

ServiceWorker: A Basic Guide to BackgroundSync

serviceworker

offline

backgroundsync



📅 2016 ⌚ 10m 💬 0

Picture the scene. You're using your mobile device to browse a website and book some important concert tickets. You've have been eagerly awaiting this concert for months. You get right through to the checkout screen, enter your card details and hit submit. And then your flaky 3G connection drops. Screaming and pain ensues!


I'm super eager to share the first of many guest posts that will be featured on Pony Foo! 📄 🌈 🇺🇸

39M

When Dean came to me with the idea of writing an article on ServiceWorker, I was immediately thrilled with his proposal to write about BackgroundSync. I hadn't quite covered that area in previous [serviceworker](#) articles, and he's put together a piece on what BackgroundSync is, how it works, and why it's useful.

— Editor's note.

I can't even begin to tell you the number of times I've tried to submit a form or complete an action, only for my connection to drop and the browser to present me with an offline page. Up until now, we've been unable to deal with situations like these, and it's a pretty crappy experience for our users. Fortunately, this is where Service Workers can come to the rescue. I've previously blogged about using Service Workers to provide [offline functionality](#) and dramatically speed up your load times using caching. While this is useful, there has been no way for the page to *actually* send something to the server without connectivity. [Background Sync](#) is the feature that has been built to handle just such a scenario.

 Background Sync Logo

Background Sync Logo

So what exactly is Background Sync? Well, it is a new web API that lets you defer actions until the user has stable connectivity. This makes it great for ensuring that whatever the user wants to send, is actually sent. The user can go offline and even close the browser, safe in the knowledge that their request will be sent when they regain connectivity.

In this article, I am going to run through a simple example that shows you how to use Background Sync to ensure that your requests are queued even when the user is offline. We are going to take a look at a basic page that makes an HTTP request and will respond differently depending on whether or not the user is connected or not.

In order to get started with this example, you'll need to have a basic understanding of Service Workers, but don't worry if you aren't familiar with them - there are some great resources online. I recommend checking out [slightlyoff/ServiceWorker](#) for more info.

Getting Started

Imagine that you have the following HTML page. It's pretty basic, but it gives you the general idea.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Home Page</title>
```

```

</head>
<body>
  <div id="connectionStatus"></div>
  <p>Click on the button below to make an HTTP request.<p>
  <button id="requestButton">Make an HTTP request - do it!</button>
</body>
</html>

```

The HTML above shows a simple button that will make an HTTP request for an image when clicked. If the webpage has a connection, the HTTP request should go through without any issues. However if there is no connection, the HTTP request will be queued and synced as soon as the user has a connection again.

In order to use Background Sync, we are going need to create a Service Worker to unlock this functionality. Let's start off by registering the Service Worker in the page that we have just created. Add the following code to your HTML.

```

if ('serviceWorker' in navigator) {
  navigator.serviceWorker
    .register('./service-worker.js')
    .then(registration => navigator.serviceWorker.ready)
    .then(registration => { // register sync
      document.getElementById('requestButton').addEventListener('click', () => {
        registration.sync.register('image-fetch').then(() => {
          console.log('Sync registered');
        });
      });
    });
} else {
  document.getElementById('requestButton').addEventListener('click', () => {
    console.log('Fallback to fetch the image as usual');
  });
}

```

The code above might look a little hairy at first, but let's break it down. First, I am doing a simple check to see if the browser supports Service Workers. If it does, I then register a file called **service-worker.js**. This file will contain the Service Worker code with all of the Background Sync magic. We are going to create this shortly. You might also notice that if the browser doesn't support Service Workers, the code simply falls back and will make an HTTP request as normal without using Background Sync.

Next, if the registration is successful, I am adding a *click* event to the button and registering a sync called *image-fetch*. This is just a simple string that I've named to help me recognize this event. You can think of these sync names a simple tags for different actions - you can have as many as you want!

The Service Worker

Next, we need to create the Service Worker file and call it 'service-worker.js'. We are going to use this Service Worker to make an HTTP request for a simple image and return it if successful.

Once I've created the *service-worker.js* file, I'll add the following code:

```

self.addEventListener('sync', function (event) {
  if (event.tag === 'image-fetch') {
    event.waitUntil(fetchDogImage());
  }
}

```

```
});
```

The code above creates an event that listens out for the *sync* event. If their page has a connection, it will try and fetch the image, and if it fulfills, the sync is complete. If it fails, another sync will be scheduled to retry. Retry syncs also wait for connectivity, and employ an exponential back-off.

Next, I need to add a function that fetches the image of the dog.

```
function fetchDogImage () {  
  fetch('./doge.png')  
    .then(function (response) {  
      return response;  
    })  
    .then(function (text) {  
      console.log('Request successful', text);  
    })  
    .catch(function (error) {  
      console.log('Request failed', error);  
    });  
}
```

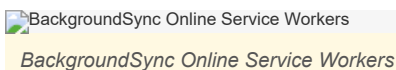
The code above uses the Fetch API to retrieve an image from the server. It will return a promise indicating if it was successful or if it failed with the HTTP response.

And that's it! You are now able to use BackgroundSync to enable offline actions that will queue depending on connectivity.

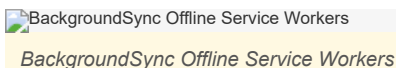
Testing in the wild

Now that you have your page ready to sync events, you can test it in action. If you'd like to see this example in action, I've created a GitHub page that you can use to experiment with. Head over to [the demo page](#) to find out more.

When you have an active connection, you should see network requests going through in the **Network** tab of your developer tools.



If you aren't connected to the internet, your request will be synced when the user regains connectivity. In order to test this, I simply disabled the wifi, made an HTTP request and then re-enabled the wifi.



As soon as you re-enable the wifi connection, you should see the HTTP request appear in the network tab as normal.

I did notice a few things with the behavior of the sync when I was testing. Firstly, the tag name of the sync should be unique for a given sync. If you register for a sync using the same tag as a pending sync, it will combine with the existing sync. Which means that in the case of this example, if you register for a "image-fetch" sync every time the user clicks the button, you will only receive one sync when they connect again. If you want 5 separate sync events, you'll need to use unique tags.

Notifying the user

The example that we've run through isn't very pretty and doesn't provide the user with much feedback. It would be a much better experience if the user received feedback about the current state of their connectivity while viewing the page.

Using a clever bit of code, I can listen out for any changes in the user connectivity.

```
// Connection Status
function isOnline () {
  var connectionStatus = document.getElementById('connectionStatus');

  if (navigator.onLine){
    connectionStatus.innerHTML = 'You are currently online!';
  } else {
    connectionStatus.innerHTML = 'You are currently offline. Any requests made will be queued and synced as soon as you are connected again.';
  }
}

window.addEventListener('online', isOnline);
window.addEventListener('offline', isOnline);
isOnline();
```

In the code above, I've added an event listener on the window object to listen for connectivity changes. As soon as this connectivity state changes, I'm updating the page with a notification message depending on the result. By adding the above code to our web page, we can update the user every time their connection state changes.

The code above isn't bulletproof - `navigator.onLine` only knows if the user can't connect to a LAN or router - it doesn't know if the the user can actually connect to the internet. That said, I still think it can be useful to provide the user with some feedback when they are trying to perform an action if they aren't connected!

The Future

There is still more to come for Background Sync! The team working on Background Sync are currently working on a **Periodic Background Sync**. This will allow you to request a "periodicsync" event that will be restricted by time, battery state and network state. The API is still in design, but it might look a little something like the code below:

```
navigator.serviceWorker.ready.then(function(registration) {
  registration.periodicSync.register({
    tag: 'get-latest-news',          // default: ''
    minPeriod: 12 * 60 * 60 * 1000, // default: 0
    powerState: 'avoid-draining',    // default: 'auto'
    networkState: 'avoid-cellular'   // default: 'online'
  }).then(function(periodicSyncReg) {
    // success
  }, function() {
    // failure
  })
});
```

The code above is pretty similar to a standard Background Sync, except that it has a few more parameters. The **minPeriod** is the the minimum time between successful sync events, **powerState** can be used to only sync when the battery is charging, and **networkState** gives you control over

whether you want to sync over wifi versus a cellular connection. Very cool!

The great thing about periodic sync's is that they don't require any server configuration, and allow the browser to optimize when they fire to be most-helpful and least-disruptive to the user. All this can be done on the client without any server configuration! I can already think of a load of powerful uses for this feature.

The API is still a work in progress, but if you'd like to keep up to date with the latest changes, I recommend keeping a close eye on the [WICG Background Sync Spec](#).

Summary

As a user, not being able to perform an action when you have no connectivity can be extremely frustrating. Fortunately, the functionality that Background Sync brings is a game changer. I am excited about the awesome possibilities that they bring to the web.

If you'd like to see a working example of this article in action, please head over to [the demo page](#). All of the code behind this is available at [deanhume/Service-Workers-BackgroundSync](#).

