you could tell the browser to simply [fetch](#) the default network request for that resource, to get the new resource from the network if it is available:

```
1 | fetch(event.request)
```

4. If a match wasn't found in the cache, and the network isn't available, you could just match the request with some kind of default fallback page as a response using `match()`, like this:

```
1 | caches.match('/fallback.html');
```

5. You can retrieve a lot of information about each request by calling parameters of the `Request` object returned by the `FetchEvent`:

```
1 | event.request.url
2 | event.request.method
3 | event.request.headers
4 | event.request.body
```

Recovering failed requests

So `caches.match(event.request)` is great when there is a match in the service worker cache, but what about cases when there isn't a match? If we didn't provide any kind of failure handling, our promise would reject and we would just come up against a network error when a match isn't found.

Fortunately service workers' promise-based structure makes it trivial to provide further options towards success. We could do this:

```
1 | this.addEventListener('fetch', function(event) {
2 |   event.respondWith(
3 |     caches.match(event.request).catch(function() {
4 |       return fetch(event.request);
5 |     })
6 |   );
7 | });
```

If the promise rejects, the `catch()` function returns the default network request for the resource instead, meaning that those who have network available can just load the resource from the server.

If we were being really clever, we would not only request the resource from the network; we would also save it into the cache so that later requests for that resource could be retrieved offline too! This would mean that if extra images were added to the Star Wars gallery, our app could automatically grab them and cache them. The following would do the trick:

```
1 | this.addEventListener('fetch', function(event) {
2 |   event.respondWith(
```

```
3     caches.match(event.request).catch(function() {
4         return fetch(event.request).then(function(response) {
5             return caches.open('v1').then(function(cache) {
6                 cache.put(event.request, response.clone());
7                 return response;
8             });
9         });
10    });
11    );
12    });
```

Here we return the default network request with `return fetch(event.request)`, which returns a promise. When this promise is resolved, we respond by running a function that grabs our cache using `caches.open('v1')`; this also returns a promise. When that promise resolves, `cache.put()` is used to add the resource to the cache. The resource is grabbed from `event.request`, and the response is then cloned with `response.clone()` and added to the cache. The clone is put in the cache, and the original response is returned to the browser to be given to the page that called it.

Why? This is because request and response streams can only be read once. In order to return the response to the browser and put it in the cache we have to clone it. So the original gets returned to the browser and the clone gets sent to the cache. They are each read once.

The only trouble we have now is that if the request doesn't match anything in the cache, and the network is not available, our request will still fail. Let's provide a default fallback so that whatever happens, the user will at least get something:

```
1  this.addEventListener('fetch', function(event) {
2      event.respondWith(
3          caches.match(event.request).catch(function() {
4              return fetch(event.request).then(function(response) {
5                  return caches.open('v1').then(function(cache) {
6                      cache.put(event.request, response.clone());
7                      return response;
8                  });
9              });
10         }).catch(function() {
11             return caches.match('/sw-test/gallery/myLittleVader.jpg');
12         })
13     );
14 });
```

We have opted for this fallback image because the only updates that are likely to fail are new images, as everything else is depended on for installation in the `install` event listener we saw earlier.

Updated code pattern suggestion

This uses more standard promise chaining and returns the response to the document without having to wait for `caches.open()` to resolve:

```
1  this.addEventListener('fetch', function(event) {
2    event.respondWith(caches.match(event.request).catch(function() {
3      return fetch(event.request);
4    })).then(function(response) {
5      caches.open('v1').then(function(cache) {
6        cache.put(event.request, response);
7      });
8      return response.clone();
9    }).catch(function() {
10     return caches.match('/sw-test/gallery/myLittleVader.jpg');
11   }));
12 });
```

Updating your service worker

If your service worker has previously been installed, but then a new version of the worker is available on refresh or page load, the new version is installed in the background, but not yet activated. It is only activated when there are no longer any pages loaded that are still using the old service worker. As soon as there are no more such pages still loaded, the new service worker activates.

You'll want to update your `install` event listener in the new service worker to something like this (notice the new version number):

```
1  this.addEventListener('install', function(event) {
2    event.waitUntil(
3      caches.open('v2').then(function(cache) {
4        return cache.addAll([
5          '/sw-test/',
6          '/sw-test/index.html',
7          '/sw-test/style.css',
8          '/sw-test/app.js',
9          '/sw-test/image-list.js',
10
11          ...
12
13          // include other new resources for the new version...
14        ]);
15      });
16    );
17  });
```

While this happens, the previous version is still responsible for fetches. The new version is installing in the background. We are calling the new cache `v2`, so the previous `v1` cache isn't disturbed.

When no pages are using the current version, the new worker activates and becomes responsible for fetches.

Deleting old caches

You also get an `activate` event. This is a generally used to do stuff that would have broken the previous version while it was still running, for example getting rid of old caches. This is also useful for removing data that is no longer needed to avoid filling up too much disk space — each browser has a hard limit on the amount of cache storage that a given service worker can use. The browser does its best to manage disk space, but it may delete the Cache storage for an origin. The browser will generally delete all of the data for an origin or none of the data for an origin.

Promises passed into `waitUntil()` will block other events until completion, so you can rest assured that your clean-up operation will have completed by the time you get your first fetch event on the new cache.

```
1  this.addEventListener('activate', function(event) {
2    var cacheWhitelist = ['v2'];
3
4    event.waitUntil(
5      caches.keys().then(function(keyList) {
6        return Promise.all(keyList.map(function(key) {
7          if (cacheWhitelist.indexOf(key) === -1) {
8            return caches.delete(key);
9          }
10         }));
11      })
12    );
13  });
```

Dev tools

Chrome has `chrome://inspect/#service-workers`, which shows current service worker activity and storage on a device, and `chrome://serviceworker-internals`, which shows more detail and allows you to start/stop/debug the worker process. In the future they will have throttling/offline modes to simulate bad or non-existent connections, which will be a really good thing.

Firefox has also started to implement some useful tools related to service workers:

- You can navigate to <about:serviceworkers> to see what SWs are registered and update/remove them.
- When testing you can get around the HTTPS restriction by checking the "Enable Service Workers over HTTP (when toolbox is open)" option in the Firefox Devtools options (gear menu.)

Specifications

Specification	Status	Comment
Service Workers	WD Working Draft	Initial definition.

Browser compatibility

	Desktop	Mobile			
Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari (WebKit)
Basic support	40.0	33.0 (33.0)[1]	未实现	24	未实现

[1] Service workers (and [Push](#)) have been disabled in the [Firefox 45 Extended Support Release](#) (ESR.)

See also

- [The Service Worker Cookbook](#)
- [Is ServiceWorker ready?](#)
- Download the [Service Workers 101 cheatsheet](#).
- [Promises](#)
- [Using web workers](#)