

Push Notifications on the Open Web



By Matt Gaunt

Matt is a contributor to WebFundamentals

Warning: This blog post is getting a bit old. If you are looking to learn more about implementing push, check out our [Web Push Notifications](#) documentation.

If you ask a room of developers what mobile device features are missing from the web, push notifications are always high on the list.

Push notifications allow your users to opt-in to timely updates from sites they love and allow you to effectively re-engage them with customized, engaging content.

As of Chrome version 42, the [Push API](#) and [Notification API](#) are available to developers.

The Push API in Chrome relies on a few different pieces of technology, including [Web App Manifests](#) and [Service Workers](#). In this post we'll look at each of these technologies, but only the bare minimum to get push messaging up and running. To get a better understanding of some of the other features of manifests and the offline capabilities of service workers, please check out the links above.

We will also look at what will be added to the API in future versions of Chrome, and finally we'll have an FAQ.

Implementing Push Messaging for Chrome

This section describes each step you need to complete in order to support push messaging in your web app.

Register a Service Worker

There is a dependency of having a service worker to implement push messages for the web. The reason for this is that when a push message is received, the browser can start up a service worker, which runs in the background without a page being open, and dispatch an event so that you can decide how to handle that push message.

Below is an example of how you register a service worker in your web app. When the registration has completed successfully we call **initialiseState()**, which we'll cover shortly.

```
var isPushEnabled = false;
```



```
...
```

```
window.addEventListener('load', function() {
  var pushButton = document.querySelector('.js-push-button');
  pushButton.addEventListener('click', function() {
    if (isPushEnabled) {
      unsubscribe();
    } else {
      subscribe();
    }
  });

  // Check that service workers are supported, if so, progressively
  // enhance and add push messaging support, otherwise continue without it.
  if ('serviceWorker' in navigator) {
    navigator.serviceWorker.register('/service-worker.js')
      .then(initialiseState);
  } else {
    console.warn('Service workers aren\'t supported in this browser.');
```

The button click handler subscribes or unsubscribes the user to push messages.

isPushEnabled is a global variable which simply tracks whether push messaging is currently subscribed or not. These will be referenced throughout the code snippets.

We then check that service workers are supported before registering the `service-worker.js` file which has the logic for handling a push message. Here we are simply telling the browser that this JavaScript file is the service worker for our site.

Set Up the Initial State

 Enable Push Notifications

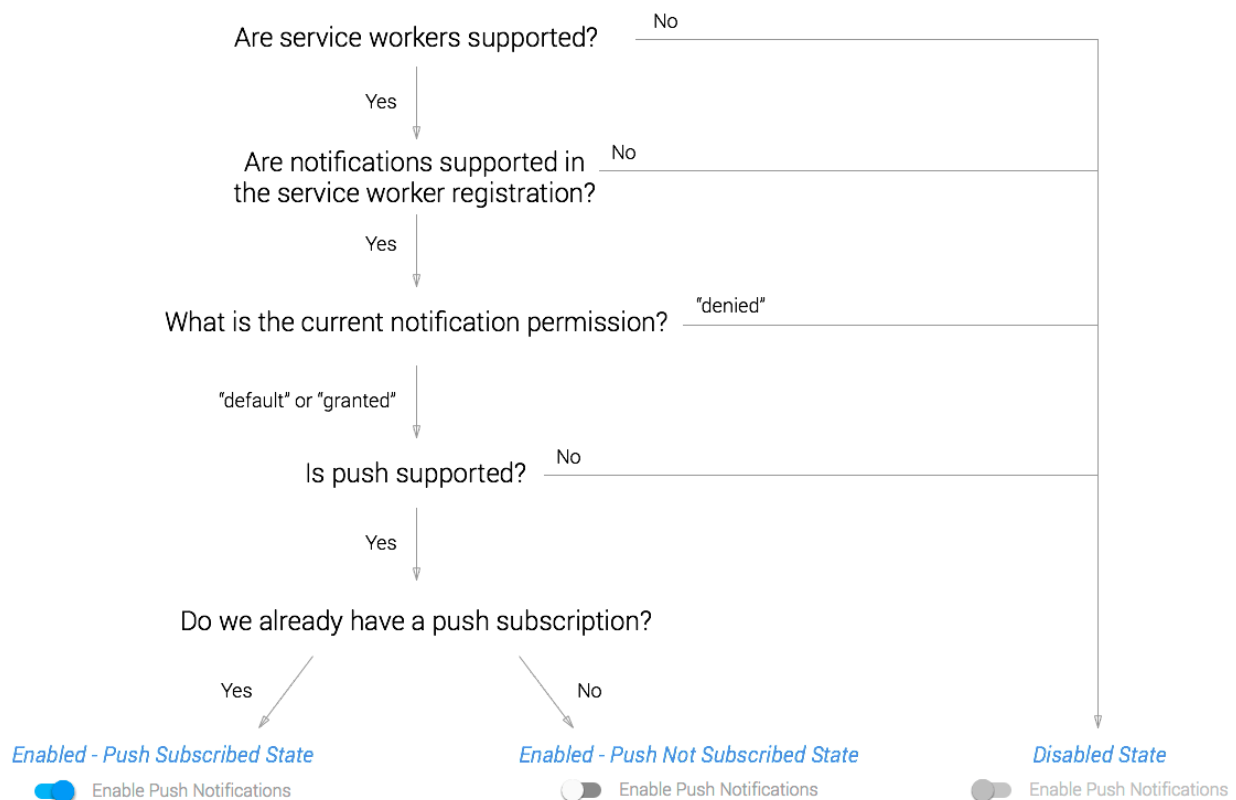
 Enable Push Notifications

Once the service worker is registered, we need to set up our UI's state.

Users will expect a simple UI to enable or disable push messages for your site, and they'll expect it to keep up to date with any changes that occur. In other words, if they enable push messages for your site, leave and come back a week later, your UI should highlight that push messages are already enabled.

You can find some [UX guidelines in this doc](#), in this article we'll be focusing on the technical aspects.

At this point you may be thinking there are only two states to deal with, enabled or disabled. There are however some other states surrounding notifications which you need to take into account.



There are a number of APIs we need to check before we enable our button, and if everything is supported, we can enable our UI and set the initial state to indicate whether push messaging is subscribed or not.

Since the majority of these checks result in our UI being disabled, you should set the initial state to disabled. This also avoids any confusion should there be an issue with your page's JavaScript, for example the JS file can't be downloaded or the user has disabled JavaScript.

```
<button class="js-push-button" disabled>
  Enable Push Messages
</button>
```



With this initial state, we can perform the checks outlined above in the **initialiseState()** method, i.e. after our service worker is registered.



```
// Once the service worker is registered set the initial state
function initialiseState() {
  // Are Notifications supported in the service worker?
  if (!('showNotification' in ServiceWorkerRegistration.prototype)) {
    console.warn('Notifications aren\'t supported.');
```

```
    return;
  }

  // Check the current Notification permission.
  // If its denied, it's a permanent block until the
  // user changes the permission
  if (Notification.permission === 'denied') {
    console.warn('The user has blocked notifications.');
```

```
    return;
  }

  // Check if push messaging is supported
  if (!('PushManager' in window)) {
    console.warn('Push messaging isn\'t supported.');
```

```
    return;
  }

  // We need the service worker registration to check for a subscription
  navigator.serviceWorker.ready.then(function(serviceWorkerRegistration) {
    // Do we already have a push message subscription?
    serviceWorkerRegistration.pushManager.getSubscription()
      .then(function(subscription) {
        // Enable any UI which subscribes / unsubscribes from
        // push messages.
        var pushButton = document.querySelector('.js-push-button');
        pushButton.disabled = false;

        if (!subscription) {
          // We aren't subscribed to push, so set UI
          // to allow the user to enable push
          return;
        }

        // Keep your server in sync with the latest subscriptionId
        sendSubscriptionToServer(subscription);

        // Set your UI to show they have subscribed for
        // push messages
        pushButton.textContent = 'Disable Push Messages';
        isPushEnabled = true;
      });
  });
}
```

```

    })
    .catch(function(err) {
        console.warn('Error during getSubscription()', err);
    });
});
}

```

A brief overview of these steps:

- We check that **showNotification** is available in the ServiceWorkerRegistration prototype. Without it we won't be able to show a notification from our service worker when a push message is received.
- We check what the current **Notification.permission** is to ensure it's not "**denied**". A denied permission means that you can't show notifications until the user manually changes the permission in the browser.
- To check if push messaging is supported we check that **PushManager** is available in the window object.
- Finally, we used **pushManager.getSubscription()** to check whether we already have a subscription or not. If we do, we send the subscription details to our server to ensure we have the right information and set our UI to indicate that push messaging is already enabled or not. We'll look at what details exist in the subscription object later in this article.

We wait until `navigator.serviceWorker.ready` is resolved to check for a subscription and to enable the push button because it's only after the service worker is active that you can actually subscribe to push messages.

The next step is to handle when the user wants to enable push messages, but before we can do this, we need to set up a Google Developer Console project and add some parameters to our manifest to use Firebase Cloud Messaging (FCM), formerly known as Google Cloud Messaging (GCM).

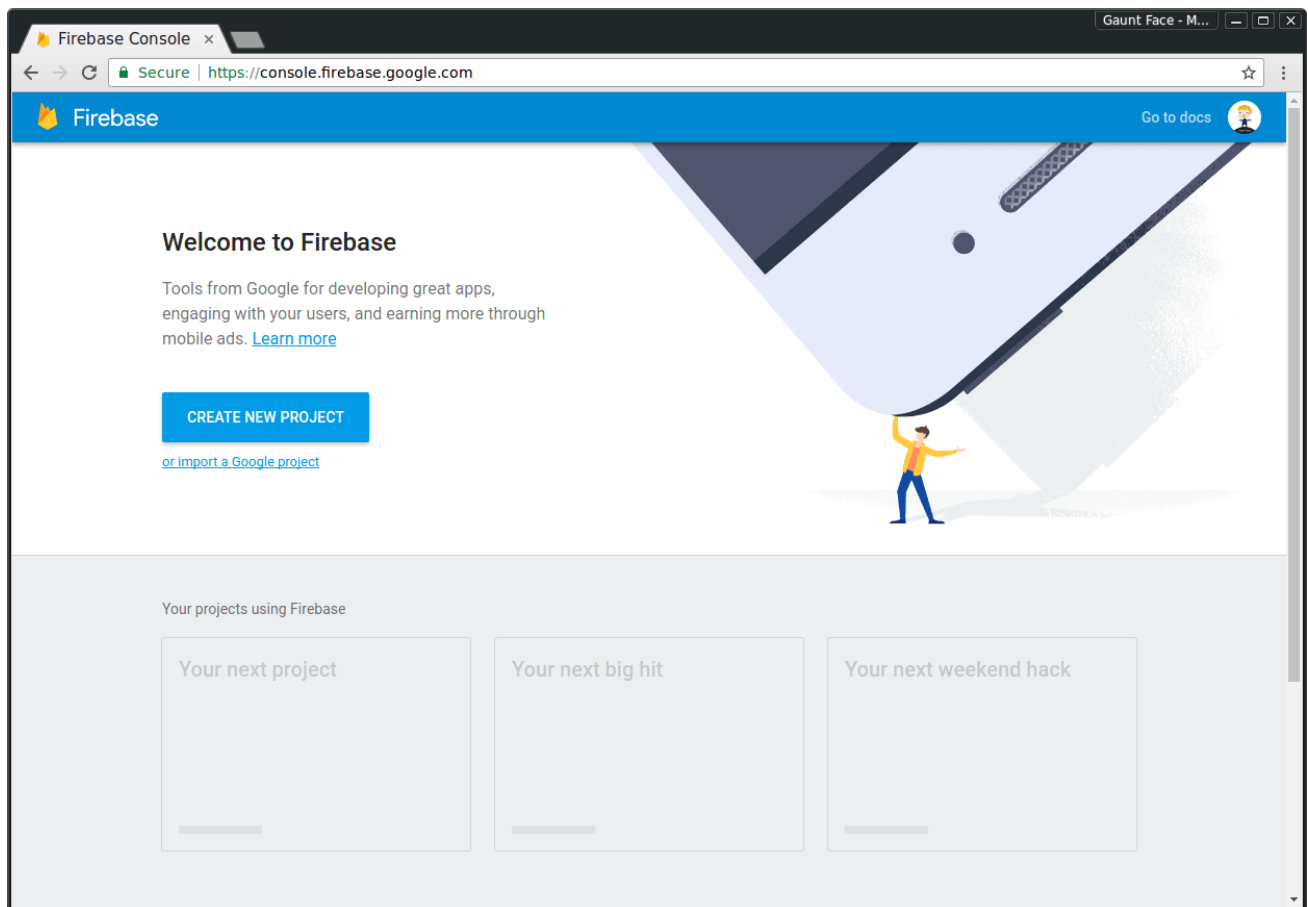
Make a Project on the Firebase Developer Console

Chrome uses FCM to handle the sending and delivery of push messages; however, to use the FCM API, you need to set up a project on the Firebase Developer Console.

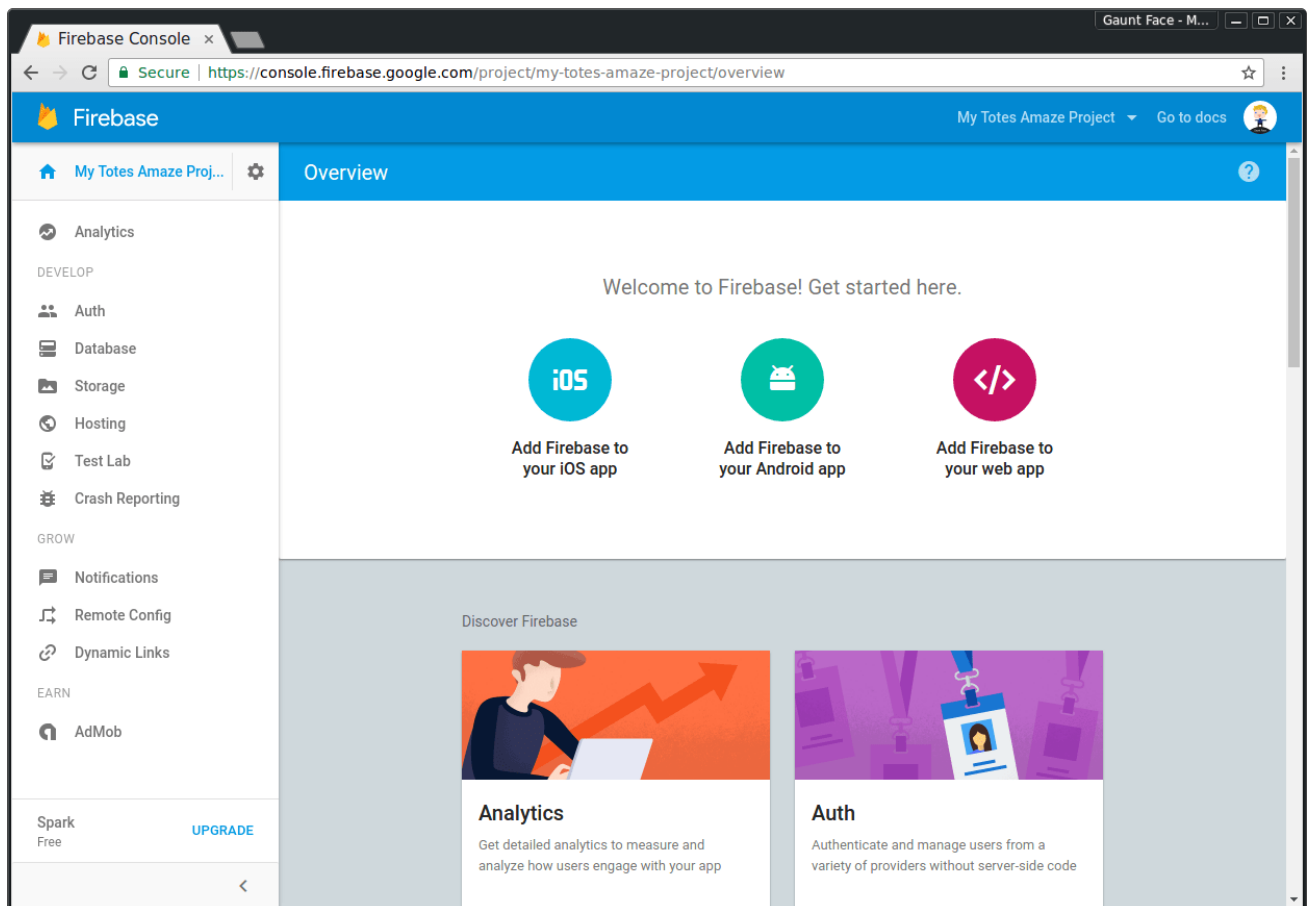
The following steps are specific to Chrome, Opera for Android and Samsung Browser they use FCM. We'll discuss how this would work in other browsers later on in the article.

Create a new Firebase Developer Project

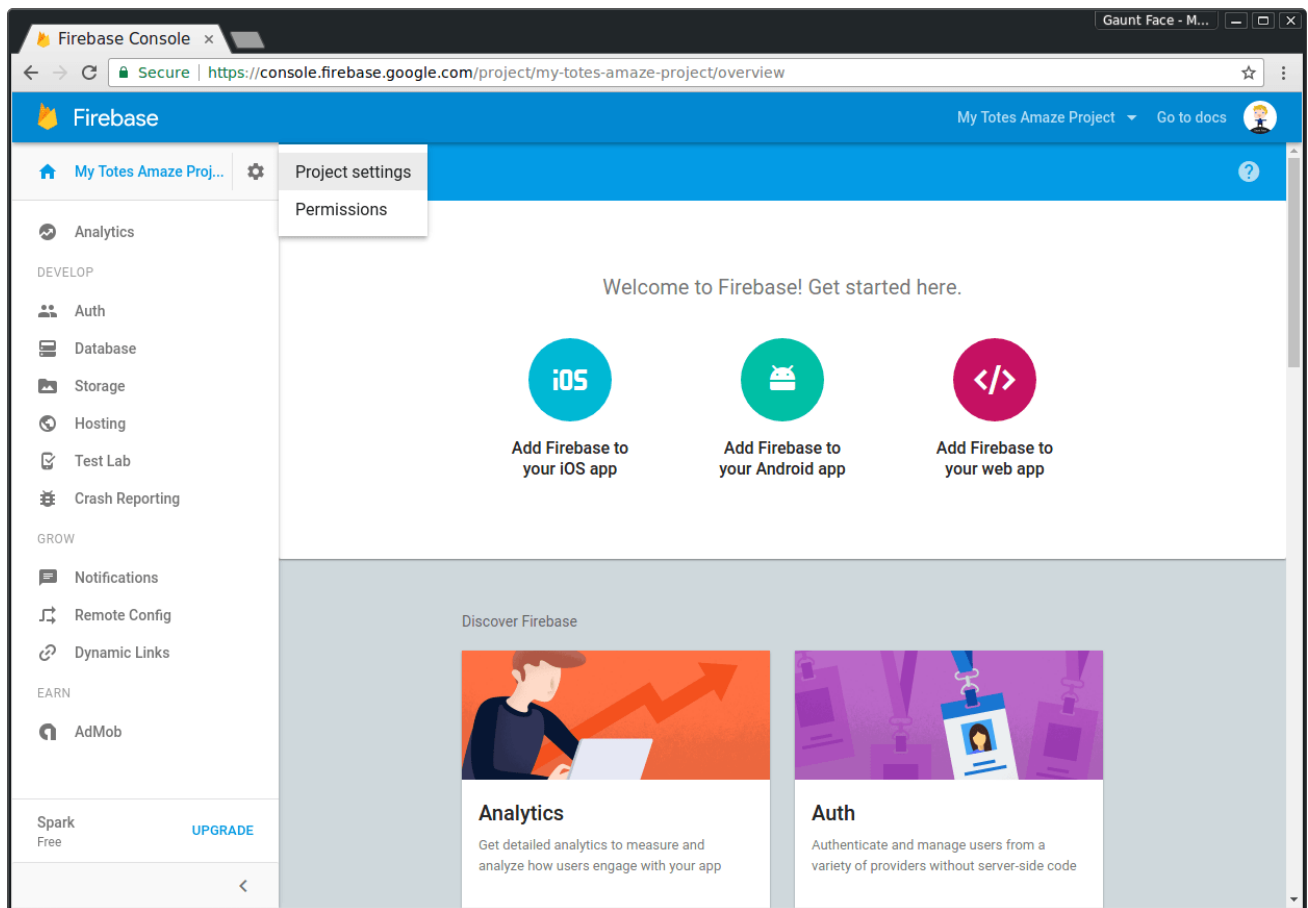
To start off with you need to create a new project on <https://console.firebase.google.com/> by clicking on the 'Create New Project'.



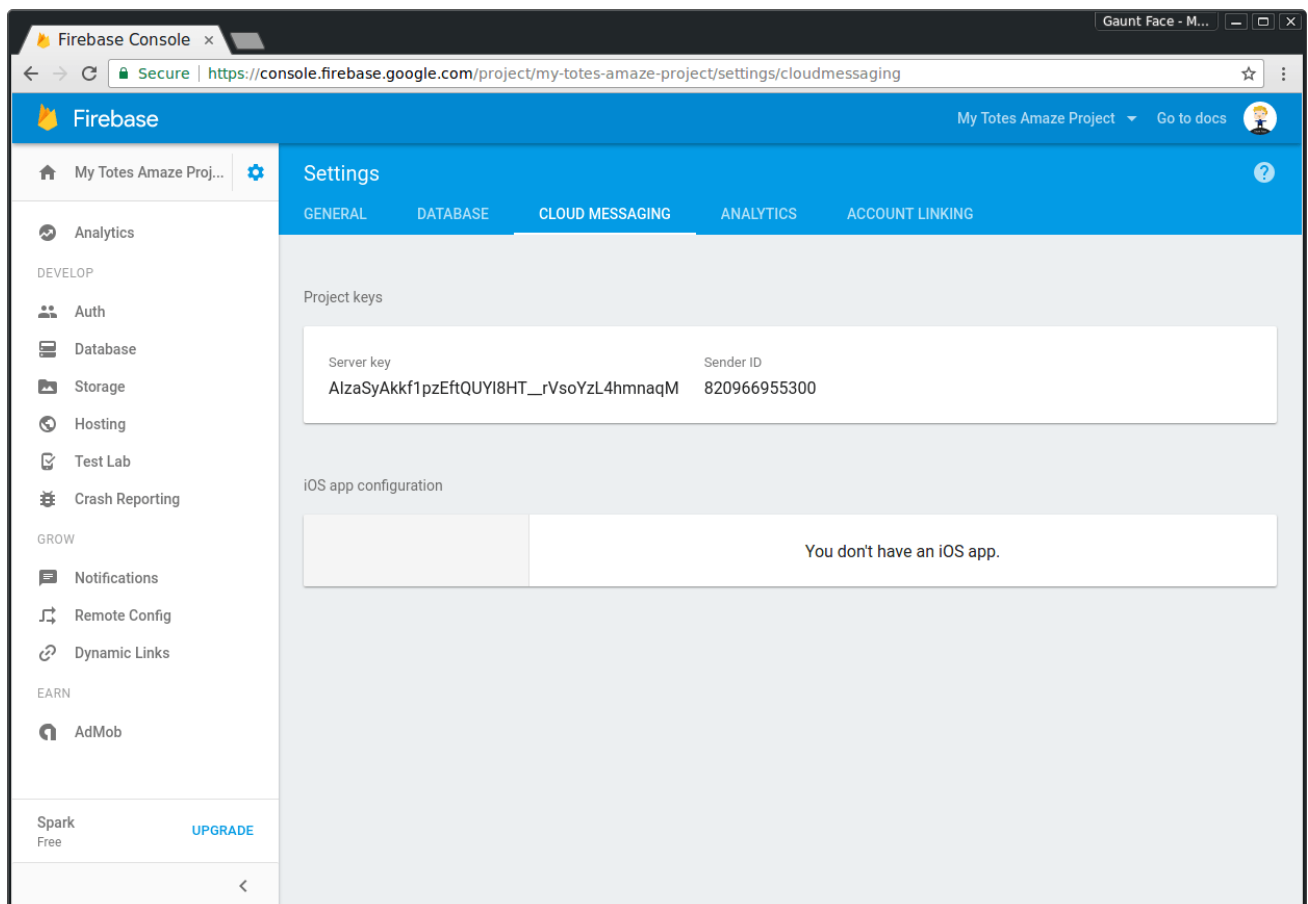
Add a project name, create the project and you'll be taken to the project dashboard:



From this dashboard, click the cog next to your project name in the top left corner and click 'Project Settings'.



In the settings page, click the 'Cloud Messaging' tab.



This page contains the API key for push messaging, which we'll use later on, and the sender ID which we need to put in the web app manifest in the next section.

Add a Web App Manifest

For push, we need to add a manifest file with a **gcm_sender_id** field, to get the push subscription to succeed. This parameter is only required by Chrome, Opera for Android and Samsung Browser so that they can use FCM / GCM.

The **gcm_sender_id** is used by these browsers when it subscribes a users device with FCM. This means that FCM can identify the user's device and make sure your sender ID matches the corresponding API key and that the user has permitted your server to send them push messages.

Below is a super-simple manifest file:

```
{
  "name": "Push Demo",
  "short_name": "Push Demo",
  "icons": [{
    "src": "images/icon-192x192.png",
    "sizes": "192x192",
    "type": "image/png"
  }],
  "start_url": "/index.html?homescreen=1",
  "display": "standalone",
  "gcm_sender_id": "<Your Sender ID Here>"
}
```



You'll need to set the **gcm_sender_id** value to the sender ID from your Firebase Project.

Once you have saved your manifest file in your project (manifest.json is a good name), reference it from your HTML with the following tag in the head of your page.

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



If you don't add a web manifest with these parameters you'll get an exception when you attempt to subscribe the user to push messages, with the error "Registration failed - no sender id provided" or "Registration failed - permission denied".

Subscribe to Push Messaging

Now that you've got a manifest set up you can go back into your sites JavaScript.

To subscribe, you have to call the **subscribe()** method on the PushManager object, which you access through the ServiceWorkerRegistration.

This will ask the user to give your origin permission to send push notifications. Without this permission, you will not be able to successfully subscribe.

If the promise returned by the **subscribe()** method resolves, you'll be given a PushSubscription object which will contain an **endpoint**.

The **endpoint** should be saved on your server for each user, since you'll need them to send push messages at a later date.

The following code subscribes the user for push messaging:

```
function subscribe() {  
  // Disable the button so it can't be changed while  
  // we process the permission request  
  var pushButton = document.querySelector('.js-push-button');  
  pushButton.disabled = true;  
  
  navigator.serviceWorker.ready.then(function(serviceWorkerRegistration) {  
    serviceWorkerRegistration.pushManager.subscribe()  
      .then(function(subscription) {  
        // The subscription was successful  
        isPushEnabled = true;  
        pushButton.textContent = 'Disable Push Messages';  
        pushButton.disabled = false;  
  
        // TODO: Send the subscription.endpoint to your server  
        // and save it to send a push message at a later date  
        return sendSubscriptionToServer(subscription);  
      })  
      .catch(function(e) {  
        if (Notification.permission === 'denied') {  
          // The user denied the notification permission which  
          // means we failed to subscribe and the user will need  
          // to manually change the notification permission to  
          // subscribe to push messages  
          console.warn('Permission for Notifications was denied');  
          pushButton.disabled = true;  
        } else {  
          // A problem occurred with the subscription; common reasons  
          // include network errors, and lacking gcm_sender_id and/or  
          // gcm_user_visible_only in the manifest.  
          console.error('Unable to subscribe to push.', e);  
          pushButton.disabled = false;  
          pushButton.textContent = 'Enable Push Messages';  
        }  
      })  
    }  
  }  
}
```




```

    }
  });
});
}

```

At this point your web app is ready to receive a push message, although nothing will happen until we add a push event listener to our service worker file.

Service Worker Push Event Listener

When a push message is received (we'll talk about how to actually send a push message in the next section), a **push event** will be dispatched in your service worker, at which point you'll need to display a [notification](#) .

```

self.addEventListener('push', function(event) {
  console.log('Received a push message', event);

  var title = 'Yay a message.';
  var body = 'We have received a push message.';
  var icon = '/images/icon-192x192.png';
  var tag = 'simple-push-demo-notification-tag';

  event.waitUntil(
    self.registration.showNotification(title, {
      body: body,
      icon: icon,
      tag: tag
    })
  );
});

```



This code registers a **push** event listener and displays a notification with a predefined title, body text, icon and a notification tag. One subtlety to highlight with this example is the **`event.waitUntil()`** method. This method takes in a [promise](#) and extends the lifetime of an event handler (or can be thought of as keeping the service worker alive), until the promise is [settled](#); In this case, the promise passed to `event.waitUntil` is the returned Promise from **`showNotification()`**.

The [notification tag](#) acts as an identifier for unique notifications. If we sent two push messages to the same endpoint, with a short delay between them, and display notifications with the same tag, the browser will display the first notification and replace it with the second notification when the push message is received.

If you want to show multiple notifications at once then use a different tag, or no tag at all. We'll look at a more complete example of showing a notification later on in this post. For now, let's keep things simple and see if sending a push message shows this notification.

Sending a Push Message

We've subscribed to push messages and our service worker is ready to show a notification, so it's time to send a push message through FCM.


This is only applicable to the browsers using FCM.

When you send the `PushSubscription.endpoint` variable to your server, the endpoint for FCM is special. It has a parameter on the end of the URL which is a `registration_id`.


An example endpoint would be:

`https://android.googleapis.com/gcm/send/APA91bHPffi8zclbIBDcToXN_LEpT6iA87p_` 

The FCM URL is:

`https://android.googleapis.com/gcm/send` 


The `registration_id` would be:

`APA91bHPffi8zclbIBDcToXN_LEpT6iA87pgR-J-MuuVVycM0SmptG-rXdCPKTM5pvKiHk2Ts-u` 

This is specific to browsers using FCM. In a normal browser you would simply get an endpoint and you would call that endpoint in a standard way and it would work regardless of the URL.

What this means is that on your server you'll need to check if the endpoint is for FCM and if it is, extract the `registration_id`. To do this in Python you could do something like:

```
if endpoint.startswith('https://android.googleapis.com/gcm/send'):  
    endpointParts = endpoint.split('/')  
    registrationId = endpointParts[len(endpointParts) - 1]  
  
    endpoint = 'https://android.googleapis.com/gcm/send'
```



Once you've got the registration ID, you can make a call to the FCM API. You can find [reference docs on the FCM API here](#).

The key aspects to remember when calling FCM are:

- An **Authorization** header with a value of **key=<YOUR_API_KEY>** must be set when you call the API, where **<YOUR_API_KEY>** is the API key from Firebase project.
 - The API key is used by FCM to find the appropriate sender ID, ensure the user has given permission for your project and finally ensuring that the server's IP address is whitelisted for that project.
- An appropriate **Content-Type** header of **application/json** or **application/x-www-form-urlencoded; charset=UTF-8** depending on whether you send the data as JSON or form data.
- An array of **registration_ids** - these are the registration ID's you'd extract from the endpoints from your users.

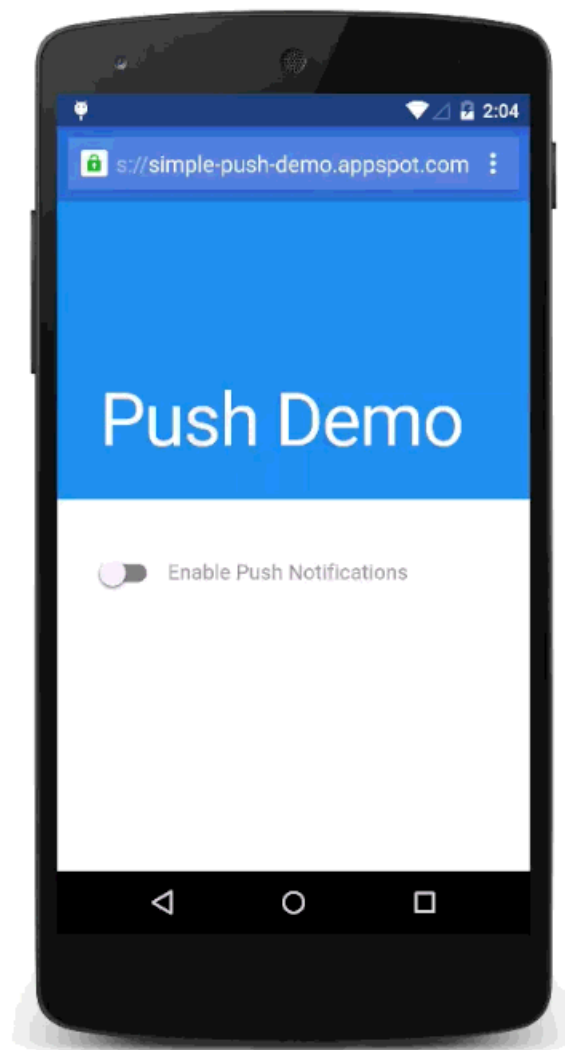
Please do [check out the docs](#) about how to send push messages from your server, but for a quick sanity check of your service worker you can use [cURL](#) to send a push message to your browser.

Swap out the **<YOUR_API_KEY>** and **<YOUR_REGISTRATION_ID>** in this cURL command with your own and run it from a terminal.

You should see a glorious notification:

```
curl --header "Authorization: key=<YOUR_API_KEY>" --header
"Content-Type: application/json" https://android.googleapis.com/gcm/send -d
"{\"registration_ids\": [\"<YOUR_REGISTRATION_ID>\"]}"
```





When developing your backend logic, remember that the Authorization header and format of the POST body are specific to the FCM endpoint, so detect when the endpoint is for FCM and conditionally add the header and format the POST body. For other browsers (and hopefully Chrome in the future) you'll need to implement the [Web Push Protocol](#).

A downside to the current implementation of the Push API in Chrome is that you can't send any data with a push message. Nope, nothing. The reason for this is that in a future implementation, payload data will have to be encrypted on your server before it's sent to a push messaging endpoint. This way the endpoint, whatever push provider it is, will not be able to easily view the content of the push message. This also protects against other vulnerabilities like poor validation of HTTPS certificates and man-in-the-middle attacks between your server and the push provider. However, this encryption isn't supported yet, so in the meantime you'll need to perform a fetch to get information needed to populate a notification.

A More Complete Push Event Example

The notification we've seen so far is pretty basic and as far as samples go, it's pretty poor at covering a real world use case.

Realistically, most people will want to get some information from their server before displaying the notification. This may be data to populate the notification title and message with something specific, or going a step further and caching some pages or data so that when the user clicks on the notification, everything is immediately available when the browser is opened—even if the network isn't available at that time.

In the following code we fetch some data from an API, convert the response to an object and use it to populate our notification.

```
self.addEventListener('push', function(event) {  
  // Since there is no payload data with the first version  
  // of push messages, we'll grab some data from  
  // an API and use it to populate a notification  
  event.waitUntil(  
    fetch(SOME_API_ENDPOINT).then(function(response) {  
      if (response.status !== 200) {  
        // Either show a message to the user explaining the error  
        // or enter a generic message and handle the  
        // onnotificationclick event to direct the user to a web page  
        console.log('Looks like there was a problem. Status Code: ' + response.st  
        throw new Error();  
      }  
  
      // Examine the text in the response  
      return response.json().then(function(data) {  
        if (data.error || !data.notification) {  
          console.error('The API returned an error.', data.error);  
          throw new Error();  
        }  
  
        var title = data.notification.title;  
        var message = data.notification.message;  
        var icon = data.notification.icon;  
        var notificationTag = data.notification.tag;  
  
        return self.registration.showNotification(title, {  
          body: message,  
          icon: icon,  
          tag: notificationTag  
        });  
      });  
    });  
  });
```

```

    }).catch(function(err) {
      console.error('Unable to retrieve data', err);

      var title = 'An error occurred';
      var message = 'We were unable to get the information for this push message';
      var icon = URL_TO_DEFAULT_ICON;
      var notificationTag = 'notification-error';
      return self.registration.showNotification(title, {
        body: message,
        icon: icon,
        tag: notificationTag
      });
    })
  );
});

```

It's worth, once again, highlighting that the **event.waitUntil()** takes a promise which results in the promise returned by **showNotification()**, meaning that our event listener won't exit until the asynchronous `fetch()` call is complete, and the notification is shown.

You'll notice that we show a notification even when there is an error. This is because if we don't, Chrome will show it's own generic notification.

Opening a URL when the User Clicks a Notification

When the user clicks a notification, a **notificationclick** event is dispatched in your service worker. Within your handler, you can take appropriate action, like focusing a tab or opening a window with a particular URL:

```

self.addEventListener('notificationclick', function(event) {
  console.log('On notification click: ', event.notification.tag);
  // Android doesn't close the notification when you click on it
  // See: http://crbug.com/463146
  event.notification.close();

  // This looks to see if the current is already open and
  // focuses if it is
  event.waitUntil(
    clients.matchAll({
      type: "window"
    })
    .then(function(clientList) {
      for (var i = 0; i < clientList.length; i++) {
        var client = clientList[i];
        if (client.url == '/' && 'focus' in client)
          return client.focus();
      }
    })
  );
});

```




```

    }
    if (clients.openWindow) {
        return clients.openWindow('/');
    }
    })
);
});

```

This example opens the browser to the root of the site's origin, by focusing an existing same-origin tab if one exists, and otherwise opening a new one.

There's a post dedicated to some of the things [you can do with the Notification API here](#).

Unsubscribe a User's Device

You've subscribed a user's device and they're receiving push messages, but how can you unsubscribe them?

The main things required to unsubscribe a users device is to call the **unsubscribe()** method on the [PushSubscription](#) object and to remove the endpoint from your servers (just so you aren't sending push messages which you know won't be received). The code below does exactly this:

```

function unsubscribe() {
    var pushButton = document.querySelector('.js-push-button');
    pushButton.disabled = true;

    navigator.serviceWorker.ready.then(function(serviceWorkerRegistration) {
        // To unsubscribe from push messaging, you need get the
        // subscription object, which you can call unsubscribe() on.
        serviceWorkerRegistration.pushManager.getSubscription().then(
            function(pushSubscription) {
                // Check we have a subscription to unsubscribe
                if (!pushSubscription) {
                    // No subscription object, so set the state
                    // to allow the user to subscribe to push
                    isPushEnabled = false;
                    pushButton.disabled = false;
                    pushButton.textContent = 'Enable Push Messages';
                    return;
                }

                var subscriptionId = pushSubscription.subscriptionId;
                // TODO: Make a request to your server to remove
                // the subscriptionId from your data store so you
                // don't attempt to send them push messages anymore
            }
        );
    });
}

```



```

// We have a subscription, so call unsubscribe on it
pushSubscription.unsubscribe().then(function(successful) {
    pushButton.disabled = false;
    pushButton.textContent = 'Enable Push Messages';
    isPushEnabled = false;
}).catch(function(e) {
    // We failed to unsubscribe, this can lead to
    // an unusual state, so may be best to remove
    // the users data from your data store and
    // inform the user that you have done so

    console.log('Unsubscription error: ', e);
    pushButton.disabled = false;
    pushButton.textContent = 'Enable Push Messages';
});
}).catch(function(e) {
    console.error('Error thrown while unsubscribing from push messaging.', e)
});
});
}

```

Keeping the Subscription Up to Date

Subscriptions may get out of sync between FCM and your server. Make sure your server parses the response body of the FCM API's send POST, looking for **error:NotRegistered** and **canonical_id** results, as explained in the [FCM documentation](#).

Subscriptions may also get out of sync between the service worker and your server. For example, after subscribing/unsubscribing successfully, a flaky network connection may prevent you from updating your server; or a user might revoke notifications permission, which triggers an automatic unsubscribe. Handle such cases by checking the result of **serviceWorkerRegistration.pushManager.getSubscription()** periodically (e.g. on page load) and synchronizing it with the server. You may also wish to re-subscribe automatically if you no longer have a subscription and `Notification.permission == 'granted'`.

In **sendSubscriptionToServer()** you will need to consider how you handle failed network requests when updating the **endpoint**. One solution is to track the state of the **endpoint** in a cookie to determine whether your server needs the latest details or not.

All of the above steps results in a full implementation of push messaging on the web in Chrome 46. There are still spec'd features that will make things easier (like a standard API for triggering push messages), but this release enables you to start building push messaging into your web apps today.

How to Debug Your Web App

While implementing push messages, bugs will live in one of two places: your page or your service worker.

Bugs in the page can be debugged using DevTools. To debug service worker issues, you have two options:

1. Go to **chrome://inspect > Service workers**. This view doesn't provide much information other than the currently running service workers.
2. Go to **chrome://serviceworker-internals** and from here you can view the state of service workers, and see errors, if there are any. This page is temporary until DevTools has a similar feature set.

One of the best tips I can give to anyone who is new to service workers is make use of the checkbox called "Open DevTools window and pause JavaScript execution on service worker startup for debugging." This checkbox will add a breakpoint at the start of your service worker and **pause execution**, this allows you to resume or step through your service worker script and see if you hit any problems.



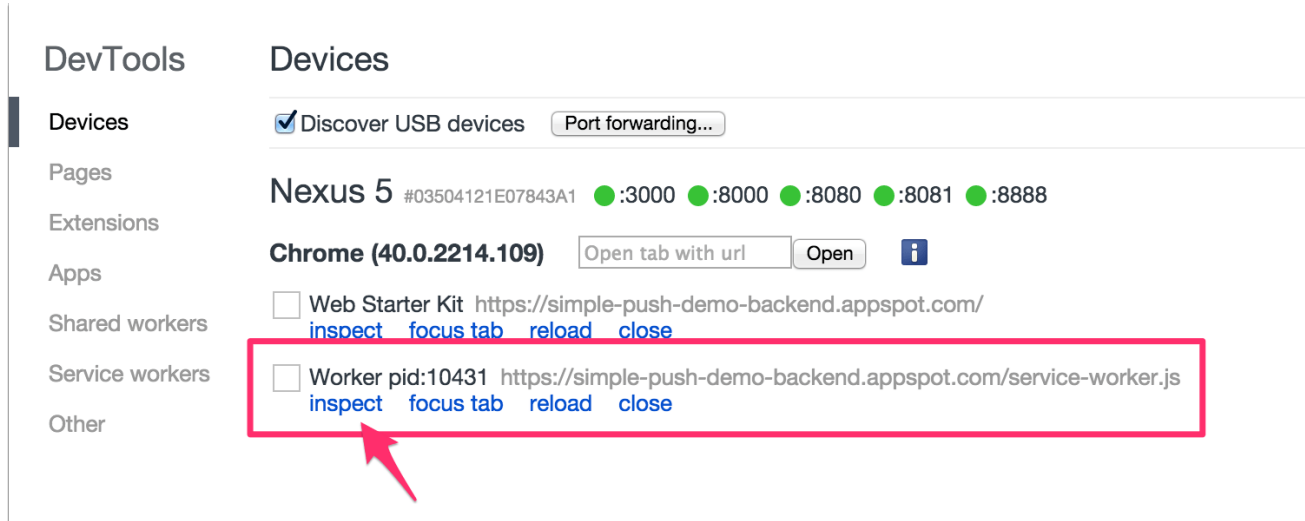
If there seems to be an issue between FCM and your service worker's push event, then there isn't much you can do to debug the problem since there is no way for you to see whether Chrome received anything. The key thing to ensure is that the response from FCM is successful when your server makes an API call. It'll look something like:

```
{"multicast_id":1234567890,"success":1,"failure":0,"canonical_ids":0,"result":{}}
```

Notice the "success": 1 response. If you see a failure instead, then that suggests that something isn't right with the FCM registration ID and the push message isn't getting sent to Chrome.

Debugging Service Workers on Chrome for Android

At the moment debugging service workers on Chrome for Android is not obvious. You need to navigate to **chrome://inspect**, find your device and look for a list item with the name "Worker pid:...." which has the URL of your service worker.



UX for Push Notifications

The Chrome team has been putting together a document of best practices for push notifications UX as well as a document covering some of the edge cases when working with push notifications.

- [Best Practices for Push Notifications Permissions UX](#)
- [Push Notifications Edge Cases and Mitigations](#)

Future of Push Messaging on Chrome and the Open Web

This section goes into a little bit of detail surrounding some of the Chrome specific parts of this implementation that you should be aware of and how it will differ from other browser implementations.

Web Push Protocol and Endpoints

The beauty of the Push API standard is that you should be able to take the **endpoint**, pass them to your server and send push messages by implementing the [Web Push Protocol](#).

The Web Push Protocol is a new standard which push providers can implement, allowing developers to not have to worry about who the push provider is. The idea is that this avoids the need to sign up for API keys and send specially formatted data, like you have to with FCM.

Chrome was the first browser to implement the Push API and FCM does not support the Web Push Protocol, which is the reason why Chrome requires the **gcm_sender_id** and you need to use the restful API for FCM.

The end goal for Chrome is to move towards using the Web Push Protocol with Chrome and FCM.

Until then, you need to detect the endpoint "https://android.googleapis.com/gcm/send" and handle it separately from other endpoints, i.e. format the payload data in a specific way and add the Authorization key.

How to Implement the Web Push Protocol?

Firefox Nightly is currently working on push and will likely be the first browser to implement the Web Push Protocol.

FAQs

Where are the specs?

https://slightlyoff.github.io/ServiceWorker/spec/service_worker/ 

<https://w3c.github.io/push-api/>

<https://notifications.spec.whatwg.org/> 

Can I prevent duplicate notifications if my web presence has multiple origins, or if I have both a web and native presence?

There isn't a solution to this at the moment, but you can follow progress [on Chromium](#).

The ideal scenario would be to have some kind of ID for a users device and then on the server side match up the native app and web app subscription ID's and decide which one to send a push message to. You could do this via screen size, device model, sharing a generated key between the web app and native app, but each approach has pro's and con's.

Why do I need a gcm_sender_id?

This is required so that Chrome, Opera for Android and the Samsung Browser can use the Firebase Cloud Messaging (FCM) API. The goal is to use the Web Push Protocol when the standard is finalized and FCM can support it.

Why not use Web Sockets or Server-Sent Events (EventSource)?

The advantage of using push messages is that even if your page is closed, your service worker will be woken up and be able to show a notification. Web Sockets and EventSource have their connection closed when the page or browser is closed.

What if I don't need background event delivery?

If you don't need background delivery then Web Sockets are a great option.

When can I use push without showing notifications (i.e. silent background push)?

There is no timeline for when this will be available yet, but there is an intent to implement background sync and while it's not decided or spec'd, there is some discussion of enabling silent push with background sync.

Why does this require HTTPS? How do I work around this during development?

Service workers require secure origins to ensure that the service worker script is from the intended origin and hasn't come about from a man-in-the-middle attack. Currently, that means using HTTPS on live sites, though localhost will work during development.

What does browser support look like?

Chrome supports in its stable version and Mozilla have push being worked on in Firefox Nightly. See the implementing the Push API bug for more info and you can track their Notification implementation here.

Can I remove a notification after a certain time period?

At the moment this isn't possible but we are planning on adding support to get a list of currently visible notifications. If you have a use case to set an expiration for notification after it's displayed created, we'd love to know what that is, so please add a comment and we'll pass it back to the Chrome team.

If you only need to stop a push notification from being sent to the user after a certain time period, and don't care how long the notification stays visible, then you can use FCM's time to live (ttl) parameter, [learn more here](#).

What are the limitations of push messaging in Chrome?

There are a few limitations outlined in this post:

- Chrome's usage of CCM as a push service creates a number of proprietary requirements. We're working together to see if some of these can be lifted in the future.
- You have to show a notification when you receive a push message.
- Chrome on desktop has the caveat that if Chrome isn't running, push messages won't be received. This differs from Chrome OS and Android where push messages will always be received.

Shouldn't we be using the Permissions API?

The [Permission API](#) [\[7\]](#) is implemented in Chrome, but it's not necessarily going to be available in all browsers. [You can learn more here](#).

Why doesn't Chrome open up the previous tab when I click a notification?

This issue only affects pages which aren't currently controlled by a service worker. You can [learn more here](#).

What if a notification is out of date by the time the users device received the push?

You always have to show a notification when you receive a push message. In the scenario where you want to send a notification but it's only useful for a certain period time, you can use the 'time_to_live' parameter on CCM so that FCM won't send the push message if it passes the expiry time.

[More details can be found here](#).

What happens if I send 10 push messages but only want the device to receive one?

FCM has a 'collapse_key' parameter you can use to tell FCM to replace any pending message which has the same 'collapse_key', with the new message.

[More details can be found here.](#)

Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](#), and code samples are licensed under the [Apache 2.0 License](#). For details, see our [Site Policies](#). Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 2, 2018.