

Introduction

Introduction

This book is for anyone looking to learn about push notifications on the web.

I started working on web push when Chrome first added support. There have been quite a few changes since then but the API's have stabilised across Chrome and Firefox so it's the perfect time for people to learn about push on the web.

This site / book covers everything I've learnt and will be updated with anything new I learn in the future.

If you spot mistakes, code errors or simple typos please [raise issues](#) or [contribute on Github](#) where this [content lives](#).

To track updates of this content follow the [Releases on Github](#), which also has an [RSS feed](#).

Finally, a massive thank you to [all the contributors](#) who have helped with this book.

How Push Works

Before getting into the API, let's look at push from a highlevel, start to finish. Then as we step through individual topics or API's later on, you'll have an idea of how and why it's important.

The three key steps when implementing push are:

1. Adding the client side logic to subscribe a user to push (i.e. the JavaScript and UI in your web app that registers a user to push messages).
2. The API call from your back-end / application that triggers a push message to a users device.
3. The service worker JavaScript file that will receive a “push event” when the push arrives on the device. It's in this JavaScript that you'll be able to show a notification.

Let's look at what each of these steps entails in a little more detail.

Step 1: Client Side

The first step is to “subscribe” a user to push messaging.

Subscribing a user requires two things. First, getting **permission** from the user to send them push messages. Second, getting a **PushSubscription** from the browser.

A **PushSubscription** contains all the information we need to send a push message to that user. You can “kind of” think of this as a ID for that user’s device.

This is all done in JavaScript with the [Push API](#).

Before subscribing a user you’ll need to generate a set of “application server keys”, which we’ll cover later on.

The application server keys, also known as VAPID keys, are unique to your server. They allow a push service to know which application server subscribed a user and ensure that it’s the same server triggering the push messages to that user.

Once you’ve subscribed the user and have a **PushSubscription**, you’ll need to send the **PushSubscription** details to your backend / server. On your server, you’ll save this subscription to a database and use it to send a push message to that user.



Figure 1: Make sure you send the PushSubscription to your backend.

Step 2: Send a Push Message

When you want to send a push message to your users you need to make an API call to a push service. This API call would include what data to send, who to send the message to and any criteria about how to send the message.

Some questions you might be asking yourself:

- Who and what is the push service?
- What does the API look like? Is it JSON, XML, something else?
- What can the API do?

Who and What is the Push Service?

A push service has the job of receiving requests and delivering push messages if the request is valid.

Each browser can use any push service they want, it's something developers have no control over. This isn't a problem because every push service expects the **same** API call. Meaning you don't have to care who the push service is, you just need to make sure that your API call is valid.

To get the appropriate URL to trigger a push message (i.e. the URL for the push service) you just need to look at the `endpoint` value in a `PushSubscription`.

Below is an example of the values you'll get from a `PushSubscription`:

```
{  
  "endpoint": "https://random-push-service.com/some-kind-of-unique-id-1234/v2/",  
  "keys": {  
    "p256dh" : "BNcRdreALRFXTk0OUHK1EtK2wtaz5Ry4YfYCA_0QTpQtUbVlUls0VJXg7A8u-Ts1XbjhazAk  
    "auth"   : "tBHIItJI5svbpez7KI4CCXg=="  
  }  
}
```

The `endpoint` in this case is `https://random-push-service.com/some-kind-of-unique-id-1234/v2/`. The push service would be 'random-push-service.com' and each endpoint is unique to a user, indicated with 'some-kind-of-unique-id-1234'. As you start working with push you'll notice this pattern.

We'll cover what the `keys` value is for later on.

What does the API look like?

I mentioned that every web push service expects the same API call. That API is the [Web Push Protocol](#).

It's an IETF standard that defines how you make an API call to a push service.

The API call requires certain headers to be set and the data to be a stream of bytes. We'll look at libraries that can perform this API call for us as well as how to do it ourselves.

What can the API do?

The API provides a way to send a message to a user, with / without data, and provide instructions of *how* to send the message.

The data you send with a push message must be encrypted. The reason for this is that it prevents push services, who could be anyone, from being able to

view the data sent with the push message. This is important given that it's the browser who decides which push service to use, which could open the door to browsers using a push service that isn't safe or secure.

When you trigger a push message, the push service will receive the API call and queue the message. This message will remain queued until the user's device comes online and the push service can deliver the messages. The instructions you can give to the push service define how the push message is queued.

The instructions include details like:

- The time-to-live for a push message. This defines how long a message should be queued before it's removed and not delivered.
- Define the urgency of the message. This is useful in case the push service is preserving the user's battery life by only delivering high priority messages.
- Give a push message a "topic" name which will replace any pending message with this new message.

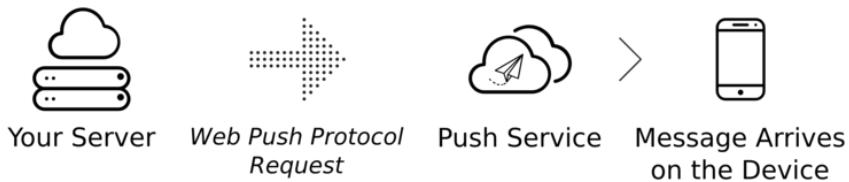


Figure 2: When your server wishes to send a push message, it makes a web push protocol request to a push service.

Step 3: Push Event on the Users Device

Once we've sent a push message, the push service will keep your message on its server until one of the following events occur:

1. The device comes online and the push service can deliver the message.
2. The message expires. If this occurs the push service removes the message from its queue and it'll never be delivered.

When the push service does deliver a message, the browser will receive the message, decrypt any data and dispatch a **push** event in your service worker.

A **service worker** is a "special" JavaScript file. The browser can execute this JavaScript without your page being open. It can even execute this JavaScript

when the browser is closed. A service worker also has API's, like push, that aren't available in the web page (i.e. API's that aren't available out of a service worker script).

It's inside the service worker's 'push' event that you can perform any background tasks. You can make analytics calls, cache pages offline and show notifications.



Figure 3: When a push message is sent from a push service to a users device, your service worker receives a push event.

That's the whole flow for push messaging. Lets go through each step in more detail.

Subscribing a User

The first step is to get permission from the user to send them push messages and then we can get our hands on a `PushSubscription`.

The JavaScript API to do this is reasonably straight forward, so let's step through the logic flow.

Feature Detection

First we need check if the current browser actually supports push messaging. We can check if push is supported with two simple checks.

1. Check the `serviceWorker` API on the `navigator`.
2. Check `PushManager` on the `window`.

```
if (!('serviceWorker' in navigator)) {  
  // Service Worker isn't supported on this browser, disable or hide UI.  
  return;  
}
```

```

if (!('PushManager' in window)) {
    // Push isn't supported on this browser, disable or hide UI.
    return;
}

```

While browser support is growing quickly for both service worker and push messaging support, it's always a good idea to feature detect for both and [progressively enhance](#).

Register a Service Worker

With the feature detect we know that both service workers and Push is supported. The next step is to “register” our service worker.

When we register a service worker, we are telling the browser where our service worker file is. The file is still just JavaScript, but the browser will “give it access” to the service worker APIs, including push. To be more exact, the browser runs the file in a service worker environment.

To register a service worker, call `navigator.serviceWorker.register()`, passing in the path to our file. Like so:

```

function registerServiceWorker() {
    return navigator.serviceWorker.register('service-worker.js')
        .then(function(registration) {
            console.log('Service worker successfully registered.');
            return registration;
        })
        .catch(function(err) {
            console.error('Unable to register service worker.', err);
        });
}

```

This code above tells the browser that we have a service worker file and where its located. In this case, the service worker file is at `/service-worker.js`. Behind the scenes the browser will take the following steps after calling `register()`:

1. Download the service worker file.
2. Run the JavaScript.
3. If everything ran correctly and there were no errors, the promise returned by `register()` will resolve. If there are errors of any kind, the promise will reject.

If `register()` does reject, double check your JavaScript for typos / errors in Chrome DevTools.

When `register()` does resolve, it returns a `ServiceWorkerRegistration`. We'll use this registration to access to the [PushManager API](#).

Requesting Permission

We've registered our service worker and ready to subscribe the user, the next step is to get permission from the user to send them push messages.

The API for getting permission is relatively simple, the downside is that the API [recently changed from taking a callback to returning a Promise](#). The problem with this, is that we can't tell what version of the API is implemented by the current browser, so you have to implement both and handle both.

```
function askPermission() {
  return new Promise(function(resolve, reject) {
    const permissionResult = Notification.requestPermission(function(result) {
      resolve(result);
    });

    if (permissionResult) {
      permissionResult.then(resolve, reject);
    }
  })
  .then(function(permissionResult) {
    if (permissionResult !== 'granted') {
      throw new Error('We weren\'t granted permission.');
    }
  });
}
```

In the above code, the important snippet of code is the call to `Notification.requestPermission()`. This method will display a prompt to the user:

Once the permission has been accepted / allowed, closed (i.e. clicking the cross on the pop-up) or blocked, we'll be given the result as a string: 'granted', 'default' or 'denied'.

In the sample code above, the promise returned by `askPermission()` resolves if the permission is granted, otherwise we throw an error making the promise reject.

One edge case that you need to handle is if the user clicks the 'Block' button. If this happens, your web app will not be able to ask the user for permission again.

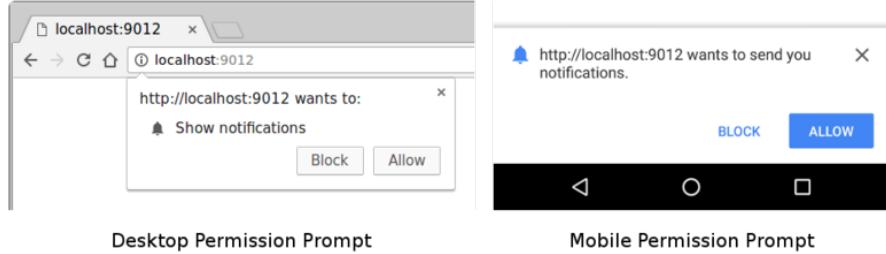


Figure 4: Permission Prompt on Desktop and Mobile Chrome.

They'll have to manually "unblock" your app by changing the permission state of your web app, which is buried in a settings panel. Think carefully about how and when you ask the user for permission, because if they click block, it's not an easy way to reverse that decision.

The good news is that most users are happy to give permission as long as it's asked in a way that they *know* why the permission is being asked.

We'll look at how some popular sites ask for permission later on.

Subscribe a User with PushManager

Once we have our service worker registered and we've got permission, we can subscribe a user by calling `registration.pushManager.subscribe()`.

```
function subscribeUserToPush() {
  return getSWRegistration()
    .then(function(registration) {
      const subscribeOptions = {
        userVisibleOnly: true,
        applicationServerKey: urlBase64ToInt8Array(
          'BE162iUYgUivxIkv69yViEuiBIa-Ib9-SkvMeAtA3LFgDzkrxZJjSgSnfckjBJuBkr3qBUYIHBQFLXY'
        )
      };
      return registration.pushManager.subscribe(subscribeOptions);
    })
    .then(function(pushSubscription) {
      console.log('Received PushSubscription: ', JSON.stringify(pushSubscription));
      return pushSubscription;
    });
}
```

When calling the `subscribe()` method, we pass in an *options* object, which consists of both required and optional parameters.

Lets look at all the options we can pass in.

userVisibleOnly Options

When push was first added to browsers, there was uncertainty about whether developers should be able to send a push message and not show a notification. This is commonly referred to as silent push, due to the user not knowing that something had happened in the background.

The concern was that developers could do nasty things like track a user's location on an ongoing basis without the user knowing.

To avoid this scenario and to give spec authors time to consider how best to support this feature, the `userVisibleOnly` option was added and passing in a value of `true` is a symbolic agreement with the browser that the web app will show a notification every time a push is received (i.e. no silent push).

At the moment you **must** pass in a value of `true`. If you don't include the `userVisibleOnly` key or pass in `false` you'll get the following error:

Chrome currently only supports the Push API for subscriptions that will result in user-visible messages. You can indicate this by calling `pushManager.subscribe({userVisibleOnly: true})` instead. See <https://goo.gl/yqv4Q4> for more details.

It's currently looking like blanket silent push will never be implemented in Chrome. Instead, spec authors are exploring the notion of a budget API which will allow web apps a certain number of silent push messages based on the usage of a web app.

Firefox doesn't require `userVisibleOnly`, however Firefox does have some notion of a budget behind the scenes but there isn't much detail beyond it expiring `PushSubscription`'s if the budget it gets too low. This is most likely calculated by the number of interactions a user has with notifications displayed.

applicationServerKey Option

We briefly mentioned the notion "application server keys" in the previous section. "Application server keys" are used by a push service to identify the application subscribing a user and ensuring that the same application is messaging that user.

Application server keys are a public and private key pair that are unique to your application. The private key should be kept a secret to your application and the public key can be shared freely.

The `applicationServerKey` option passed into the `subscribe()` call is the applications public key. The browser passes this onto a push service when subscribing the user, meaning the push service can tie your applications public key to the users PushSubscription.

The diagram below illustrates these steps.

1. Your web app is loaded in a browser and you call `subscribe()` passing in your public application server key.
2. The browser then makes a network request to a push service who will generate an endpoint, associate this endpoint with the applications public key and return the endpoint to the browser.
3. The browser will add this endpoint to the PushSubscription which is returned via the `subscribe()` promise.

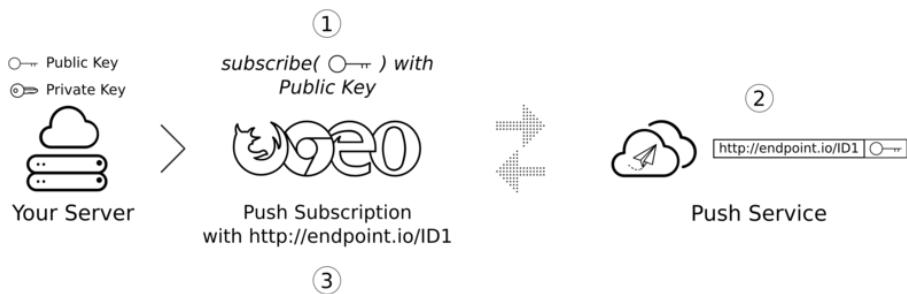


Figure 5: Illustration of the public application server key is used in subscribe method.

When you later want to send a push message, you'll need to create an **Authorization** header which will contain information signed with your application server's **private key**. When the push service receives a request to send a push message, it can validate this signed **Authorization** header by looking up the public key linked to the endpoint receiving the request. If the signature is valid the push service knows that it must have come from the application server with the **matching private key**. It's basically a security measure that prevents anyone else sending messages to an applications users.

Technically, the `applicationServerKey` is optional. However, the easiest implementation on Chrome requires it and other browsers may require it in the future. It's optional on Firefox.

The specification that defines *what* the application server key should be is the [VAPID spec](#). Whenever you read something referring to “*application server keys*” or “*VAPID keys*”, just remember that they are the same thing.

Those are the only subscribe option.

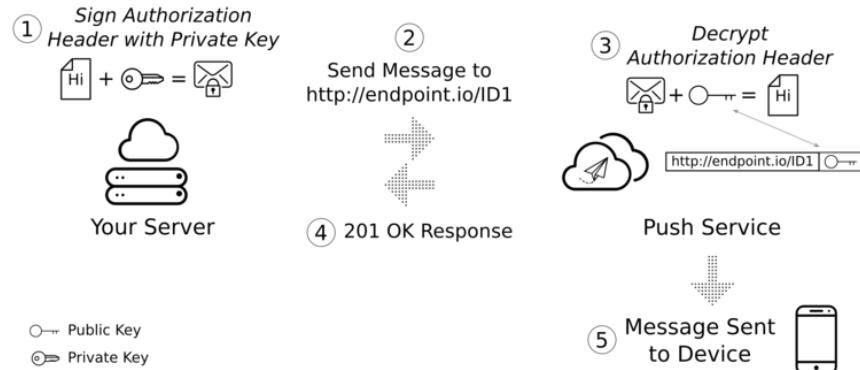


Figure 6: Illustration of how the private application server key is used when sending a message.

How to Create Application Server Keys You can create a public and private set of application server keys by visiting this web site web-push-codelab.appspot.com or you can use the [web-push](#) to generate keys by doing the following:

```
$ npm install -g web-push
$ web-push generate-vapid-keys
```

You only need to create these keys once for your application, just make sure you keep the private key private....yeah I just said that.

Permissions and subscribe()

There is one side effect of calling `subscribe()`. If your web app doesn't have permissions for showing notifications at the time of calling `subscribe()`, the browser will request the permissions for you. This is useful if your UI works with this flow, but if you want more control (and I think most developers will), stick to the `Notification.requestPermission()` API that we used earlier.

What is a PushSubscription?

We call `subscribe()`, pass in some options, in return we get a promise that resolves to a `PushSubscription` resulting in some code like so:

```
function subscribeUserToPush() {
  return getSWRegistration()
```

```

    .then(function(registration) {
      const subscribeOptions = {
        userVisibleOnly: true,
        applicationServerKey: urlBase64ToInt8Array(
          'BE162iUYgUivxIkv69yViEuiBIA-Ib9-SkvMeAtA3LFgDzkrxZJjSgSnfckjBJuBkr3qBUYIHBQFLXY'
        )
      };
      return registration.pushManager.subscribe(subscribeOptions);
    })
    .then(function(pushSubscription) {
      console.log('Received PushSubscription: ', JSON.stringify(pushSubscription));
      return pushSubscription;
    });
  }
}

```

The `PushSubscription` object contains all the required information needed to send a push messages to that user. If you print out the contents using `JSON.stringify()`, you'll see the following:

```
{
  "endpoint": "https://some.pushservice.com/something-unique",
  "keys": {
    "p256dh": "BIPUL12DLfytvTajnryr2PRdAgXS3HGKiLqndGcJGabyhHheJY1NGCeXl1dn18gSJ1WAkAPIx",
    "auth": "FPssNDTKnInHVndSTdbKFw=="
  }
}
```

The `endpoint` is the push services URL, you make a POST request to this URL to trigger a push message.

The `keys` object contains the values used to encrypt message data sent with a push message (which we'll discuss later on in this book).

Send a Subscription to Your Server

Once you have a push subscription you'll want to send it to your server. It's up to you how you do that but a tiny tip is to use `JSON.stringify()` to get all the necessary data out of the subscription object, otherwise you can piece together the same result manually:

```
const subscriptionObject = {
  endpoint: pushSubscription.endpoint,
  keys: {
```

```

    p256dh: pushSubscription.getKeys('p256dh'),
    auth: pushSubscription.getKeys('auth')
}
};

// The above is the same output as:

const subscriptionObjectToo = JSON.stringify(pushSubscription);

```

In [the demo used throughout this book](#), we make a POST request to send a subscription to our node server that stores the subscription in a database.

Sending the subscription is done in the web page like so:

```

function sendSubscriptionToBackEnd(subscription) {
  return fetch('/api/save-subscription/', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(subscription)
  })
  .then(function(response) {
    if (!response.ok) {
      throw new Error('Bad status code from server.');
    }

    return response.json();
  })
  .then(function(responseData) {
    if (!(responseData.data && responseData.data.success)) {
      throw new Error('Bad response from server.');
    }
  });
}

```

The node server receives this request and saves the data to a database for use later on.

```

app.post('/api/save-subscription/', function (req, res) {
  if (!isValidSaveRequest(req, res)) {
    return;
  }

  return saveSubscriptionToDatabase(req.body)
}

```

```

.then(function(subscriptionId) {
  res.setHeader('Content-Type', 'application/json');
  res.send(JSON.stringify({ data: { success: true } }));
})
.catch(function(err) {
  res.status(500);
  res.setHeader('Content-Type', 'application/json');
  res.send(JSON.stringify({
    error: {
      id: 'unable-to-save-subscription',
      message: 'The subscription was received but we were unable to save it to our database'
    }
  }));
});

```

With the `PushSubscription` details on our server we are good to send our user a message whenever we want.

FAQs

A few common questions people have asked at this point:

Can I change the push service a browser uses?

No. The push service is selected by the browser and as we saw with the `subscribe()` call, the browser will make network requests to the push service to retrieve the details that make up the `PushSubscription`.

Each browser uses a different Push Service, don't they have different API's?

All push services will expect the same API.

This common API is called the `Web Push Protocol` and describes the network request / API call your application will need to make to trigger a push message.

If I subscribe a user on their desktop, are they subscribed on their phone as well?

Unfortunately not. A user must register for push on each browser they wish to receive messages on. It's also worth noting that this will require the user granting permission on each device.

Permission UX

The natural step after getting a `PushSubscription` and saving it our server is to trigger a push message, but there is one thing I flagrantly glossed over. The user experience when asking for permission from the user to send them push messages.

Sadly, very few sites give much consideration as to how they ask their user for permission, so lets take a brief aside to look at both good and bad UX.

Common Patterns

There have been a few common patterns emerging that should guide / help you when deciding what is best for your users and use case.

Value Proposition

Encouraging a user to subscribe to push messaging at a time where the benefit is obvious given the current context is a fantastic way to get users to grant permission.

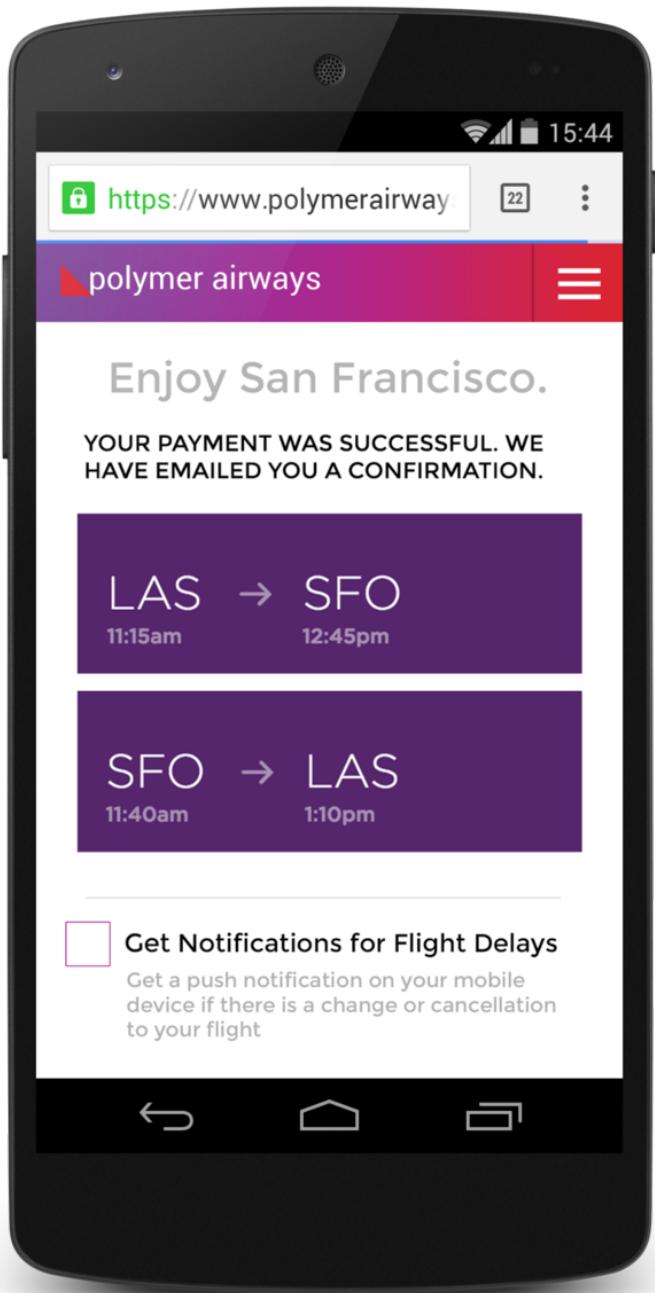
For example, a user has just bought an item on an online store and finished the checkout flow, the site can then offer updates on the delivery status.

There are a range of situations where this pattern / approach works: - A particular item is out of stock, would you like to be notified when it's next available? - This breaking news story will be regularly updated, would you like to be notified as the story develops? - You're the highest bidder, would you like to be notified if you are outbid?

These are all points where the user has invested in your service and there is a clear value proposition for them to enable push notifications.

[Owen Campbell-Moore](#) created a mock site for an airline that demonstrates this approach.

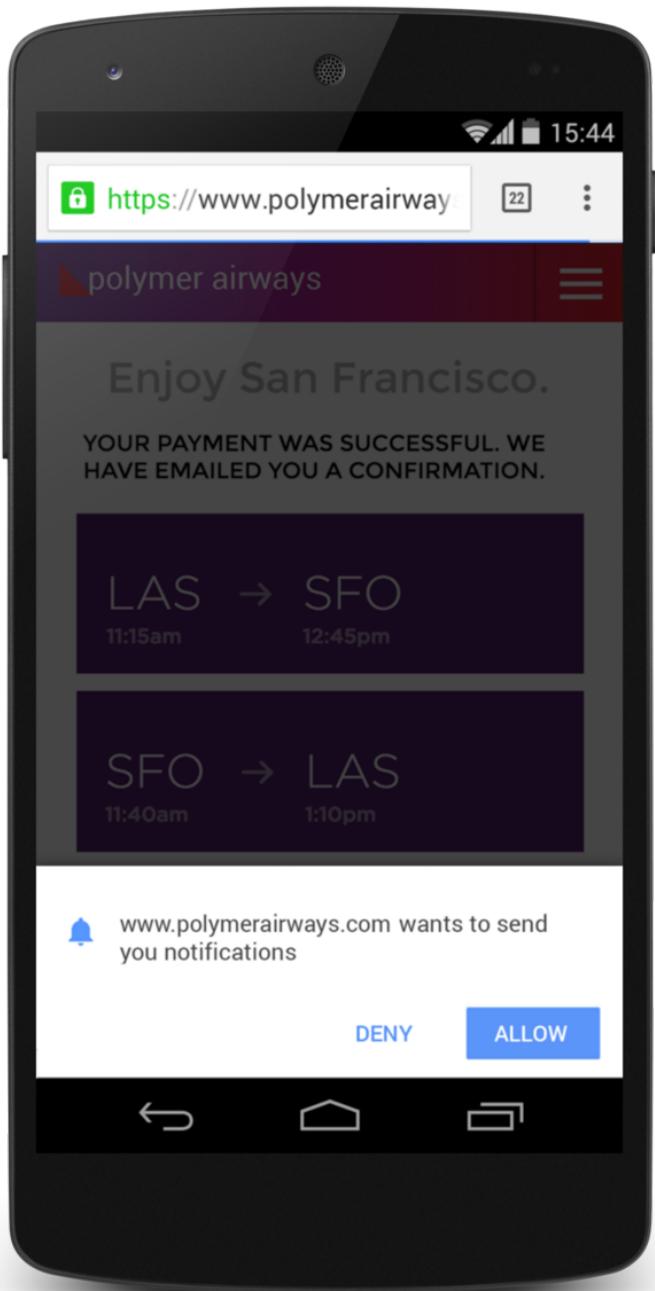
After the user has booked a flight asks if the user would like notifications in case of flight delays.



```
.device-image .center-image }
```

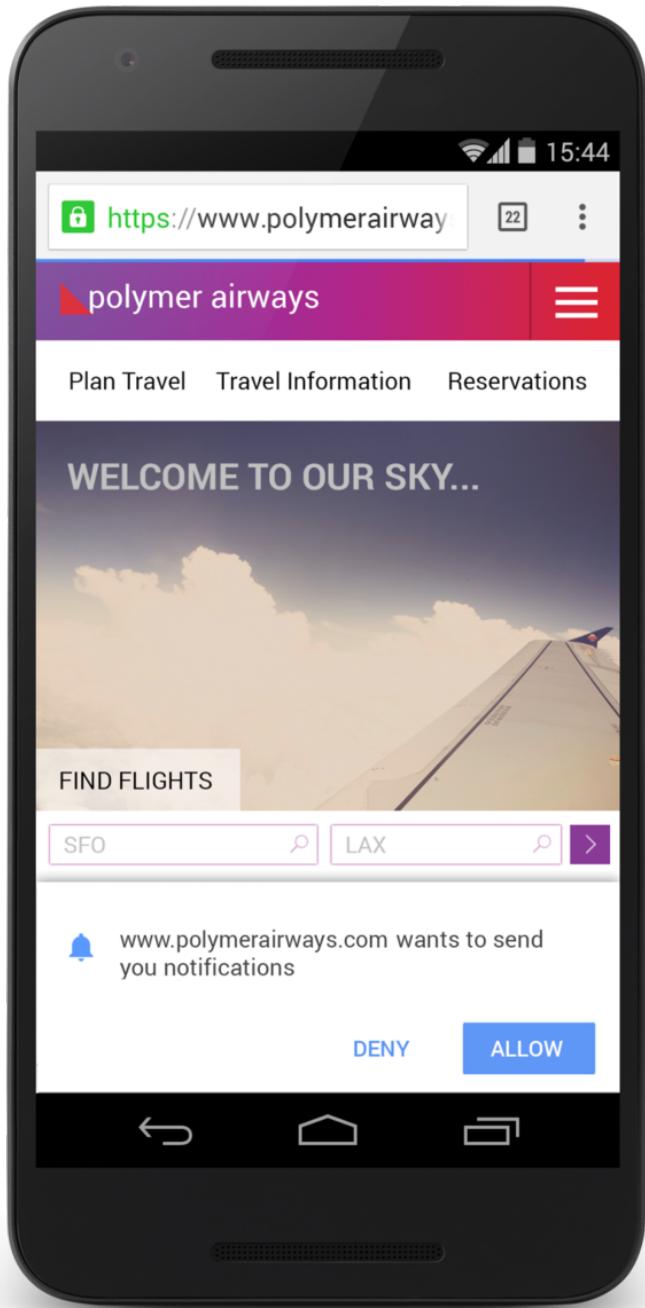
Note that this isn't the browser UI as well, this is a custom UI in the web site asking the user, allowing the site full control over the messaging to the user.

Another nice touch to Owen's demo is that if the user does click to enable notification, the site adds a semi-transparent overlay over the entire page when it shows the permission prompt. This helps draw the users attention to the permission prompt.



```
.device-image .center-image }
```

The alternative to this example, the **bad UX** for asking permission, is to request permission as soon as a user lands on the airline's site.



```
.device-image .center-image {
```

No context as to what the notifications are for, if the user wanted to get a task done (i.e. check a flight or book a flight), this permission prompt gets in the way of that and secondly it's not a good look having a pop-up over the site.

Double Permission

You may feel that your site has a clear need for push messaging and as a result want to ask the user for permission as soon as possible. An example of this sites that fit into this category are messaging sites or email clients. You receiving a message or email, show a notification.

In these cases, it's worth considering the double permission UX.

With this approach you display a custom permission prompt in your web app which asks the user to enable notifications. By doing this the user can say enable or disable without risk of being permanently blocked. If the user selects enable on the custom UI, display the actual permission prompt, otherwise hide your custom pop-up and ask some other time.

A good example of this is [Slack](#). They show a prompt at the top of their page once you've signed in asking if you'd like to enable notifications.

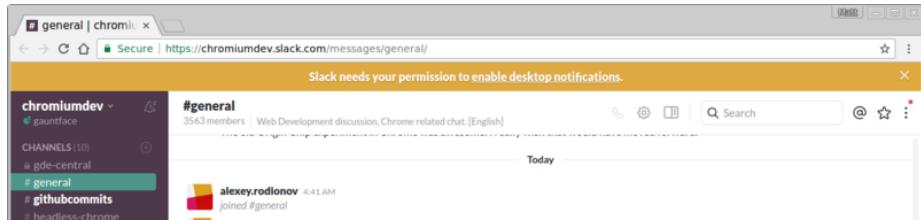


Figure 7: Example of slack.com showing custom banner for permission to show notifications.

If the user clicks accept, the actual permission prompt is shown:

I was also a big fan of Slacks first notification when you allow the permission.

Settings Panel

You can move notifications into a settings panel, giving users an easy way to enable and disable push messaging, without the need of cluttering your web app's UI.

A good example of this is [Google I/O's 2016 site](#). When you first load up the Google I/O site, you aren't asked to do anything, the user is left to explore the site.

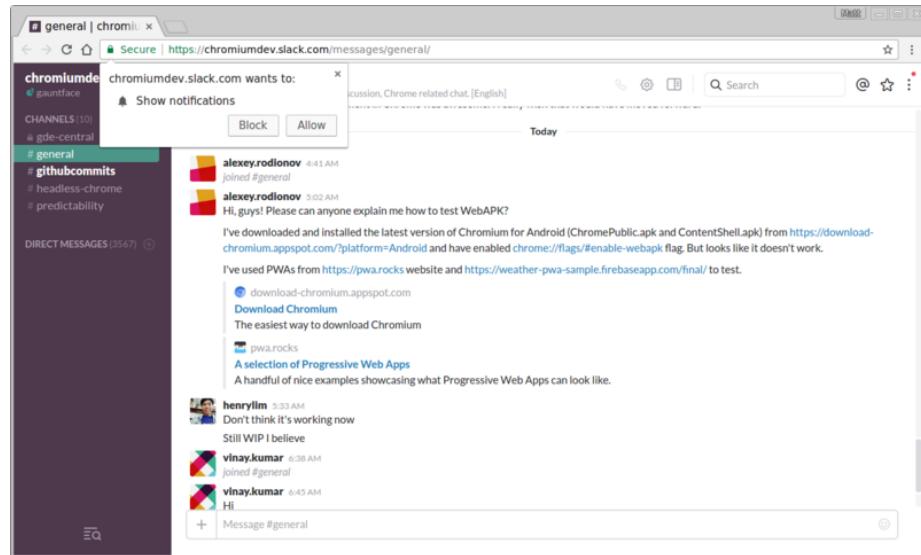


Figure 8: Actual permission prompt on slack.com.

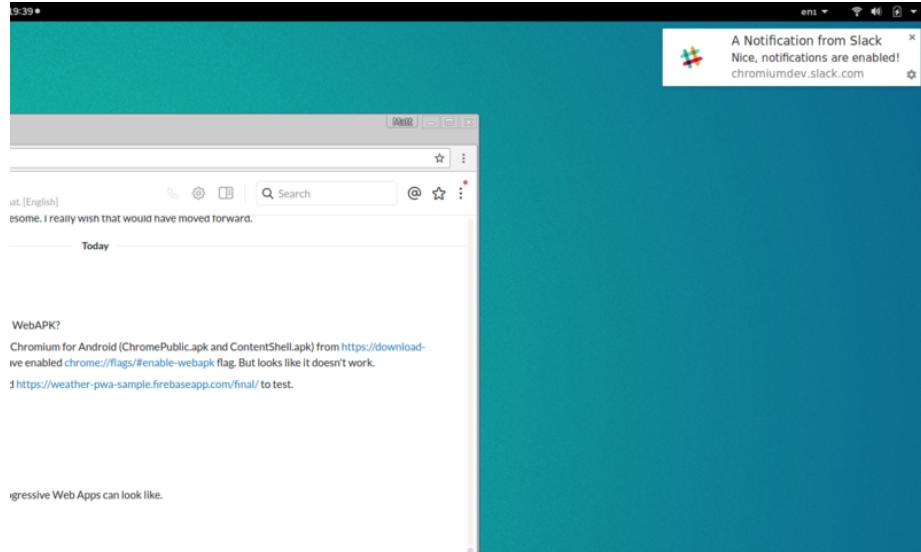
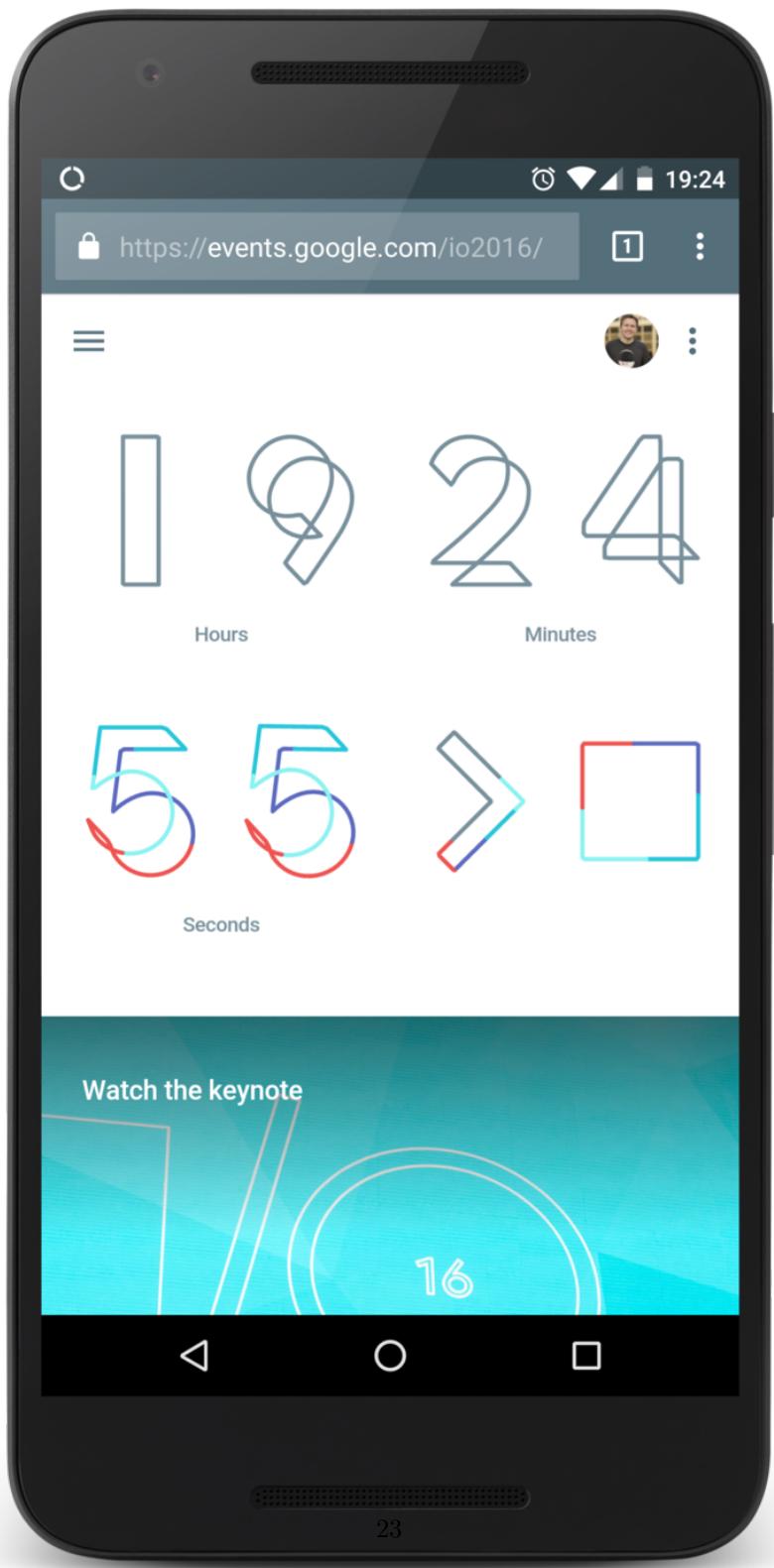


Figure 9: Cute “It’s working” notification from slack.com.

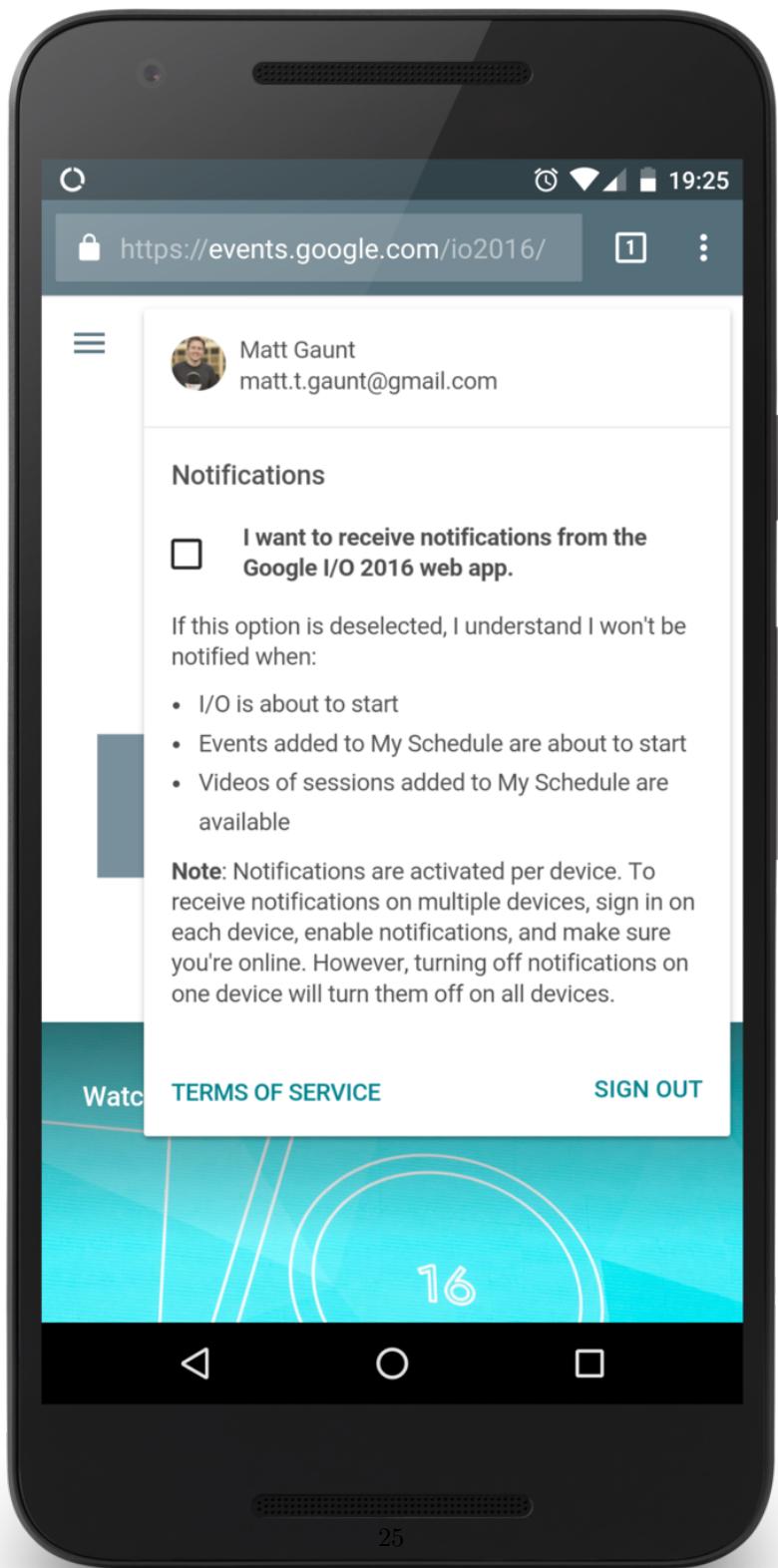


23

{:

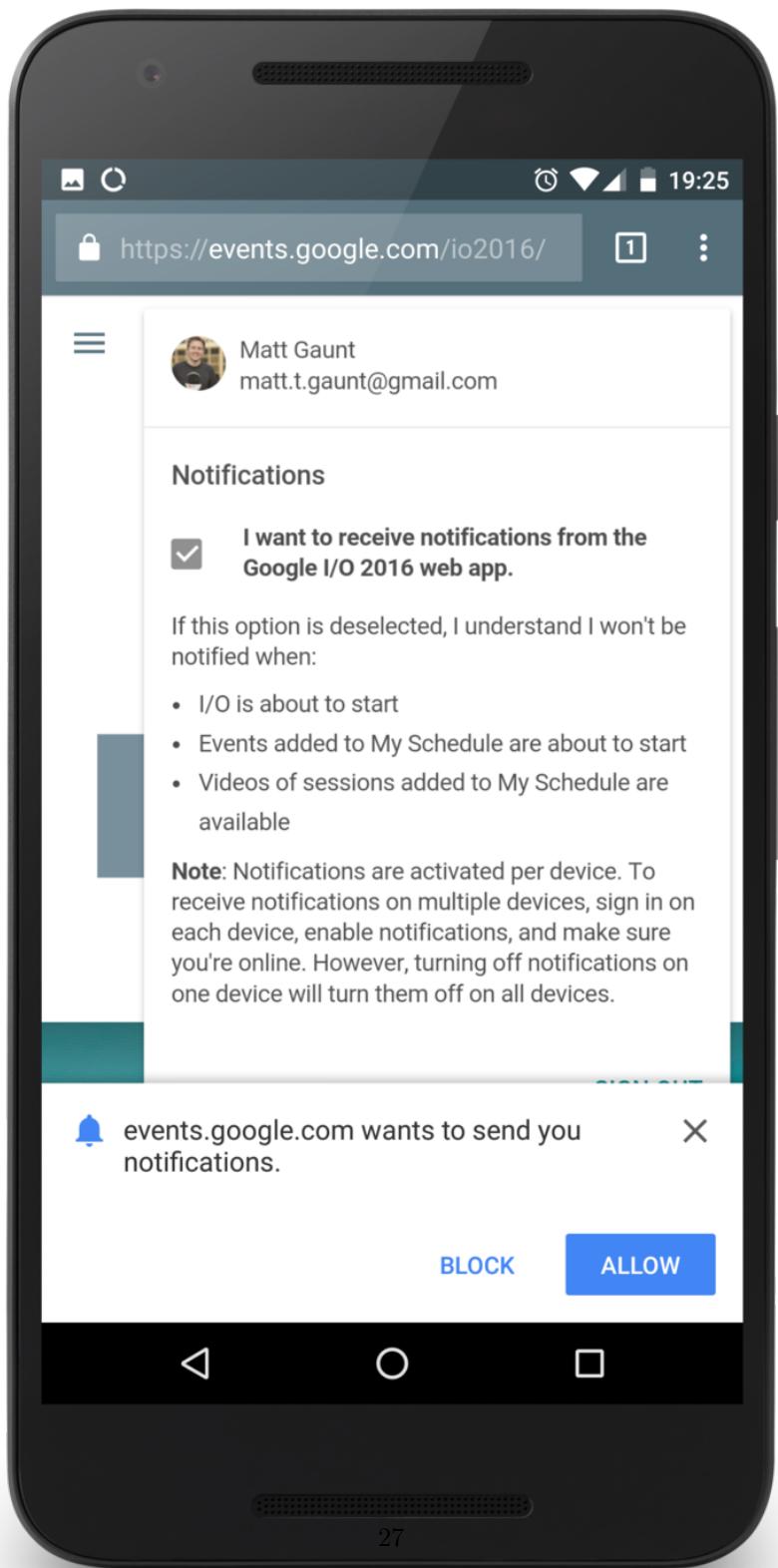
```
.device-image .center-image }
```

After a few visits of clicking the menu item on the right, a settings panel is reveal to the user, allowing them to set up and manage notifications.



```
.device-image .center-image }
```

Clicking on the checkbox displays the permission prompt, no hidden surprises.



```
.device-image .center-image {
```

After the permission has been granted the checkbox is checked and the user is good to go. The great thing about this UX is that the location to sign up for push is the same location to disable push.

Slack also does a good job at giving users control over their notifications. They offer a host of options allowing users to customise the notifications they receive.

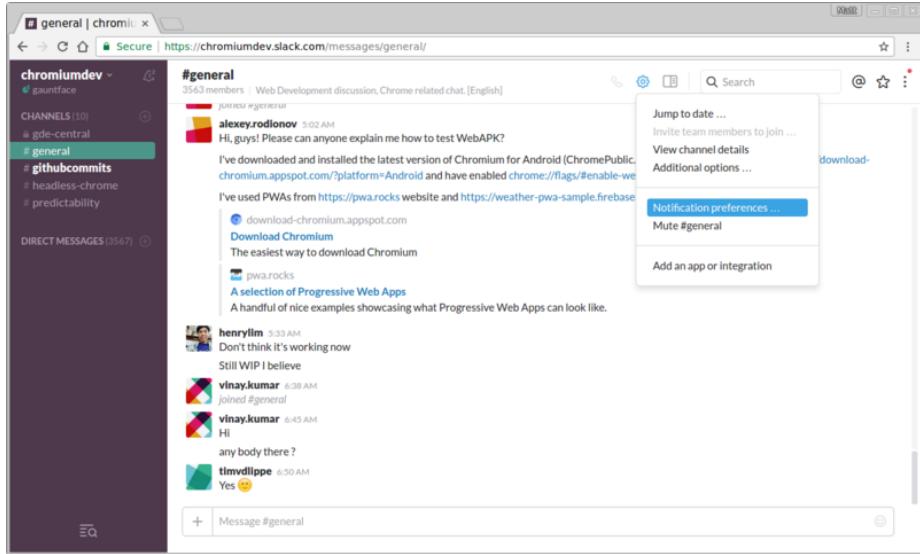


Figure 10: The notification preferences on slack.com easily found under settings drop down.

Passive Approach

One of the easiest ways to offer push to a user is to have a button or toggle switch that enables / disables push messages in a location on the page that is consistent throughout a site.

This doesn't drive users to enable push notifications, but consistency and allowing users to opt in and out easily without constant nudging gives users a way to engage with a site / service / brand if and when they want to. For sites like blogs that might have some regular viewers as well as high bounce rates, this is a solid option as it targets regular viewers without annoying drive-by visitors.

On my personal site I have a toggle switch for push messaging in the footer.

It's fairly out of the way, but for regular visitors it should get enough attention from readers wanting to get updates. People landing on my site to get some

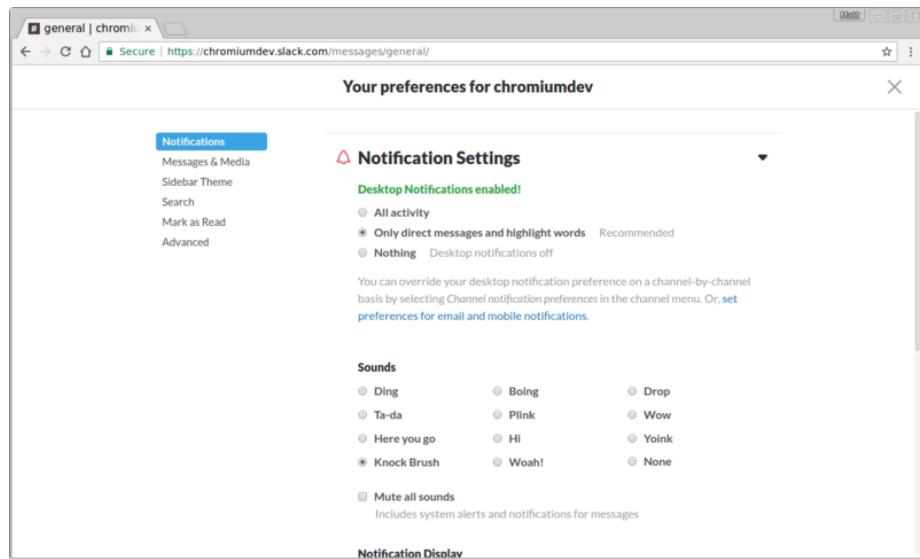


Figure 11: The settings panel for notifications on slack.com.

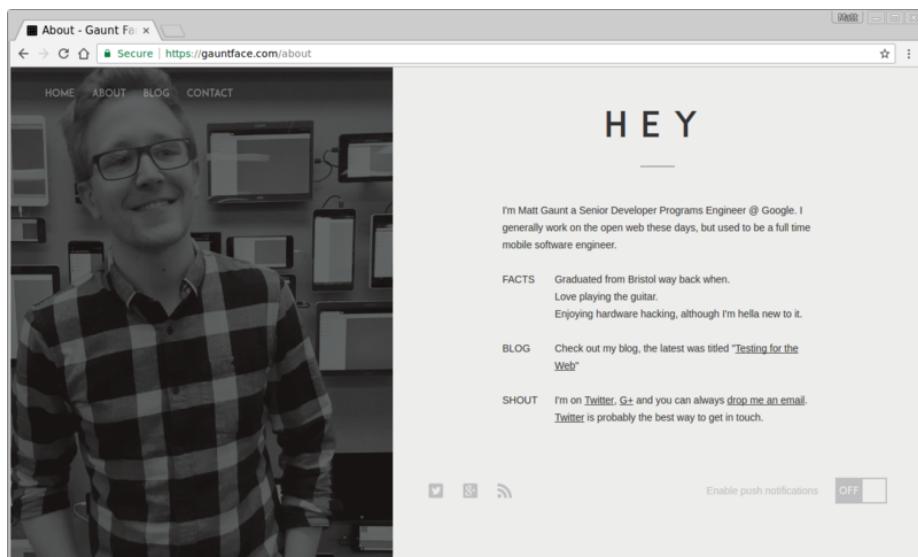


Figure 12: Example of Gauntface.com push notification toggle in footer.

information and leave are completely unaffected, I doubt they even notice the the toggle switch.

If you grant permission the state of the toggle switch changes and remains in the same location through the pages.

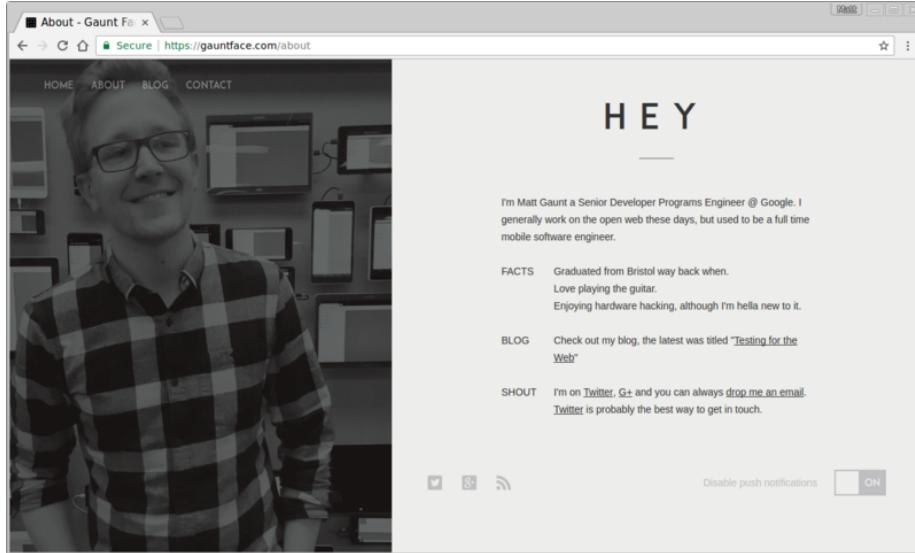


Figure 13: Example of Gauntface.com with notifications enabled.

The Bad UX

Those are some of the common practices I've noticed on the web. Sadly there is one very common bad practice.

The worst thing you can do is instantly show the permission dialog to a user as soon as they land on your site.

They have zero context on why they are being asked for a permission, they may not even know what your website is for, what it does or what it offers. Blocking permissions at this point out of frustration is not uncommon, this pop-up is getting in the way of what they are trying to do.

Remember, if the user *blocks* the permission request, your web app can't ask for permission again. To get permission after being blocked the user has to change the permission in the browsers UI and it is not easy, obvious or fun for the user.

No matter what, don't ask for permission as soon as the user opens your site, consider some other UI or approach that has an incentive for the user to grant permission.

Offer a Way Out

Aside from considering the UX to subscribe a user to push, **please** consider how a user should unsubscribe / opt out of push messaging.

The number of sites that ask for permission as soon as the page load and then offer no UI for disabling push notifications is astounding.

[Vice News](#) is an example of this practice. (p.s. sorry Vice for using you as an example, you were first site I recalled doing this, although I believe it's fixed now.)

When you land on Vice News you'd get the permission prompt. This isn't the end of the world, but it does offend the senses.

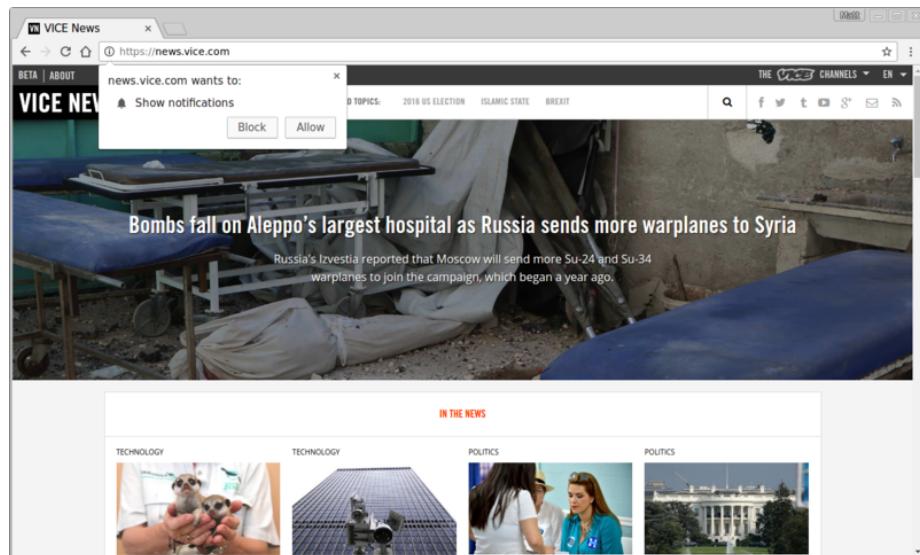


Figure 14: Vice news instantly asks for permission.

If you allow notifications, where would you go to disable them? The websites UI doesn't change at all.

This UX pushes the responsibility of notification management onto the browser, which frankly is awful in Chrome.

As a result, sites are getting users signed up, then forcing them to trial and error their way through the browser UX to disable notifications.

If you're curious what the Browser UX is for disabling push, the desktop has two options. You can visit the web page and click the padlock in the URL bar to configure permissions.

If the web page is closed, users can click the cog on a notification, which takes the user to this page in the settings of Chrome.

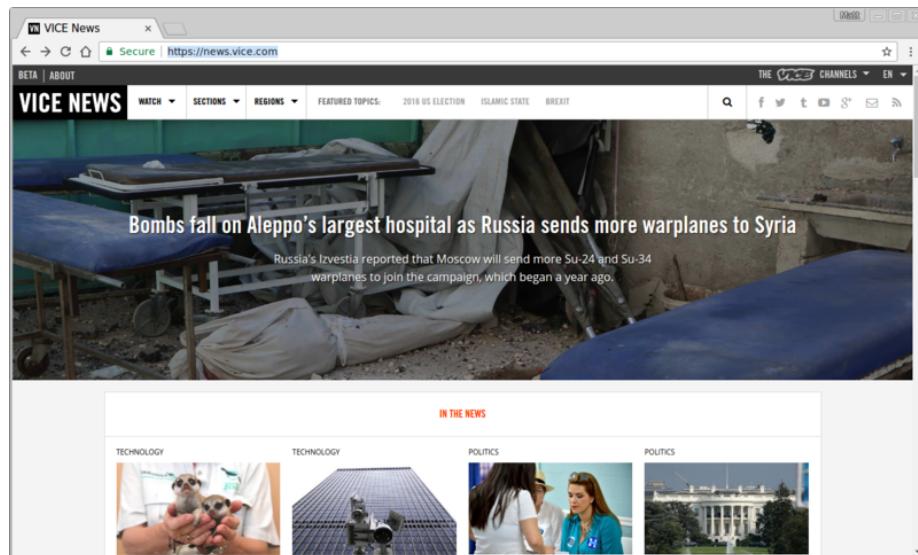


Figure 15: Vice news after granting permission.

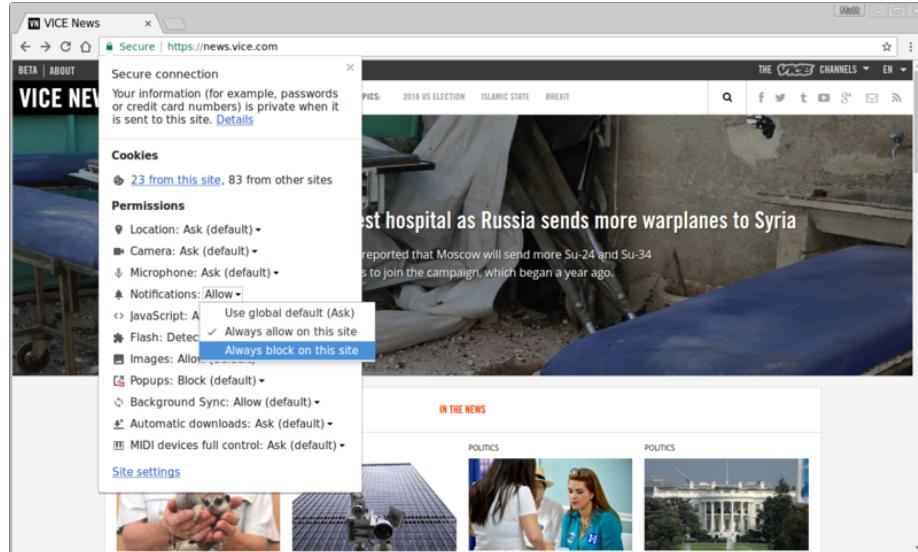


Figure 16: Chrome Notification Permissions from URL Bar.

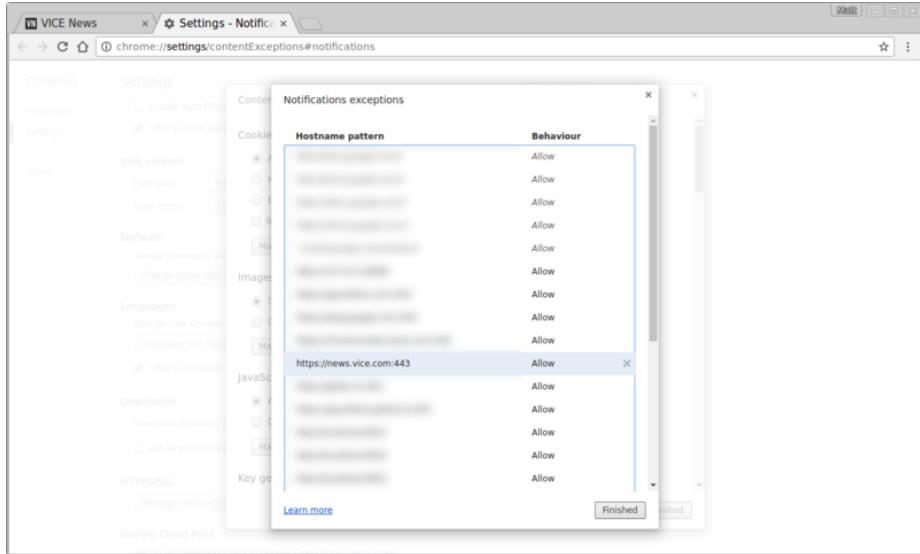


Figure 17: Chrome Notification Permissions from URL Bar.

Neither of these options are particularly pleasant for the user.

Your site should explain to your users how they can disable push. If you don't, users are likely to take the nuclear option and block permission permanently.

Sending Messages with Web Push Libraries

One of the pain points when working with web push is that triggering a push message is extremely "fiddly". To trigger a push message an application needs to make a POST request to a push service following the [web push protocol](#). To use push across all browsers you need to use [VAPID](#) (a.k.a application server keys) which basically requires setting a header with a value proving your application can message a user. To send data with a push message, the data needs to be [encrypted](#) and specific headers added so the browser can decrypt the message correctly.

The main issue with triggering push is that if you hit a problem, it's difficult to diagnose the issue. This is improving with time and wider browser support but it's far from easy. For this reason, I strongly recommend using a library to handle the encryption, formatting and triggering of your push message.

If you really want to learn about what the libraries do and look at each closer, we'll cover it in the next section. For now, we are going to look at how to manage subscriptions and use an existing web-push library to make the push requests.

In this section we'll be using the [web-push for node library](#). Other languages will have differences, but they won't be too dissimilar. We are looking at Node since it's JavaScript and should be the most accessible for readers.

Remember: If you want a library for a different language, checkout the [web-push-libs org on Github](#).

We'll go through the following steps:

1. Look at an example of how to send a subscription to our backend and how that can be saved.
2. What it looks like to retrieve the saved subscriptions and trigger a push message.

Saving Subscriptions

Saving and querying `PushSubscriptions` from a database will vary depending on your server side language and database choice but it might be useful to see an example of how it could be done.

In the demo web page the `PushSubscription` is sent to our backend by making a simple POST request:

```
function sendSubscriptionToBackEnd(subscription) {
  return fetch('/api/save-subscription/', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(subscription)
  })
  .then(function(response) {
    if (!response.ok) {
      throw new Error('Bad status code from server.');
    }

    return response.json();
  })
  .then(function(responseData) {
    if (!(responseData.data && responseData.data.success)) {
      throw new Error('Bad response from server.');
    }
  });
}
```

The `Express` server in our demo has a matching request listener for `/api/save-subscription/` endpoint:

```
app.post('/api/save-subscription/', function (req, res) {
```

In this route we validate the subscription, just to make sure the request is ok and not full of garbage:

```
const isValidSaveRequest = (req, res) => {
  // Check the request body has at least an endpoint.
  if (!req.body || !req.body.endpoint) {
    // Not a valid subscription.
    res.status(400);
    res.setHeader('Content-Type', 'application/json');
    res.send(JSON.stringify({
      error: {
        id: 'no-endpoint',
        message: 'Subscription must have an endpoint.'
      }
    }));
    return false;
  }
  return true;
};
```

In this route we only check for an endpoint. If you **require** payload support, make sure you check for the auth and p256dh keys as well.

If the subscription is valid, we need to save it and return an appropriate JSON response:

```
return saveSubscriptionToDatabase(req.body)
.then(function(subscriptionId) {
  res.setHeader('Content-Type', 'application/json');
  res.send(JSON.stringify({ data: { success: true } }));
})
.catch(function(err) {
  res.status(500);
  res.setHeader('Content-Type', 'application/json');
  res.send(JSON.stringify({
    error: {
      id: 'unable-to-save-subscription',
      message: 'The subscription was received but we were unable to save it to our database'
    }
  }));
});
```

This demo uses `nedb` to store the subscriptions, it's a simple file based database, but you could use any database you chose. We are only using this as it requires zero set-up. For production you'd want to use something more reliable (I tend to stick with good old MySQL).

```
function saveSubscriptionToDatabase(subscription) {
  return new Promise(function(resolve, reject) {
    db.insert(subscription, function(err, newDoc) {
      if (err) {
        reject(err);
        return;
      }

      resolve(newDoc._id);
    });
  });
}
```

Sending Push Messages

When it comes to sending a push message we ultimately need some event to trigger the process of sending a message to users. A common approach would be creating an admin page that lets you configure and trigger the push message. But you could create a program to run locally or any other approach that allows accessing the list of `PushSubscriptions` and running the code to trigger the push message.

Our demo has an “admin like” page that lets you trigger a push. Since it’s just a demo it’s a public page.

I’m going to go through each step involved in getting the demo working, these will be baby steps to everyone follow along, including anyone who is new to node.

When we discussed subscribing a user we covered adding an `applicationServerKey` to the `subscribe()` options. It’s on the back end that we’ll need the private key.

Remember you can use the web-push tool to generate application server keys or use web-push-codelab.appspot.com to generate some application server keys. See “[How to Create Application Server Keys](#)” for more details.

In the demo these values are added to our node app like so (boring code I know, but just want you to know there is no magic):

```

const vapidKeys = {
  publicKey: 'BE162iUYgUivxIkv69yViEuiB1a-Ib9-SkvMeAtA3LFgDzkrxZJjSgSnfckjBJuBkr3qBUYIHB
  privateKey: 'UUxI408-FbRouAevSmBQ6o18hgE4nSG3qwvJTfKc-1s'
};

```

Next we need to install the `web-push` module for our node server:

```
npm install web-push --save
```

Then in our node script we require in the `web-push` module like so:

```
const webpush = require('web-push');
```

Now we can start to use the `web-push` module. First we need to tell the `web-push` module about our application server keys (remember they are also known as VAPID keys because that's the name of the spec).

```

const vapidKeys = {
  publicKey: 'BE162iUYgUivxIkv69yViEuiB1a-Ib9-SkvMeAtA3LFgDzkrxZJjSgSnfckjBJuBkr3qBUYIHB
  privateKey: 'UUxI408-FbRouAevSmBQ6o18hgE4nSG3qwvJTfKc-1s'
};

webpush.setVapidDetails(
  'mailto:web-push-book@gautface.com',
  vapidKeys.publicKey,
  vapidKeys.privateKey
);

```

We also include a “mailto:” string as well. This string needs to be either a URL or a mailto email address. This piece of information will actually be sent to web push service as part of the request to trigger a push. The reason this is done is so that if a web push service needs to get in touch, they have some information that will enable them to (i.e. a URL, hopefully with some contact information or an email address).

With this, the `web-push` module is ready to use, the next step is to trigger a push message.

The demo uses the pretend admin panel to trigger push messages.

Clicking the “Trigger Push Message” button will make a POST request to `/api/trigger-push-msg/` which is the signal for our backend to start send push messages, so we create the route in express for this endpoint:

```
app.post('/api/trigger-push-msg/', function (req, res) {
```

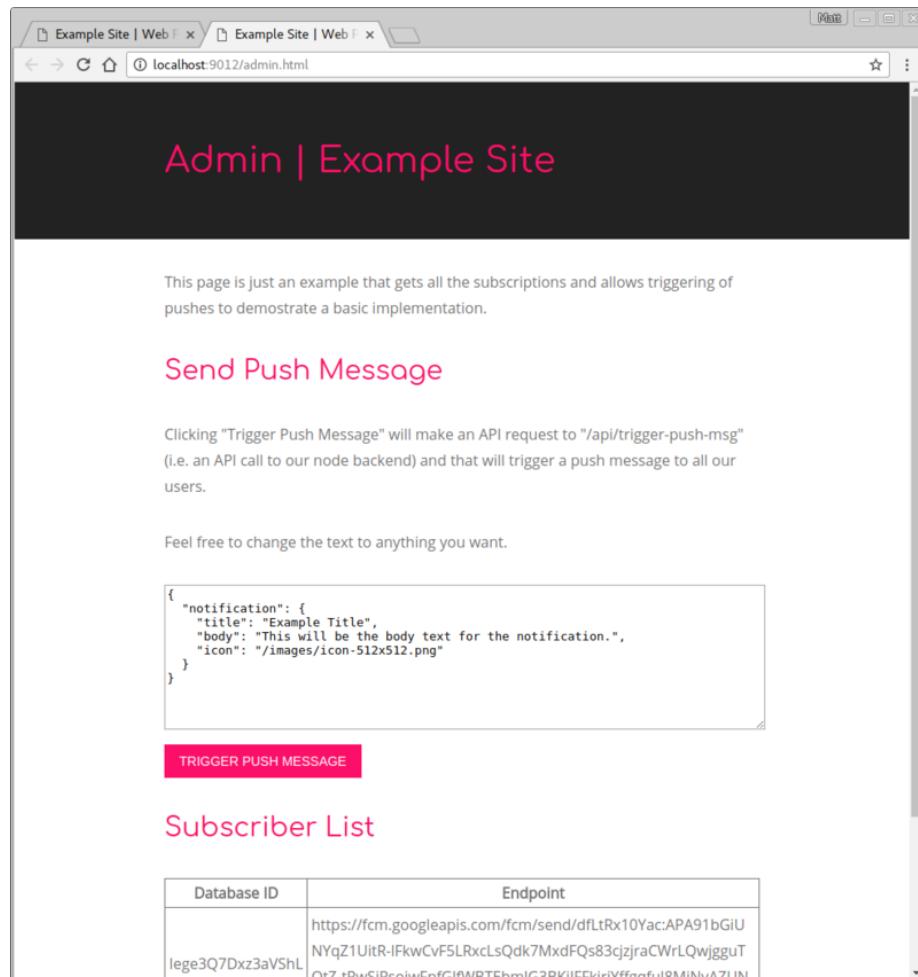


Figure 18: Screenshot of the Admin Page.

When this request is received, we grab the subscriptions from the database and for each one, we trigger a push message.

```
return getSubscriptionsFromDatabase()
.then(function(subscriptions) {
  let promiseChain = Promise.resolve();

  for (let i = 0; i < subscriptions.length; i++) {
    const subscription = subscriptions[i];
    promiseChain = promiseChain.then(() => {
      return triggerPushMsg(subscription, dataToSend);
    });
  }

  return promiseChain;
})
```

The function `triggerPushMsg()` can then use the web-push library to send a message to the provided subscription.

```
const triggerPushMsg = function(subscription, dataToSend) {
  return webpush.sendNotification(subscription, dataToSend)
  .catch((err) => {
    if (err.statusCode === 410) {
      return deleteSubscriptionFromDatabase(subscription._id);
    } else {
      console.log('Subscription is no longer valid: ', err);
    }
  });
};
```

The call to `webpush.sendNotification()` will return a promise. If the message was sent successfully the promise will resolve and there is nothing we need to do. If the promise rejects, we need to handle the error, which may involve removing the subscription from our data (i.e. the subscription is invalid) or perhaps the push service just could not fulfill our request and we need to try again later.

To determine the type of error from a push service it's best to look at the status code. Error messages vary between push services and some are more helpful than others.

In this example it checks for status code '404' and '410', which are the HTTP status codes for 'Not Found' and 'Gone'. If we receive this status code, it means the subscription has expired or is no longer valid, in these scenarios we need to remove the subscriptions from our database.

We'll cover some of the other status codes in the next section when we look at the web push protocol in more detail.

If you hit problems at this stage, it's worth looking at the error logs from Firefox before Chrome. The Mozilla push service has much more helpful error messages compared to Chrome / FCM.

After looping through the subscriptions, we need to return a JSON response.

```
.then(() => {
  res.setHeader('Content-Type', 'application/json');
  res.send(JSON.stringify({ data: { success: true } }));
})
.catch(function(err) {
  res.status(500);
  res.setHeader('Content-Type', 'application/json');
  res.send(JSON.stringify({
    error: {
      id: 'unable-to-send-messages',
      message: `We were unable to send messages to all subscriptions : ` +
        `${err.message}`
    }
  }));
});
```

We've gone over the major implementation steps.

1. Create an API to send subscriptions from our web page to our back-end so it can save them in a database.
2. Created an API to trigger sending push messages (in this case an API called from the pretend admin panel).
3. Looked at an example of how to get all the subscriptions from our backend and send a message to each one with one of the [web-push libraries](#).

Regardless of your backend (Node, PHP, Python,) the steps for implementing push are going to be the same.

Next up, what exactly are these Web Push libraries doing for us?

The Web Push Protocol

We've seen how a library can be used to trigger push messages, but what exactly are these libraries doing?

Well, it's primarily making a network request, making sure it's the right format and sent the right way. The spec that defines this network request is the [Web Push Protocol](#).

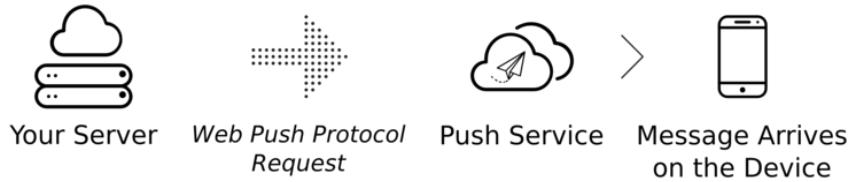


Figure 19: Diagram of sending a push message from your server to a push service.

This section outlines how the server can identify itself with application server keys and how the encrypted payload and associated data is sent.

This isn't a pretty side of web push and I'm no expert at encryption, but let's look through each piece since it's handy to know what these libraries are doing under the hood.

Application Server Keys

When we subscribe a user, we pass in an `applicationServerKey`. This key is passed to the push service and used to check the application that subscribed the user is also the same application that is trigger push messages.

When we trigger a push message, there are a set of headers that we send that allows the push service to authenticate the application (This is defined by the [VAPID spec](#)).

What does all this actually mean and what exactly happens? Well these are the steps taken for application server authentication:

1. The application signs some information with its **private application key**.
2. This signed information is sent to the push service as a header in the POST request.
3. The push service uses the stored public key is received from `subscribe()` to check that the signed information is signed by the private key relating to the public key. *Remember:* The public key is the `applicationServerKey` passed into the `subscribe` call.
4. If the signed information is valid the push service sends the push message to the user.

An example of this flow of information is below (Not the legend in the bottom left to indicate public and private keys):

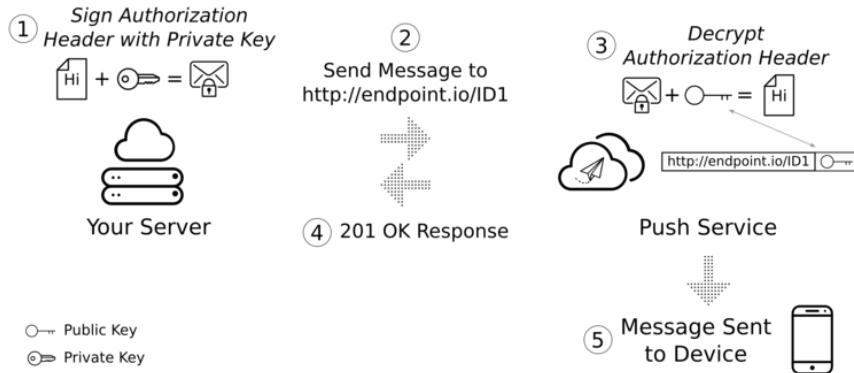


Figure 20: Illustration of how the private application server key is used when sending a message.

The “signed information” added to a header in the request is a JSON Web Token.

JSON Web Token

A [JSON web token](#) (or JWT for short) is a way of sending a message to a third party such that if the third party has an applications public key, they can validate the signature of the JWT using the public key and check that it's from the expected application because it has to have been signed by the matching private key.

There are a host of libraries on <https://jwt.io/> that can do the signing for you and I'd recommend you do that where you can, but let's look at how we create a signed JWT.

Web Push and Signed JWT

A signed JWT is just a string, but it can be thought of as three strings joined by dots.

The first and second strings (The JWT info and JWT data) are pieces of JSON that have been base 64 encoded, meaning it's publicly readable.

The first string is information about the JWT itself, indicating what algorithm was used to create the signature.

The JWT info for web push must be be JSON containing the following information, encoded as a URL safe base64 string.

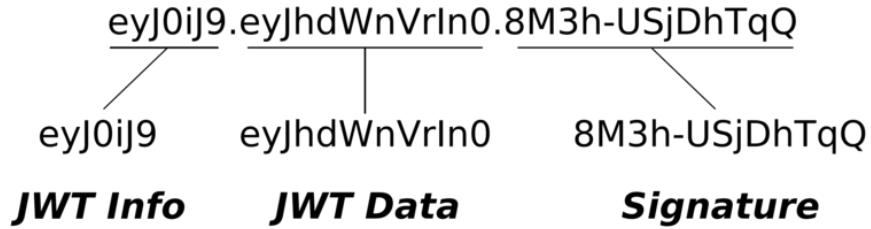


Figure 21: A illustration of the strings in a JSON Web Token

```
{
  "typ": "JWT",
  "alg": "ES256"
}
```

The second string is the JWT Data. This provides information about the sender of the JWT, who it's intended for and how long it's valid.

For web push, the data would look something like this:

```
{
  "aud": "https://some-push-service.org",
  "exp": "1469618703",
  "sub": "mailto:example@web-push-book.org"
}
```

The `aud` value is the “audience”, i.e. who the JWT is for. For web push the audience is the push service, so we set it to the **origin of the push service**.

The `exp` value is the expiration of the JWT, this prevent snoopers from being able to re-use a JWT if they intercept it. The expiration is a timestamp in seconds and must be no longer 24 hours.

In the Node.js library the expiration is set to `Math.floor(Date.now() / 1000) + (12 * 60 * 60)`. It's 12 hours rather than 24 hours to avoid any issues with time differences between the sending application and the push service.

Finally, the `sub` value needs to be either a URL or a `mailto` email address. This is so that if a push service needed to reach out to sender, it can find contact info from the JWT. (This is why the web-push library needed an email address).

Just like the JWT Info, the JWT Data is encoded as a URL safe base64 string.

The third string, the signature, is the result of taking the first two strings (the JWT Info and JWT Data), joining them with a dot character, which we'll call the “unsigned token”, and signing it.

The signing process requires encrypting the “unsigned token” using ES256. According to the [JWT spec](#), ES256 is short for “ECDSA using the P-256 curve and the SHA-256 hash algorithm”. Using web crypto you can create the signature like so:

```
// Utility function for UTF-8 encoding a string to an ArrayBuffer.
const utf8Encoder = new TextEncoder('utf-8');

// The unsigned token is the concatenation of the URL-safe base64 encoded
// header and body.
const unsignedToken = .....;

// Sign the |unsignedToken| using ES256 (SHA-256 over ECDSA).
const key = {
  kty: 'EC',
  crv: 'P-256',
  x: window.uint8ArrayToBase64Url(
    applicationServerKeys.publicKey.subarray(1, 33)),
  y: window.uint8ArrayToBase64Url(
    applicationServerKeys.publicKey.subarray(33, 65)),
  d: window.uint8ArrayToBase64Url(applicationServerKeys.privateKey),
};

// Sign the |unsignedToken| with the server's private key to generate
// the signature.
return crypto.subtle.importKey('jwk', key, {
  name: 'ECDSA', namedCurve: 'P-256',
}, true, ['sign'])
.then((key) => {
  return crypto.subtle.sign({
    name: 'ECDSA',
    hash: {
      name: 'SHA-256',
    },
  }, key, utf8Encoder.encode(unsignedToken));
})
.then((signature) => {
  console.log('Signature: ', signature);
});
```

For the push service receiving our JWT, they can check the application sending the JWT by using the public key to decrypt the signature and make sure the decrypted string is the same as the “unsigned token” (i.e. the first two strings in the JWT).

The signed JWT (i.e. all three strings joined by dots), is sent to the web push

service as the `Authorization` header with `WebPush` prepended, like so:

```
Authorization: 'WebPush <JWT Info>.<JWT Payload>.<Signature>'
```

The Web Push Protocol also states the public application server key must be sent in the `Crypto-Key` header as a URL safe base64 encoded string with `p256ecdsa=` prepended to it.

```
Crypto-Key: p256ecdsa=<URL Safe Base64 Public Application Server Key>
```

The Payload Encryption

Next let's look at how we can send a payload with a push message so that our when our web app receives a push message, it can access the data it receives.

A common question that arises from any who's used other push services is why does the web push payload need to be encrypted? With native apps, push messages can send data as plain text.

Part of the beauty of web push is that because all push services use the same API (the web push protocol), developers don't have to care who the push service is. We can make a request in the right format and expect a push message to be sent. The downside of this is that developers could conceivably send messages to a push service that isn't trustworthy. By encrypting the payload, push service can't read the data that's sent, only the browser can decrypt the information. This protects the user's data.

The encryption of the payload is defined in the [Message Encryption spec](#).

Before we look at the specific steps to encrypt our payload for a `PushSubscription`, we should cover some techniques that'll be used during the encryption process (Massive H/T to Matt Scales for his excellent article on push encryption).

ECDH and HKDF

Both ECDH and HKDF are used throughout the encryption process and offer benefits for the purpose of encrypting information.

ECDH: Elliptic Curve Diffie-Hellman Key Exchange Imagine you have two people who want to share information, Alice and Bob. Both Alice and Bob have their own public and private keys. With ECDH, Alice and Bob share their public keys with each other. The useful property of ECDH is that Alice can use her private key and Bob's public key to create secret value 'X'. Bob can then

do the same, taking his private key and Alice's public key to create the same secret value 'X'.

Now Bob and Alice can use 'X' to encrypt and decrypt messages between them. ECDH, to the best of my knowledge, defines the properties of curves which allow this "feature" of making a shared secret 'X'.

This is a high level explanation of ECDH, if you want to learn more [I recommend checking out this video](#).

In terms of code; most languages / platforms come with libraries to make it easy to generate these keys.

In node we'd do the following:

```
const keyCurve = crypto.createECDH('prime256v1');
keyCurve.generateKeys();

const publicKey = keyCurve.getPublicKey();
const privateKey = keyCurve.getPrivateKey();
```

HKDF: HMAC Based Key Derivation Function Wikipedia has a succinct description of [HKDF](#):

HKDF is an HMAC based key derivation function that transforms any weak key material into cryptographically strong key material. It can be used, for example, to convert Diffie Hellman exchanged shared secrets into key material suitable for use in encryption, integrity checking or authentication.

– [Wikipedia](#)

Essentially, HKDF will take input that is not particular secure and make it more secure.

The spec defining this encryption requires use of SHA-256 as our hash algorithm and the resulting keys for HKDF in web push should be no longer than 256 bits (32 bytes).

In node this could be implemented like so:

```
// Simplified HKDF, returning keys up to 32 bytes long
function hkdf(salt, ikm, info, length) {
    // Extract
    const keyHmac = crypto.createHmac('sha256', salt);
    keyHmac.update(ikm);
    const key = keyHmac.digest();
```

```

// Expand
const infoHmac = crypto.createHmac('sha256', key);
infoHmac.update(info);

// A one byte long buffer containing only 0x01
const ONE_BUFFER = new Buffer(1).fill(1);
infoHmac.update(ONE_BUFFER);

return infoHmac.digest().slice(0, length);
}

```

H/T to [Mat Scale's article for this example code](#).

This loosely covers [ECDH](#) and [HKDF](#).

ECDH a secure way to share public keys and generate a shared secret. HKDF is a way to take insecure material and make it secure.

This will be used during the encryption of our payload, so next let's look at what we take as input and how that's encrypted.

Inputs

When we want to send a push message to a user with a payload, there are three inputs we need:

1. The payload itself.
2. The `auth` secret from the PushSubscription.
3. The `p256dh` key from the PushSubscription.

We've seen the `auth` and `p256dh` values being retrieved from a `PushSubscription` but for a quick reminder, given a subscription we'd need these values:

```

subscription.toJSON().keys.auth
subscription.toJSON().keys.p256dh

subscription.getKey('auth')
subscription.getKey('p256dh')

```

The `auth` value should be treated as a secret and not shared outside of your application.

The `p256dh` key is a public key, this is sometimes referred to as the client public key. Here we'll refer to `p256dh` as the subscription public key. The subscription

public key is generated by the browser. The browser will keep the private key secret and use it for decrypting the payload.

These three values, `auth`, `p256dh` and `payload` are needed as inputs and the result of the encryption process will be the encrypted payload, a salt value and a public key used just for encrypting the data.

Salt

The salt needs to be 16 bytes of random data. In NodeJS, we'd do the following to create a salt:

```
const salt = crypto.randomBytes(16);
```

Public / Private Keys

The public and private keys should be generated using a P-256 elliptic curve, which we'd do in Node like so:

```
const localKeysCurve = crypto.createECDH('prime256v1');
localKeysCurve.generateKeys();

const localPublicKey = localKeysCurve.getPublicKey();
const localPrivateKey = localKeysCurve.getPrivateKey();
```

We'll refer these keys as "local keys". They are used *just* for encryption and have **nothing** to do with application server keys.

With the payload, auth secret and subscription public key as inputs and with a newly generated salt and set of local keys, we are ready to actually do some encryption.

Shared Secret

The first step is to create a shared secret using the subscription public key and our new private key (remember the ECDH explanation with Alice and Bob? Just like that).

```
const sharedSecret = localKeysCurve.computeSecret(
  subscription.keys.p256dh, 'base64');
```

This is used in the next step to calculate the Pseudo Random Key (PRK).

Pseudo Random Key

The Pseudo Random Key (PRK) is the combination of the push subscription's auth secret, the shared secret we just created.

```
const authEncBuff = new Buffer('Content-Encoding: auth\0', 'utf8');
const prk = hkdf(subscription.keys.auth, sharedSecret, authEncBuff, 32);
```

What is up with the create this Content-Encoding: auth buffer?

This piece of information is included in the final output of the HKDF and I imagine is largely there to help browsers identify that the decryption was successful / what the decrypted information is. *Note* the \0 is just to add a byte with value of 0 to end of the Buffer.

So our Pseudo Random Key is simply running the auth, shared secret and a piece of encoding info through HKDF (i.e. making it cryptographically stronger).

Context

The “context” is a set of bytes that is used to calculate two values later on in the encryption browser. It’s essentially an array of bytes containing the subscription public key and the local public key.

```
const keyLabel = new Buffer('P-256\0', 'utf8');

// Convert subscription public key into a buffer.
const subscriptionPubKey = new Buffer(subscription.keys.p256dh, 'base64');

const subscriptionPubKeyLength = new Uint8Array(2);
subscriptionPubKeyLength[0] = 0;
subscriptionPubKeyLength[1] = subscriptionPubKey.length;

const localPublicKeyLength = new Uint8Array(2);
subscriptionPubKeyLength[0] = 0;
subscriptionPubKeyLength[1] = localPublicKey.length;

const contextBuffer = Buffer.concat([
  keyLabel,
  subscriptionPubKeyLength.buffer,
  subscriptionPubKey,
  localPublicKeyLength.buffer,
  localPublicKey,
]);
```

The final context buffer is a label, the number of bytes in the subscription public key, followed by the key itself, then the number of bytes local public key, followed by the key itself.

With this context value we can use it in the creation of a nonce and a content encryption key (CEK).

Content Encryption Key and Nonce

A `nonce` is a value that prevents replay attacks as it should only be used once.

The content encryption key (CEK) is the key that will ultimately be used to encrypt our payload.

First we need to create the bytes of data for the nonce and CEK, which is simply a content encoding string followed by the context buffer we just calculated:

```
const nonceEncBuffer = new Buffer('Content-Encoding: nonce\0', 'utf8');
const nonceInfo = Buffer.concat([nonceEncBuffer, contextBuffer]);

const cekEncBuffer = new Buffer('Content-Encoding: aesgcm\0');
const cekInfo = Buffer.concat([cekEncBuffer, contextBuffer]);
```

This information is run through HKDF combining the salt and PRK with the `nonceInfo` and `cekInfo`:

```
// The nonce should be 12 bytes long
const nonce = hkdf(salt, prk, nonceInfo, 12);

// The CEK should be 16 bytes long
const contentEncryptionKey = hkdf(salt, prk, cekInfo, 16);
```

This gives us our nonce and content encryption key.

Perform the Encryption

Now that we have our content encryption key, we can encrypt the payload.

We create an AES128 Cipher using the content encryption key as the key and the nonce is as an initialization vector.

In Node this is done like so:

```
const cipher = crypto.createCipheriv(
  'id-aes128-GCM', contentEncryptionKey, nonce);
```

Before we encrypt our payload, we need to define how much padding we wish to add to the front of the payload. The reason you'd want to add padding is that it further hides what the content of the payload *could* be.

The padding needs to be at least 2 bytes and these first two bytes should indicate the length of the padding. If you do add padding, create enough bytes of data to add the two bytes used to include the padding length.

```
const padding = new Buffer(2 + paddingLength);
// The buffer must be only zeros, except the length
padding.fill(0);
padding.writeUInt16BE(paddingLength, 0);
```

We then run our padding and payload through this cipher.

```
const result = cipher.update(Buffer.concat(padding, payload));
cipher.final();

// Append the auth tag to the result -
// https://nodejs.org/api/crypto.html#crypto_cipher_getauthtag
const encryptedPayload = Buffer.concat([result, cipher.getAuthTag()]);
```

We now have our encrypted payload - Yay.

The remaining piece of work is to determine how this payload is sent to the push service.

Encrypted Payload Headers & Body

To send this encrypted payload to the Push Service we need to define a few different headers in our POST request.

Encryption Header The ‘Encryption’ header must contain the *salt* used for encrypting the payload.

The 16 byte salt should be base64 URL safe encoded and added to the Encryption header, like so:

`Encryption: salt=<URL Safe Base64 Encoded Salt>`

Crypto-Key Header We saw that the ‘Crypto-Key’ header is used under the ‘Application Server Keys’ section to contain the public application server key.

This header is also used to share the local public key used to encrypt the payload.

The resulting header looks like this:

Crypto-Key: dh=<URL Safe Base64 Encoded Local Public Key String>; p256ecdsa=<URL Safe Base64 Encoded Local Public Key String>

Content Type, Length & Encoding Headers The ‘Content-Length’ header is the number of bytes in the encrypted payload and ‘Content-Type’ and ‘Content-Encoding’ headers are fixed values, as shown below:

```
Content-Length: <Number of Bytes in Encrypted Payload>
Content-Type: 'application/octet-stream'
Content-Encoding: 'aesgcm'
```

With these headers set, we need to send the encrypted payload as the body of our request. Notice that the Content-Type is set to ‘application/octet-stream’. This is because the encrypted payload must be sent as a stream of bytes.

In NodeJS we would do this like so:

```
const pushRequest = https.request(httpsOptions, function(pushResponse) {
pushRequest.write(encryptedPayload);
pushRequest.end();
```

More Headers?

We’ve covered the headers used for JWT / Application Server Keys (i.e. how to identify the application with the push service) and we’ve covered the headers used to send an encrypted payload.

There are some additional headers that can be used by Push Services to alter the behavior of the send messages. Some of these headers are required, while others are optional.

TTL Header

This is a **required header**.

TTL (or time to live) should be an integer for the number of seconds you want your push message to live on the push service before it’s delivered. If the TTL is expired, the message will be removed from the Push Service queue and it won’t be delivered.

TTL: <Time to live in seconds>

If you set a TTL of zero, the push service will attempt to deliver the message immediately **but** if the device can’t be reached, your message will be immediately dropped from the push service queue.

Technically a push service can reduce the TTL of a push message if it wants. You can tell if this has happened by examining the TTL header in the response from a push service.

Topic This is an **optional header**.

Topics are strings that can be used to replace any pending notifications with new notifications if they have matching topic names.

This is useful in scenarios where the user's device might have been offline during which you might have sent three messages and you may want the user to receive only the latest message when their device is eventually turned back on.

Urgency This is an **optional header**.

This can be used to indicate to the push service how important a message is to the user. This could be used by the push service to help conserve the battery life of a users device by only waking up for important messages when battery is low.

The header value should a string value of “very-low” | “low” | “normal” | “high”. The default value is “normal”.

Urgency: <very-low | low | normal | high>

Everything Together

If you have further questions about how this all works you can always see how libraries trigger push messages on [the web-push-libs org](#).

Once you have an encrypted payload, and the headers above, you just need to make a POST request to the `endpoint` in a `PushSubscription`.

So what do we do with the response to this POST request?

Response from Push Service

Once you've made a request to a Push Service, you need to check the status code of the response as that'll inform you as to whether the request was successful or not.

Status Code

Description

201

Created. The request to send a push message was received and accepted.

429

Too many requests. Meaning your application server has reached a rate limit with a push service. The push service should include a ‘Retry-After’ header to indicate how long before another request can be made.

400

Invalid request. This generally means one of your headers is invalid or poorly formatted.

404

Not Found. This is an indication that the subscription is expired and can’t be used. In this case you should delete the PushSubscription and wait for the client to resubscribe the user.

410

Gone. The subscription is no longer valid and should be removed from back end. This can be reproduced by calling `unsubscribe()` on a PushSubscription.

413

Payload size too large. The minimum size payload a push service must support is 4096 bytes (or 4kb).

Push Events

By this point covered subscribing a user for push sending them a message. The next step is to receive this push message on the users device and display a notification (as well as any other work we might want to do).

The Push Event

When a message is received, it’ll result in a push event being dispatched in your service worker.

The code for setting up a push event listener should be pretty similar to any other event listener you’d write in JavaScript:

```
self.addEventListener('push', function(event) {
  if (event.data) {
    console.log('This push event has data: ', event.data.text());
  } else {
    console.log('This push event has no data.');
  }
});
```

The weirdest bit of this code to most developers who are new to service workers is the `self` variable. `self` is commonly used in Web Workers, which a service worker is. `self` refers to the global scope, kind of like `window` in a web page, but for a Web Worker / Service Worker, `self` refers to the the worker itself.

In the example above `self.addEventListener()` can be thought of as adding an event listener to the service worker itself.

Inside the push event example we check if there is any data and print something to the terminal.

There are other ways you can parse data from a push event:

```
// Returns string
event.data.text()

// Parses data as JSON string and returns an Object
event.data.json()

// Returns blob of data
event.data.blob()

// Returns an arrayBuffer
event.data.arrayBuffer()
```

Most people use `json()` or `text()` depending on what they are expecting from their application.

This example demonstrates how to add a push event listener and how to access data, but it's missing two very important pieces of functionality. It's not showing a notification and it's not making use of `event.waitUntil()`.

Wait Until

One of the things to understand about service workers is that you have little control over when the service worker code is going to run. The browser decides when to wake it up and when to terminate it. The only way you can tell the browser, "Hey I'm super busy doing important stuff", is to pass a promise into the `event.waitUntil()` method. With this, the browser will keep the service worker running until the promise you passed in has settled.

With push events there is an additional requirement that you must display a notification before the promise you passed in has settled.

Here's a basic example of showing a notification:

```
self.addEventListener('push', function(event) {
  const promiseChain = self.registration.showNotification('Hello, World.');
```

```
    event.waitUntil(promiseChain);
});
```

Calling `self.registration.showNotification()` is the method that displays a notification to the user and it returns a promise that will resolve once the notification has been displayed.

For the sake of keeping this example as clear as possible I've assigned this promise to a variables called `promiseChain`. This is then passed into `event.waitUntil()`. I know this is very verbose, but I've seen a number of issues that have culminated as a result of misunderstanding what should be passed into `waitUntil()` or is the result of a broken promise chains.

A more complicated example with a network request for data and tracking the push event with analytics could look like this:

```
self.addEventListener('push', function(event) {
  const analyticsPromise = pushReceivedTracking();
  const pushInfoPromise = fetch('/api/get-more-data')
    .then(function(response) {
      return response.json();
    })
    .then(function(response) {
      const title = response.data.userName + ' says...';
      const message = response.data.message;

      return self.registration.showNotification(title, {
        body: message
      });
    });

  const promiseChain = Promise.all([
    analyticsPromise,
    pushInfoPromise
  ]);

  event.waitUntil(promiseChain);
});
```

Here we are calling a function that returns a promise `pushReceivedTracking()`, which we can pretend will make a network request to our analytics provider. We are also making a network request, getting the response and showing a notification using the responses data for the title and message of the notification.

We can ensure the service worker is kept alive while both of these tasks are done by combining these promises with `Promise.all()`. The resulting promise

is passed into `event.waitUntil()` meaning the browser will wait until both promises have finished before checking a notification has been displayed and terminating the service worker.

Tip: If you ever find your promise chains confusing or a little messy I find that breaking things into functions help to reduce complication. I'd also recommend [this blog post by Philip Walton on untangling promise chains](#). The main point to take away is that you should experiment with how promises can be written and chained to find a style that works for you.

The reason we should be concerned about `waitUntil()` and how to use it is that one of the most common issues developers face is that when the promise chain is incorrect / broken, Chrome will show this “default” notification:

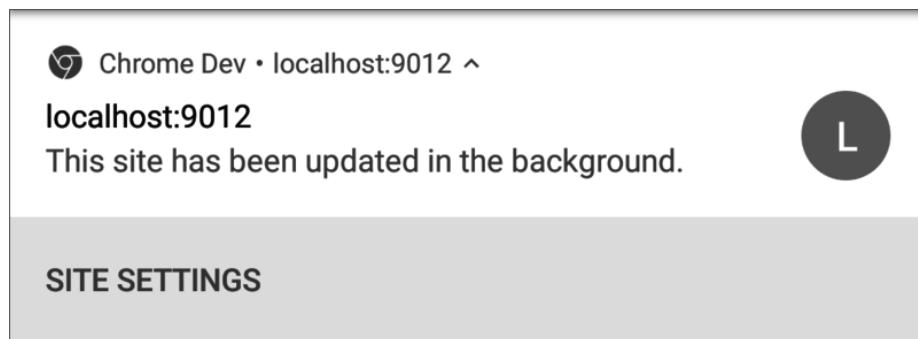


Figure 22: An Image of the default notification in Chrome

Chrome will only show the “This site has been updated in the background.” notification when a push message is received and the push event in the service worker **does not** show a notification after the promise passed to `event.waitUntil()` has finished.

The main reason developers get caught out by this is that their code will often call `self.registration.showNotification()` but they **aren't** doing anything with the promise it returns. This intermittently results in the default notification being displayed. For example, we could remove the return for `self.registration.showNotification()` in the example above and we run the risk of seeing this notification.

```
self.addEventListener('push', function(event) {  
  const analyticsPromise = pushReceivedTracking();  
  const pushInfoPromise = fetch('/api/get-more-data')  
    .then(function(response) {  
      return response.json();  
    })  
  event.waitUntil(Promise.all([analyticsPromise, pushInfoPromise]));  
});
```

```

})
.then(function(response) {
  const title = response.data.userName + ' says...';
  const message = response.data.message;

  self.registration.showNotification(title, {
    body: message
  });
});

const promiseChain = Promise.all([
  analyticsPromise,
  pushInfoPromise
]);

event.waitUntil(promiseChain);
});

```

You can see how it's an easy thing to miss.

Just remember - if you see that notification, check your promise chains and `event.waitUntil()`.

In the next section we're going to look at what we can do in terms of styling notifications / what content we can display.

Displaying a Notification

I've split up notification options into two sections, one that deals with the visual aspects (this section) and one section that explains the behavioural aspects of notifications.

The reason for this is that every developer will need to be worried about the visual aspects but the behavioural aspects will depend on your specific to your use case.

All of the source code for these demo's is taken from a demo page I put together. If you want to test them out for yourself then click the button below.

[Notification Demos](#)

Visual Options

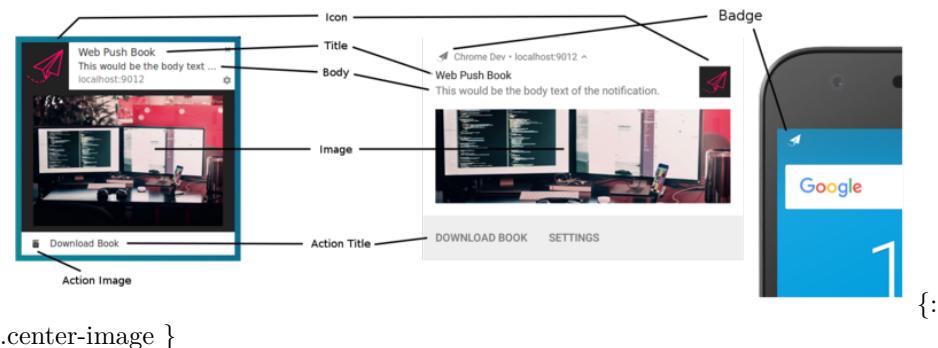
The API for showing a notification is simply:

```
<ServiceWorkerRegistration>.showNotification(<title>, <options>);
```

Where the title is a string and options can be any of the following:

```
{  
  "://" : "Visual Options",  
  "body": "<String>",  
  "icon": "<URL String>",  
  "image": "<URL String>",  
  "badge": "<URL String>",  
  "vibrate": "<Array of Integers>",  
  "sound": "<URL String>",  
  "dir": "<String of 'auto' | 'ltr' | 'rtl'>",  
  
  "://" : "Behavioural Options",  
  "tag": "<String>",  
  "data": "<Anything>",  
  "requireInteraction": "<boolean>",  
  "renotify": "<Boolean>",  
  "silent": "<Boolean>",  
  
  "://" : "Both Visual & Behavioural Options",  
  "actions": "<Array of Strings>",  
  
  "://" : "Doesn't seem to affect any visual aspect of the notification.",  
  "timestamp": "<Long>"  
}
```

First let's look at the visual options. For desktop and mobile, here are how the different options are used.



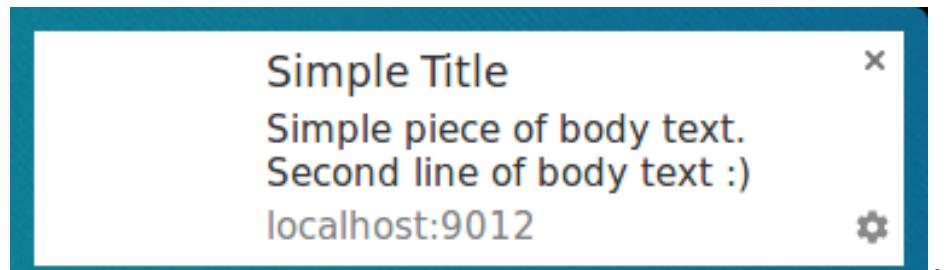
Title and Body Options

The title and body options are exactly as they sound, two different pieces of text to display on the notification.

If we ran the following code:

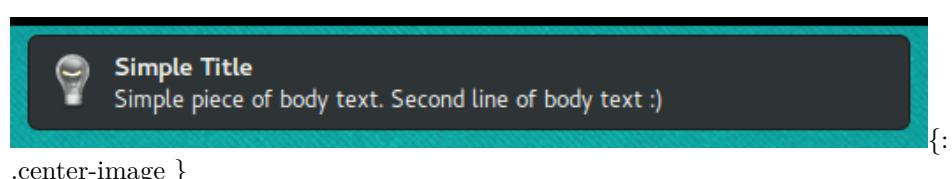
```
const title = 'Simple Title';
const options = {
  body: 'Simple piece of body text.\nSecond line of body text :)'
};
registration.showNotification(title, options);
```

We'd get this notification on Chrome:



.center-image }

On Firefox on Linux it would look like this:



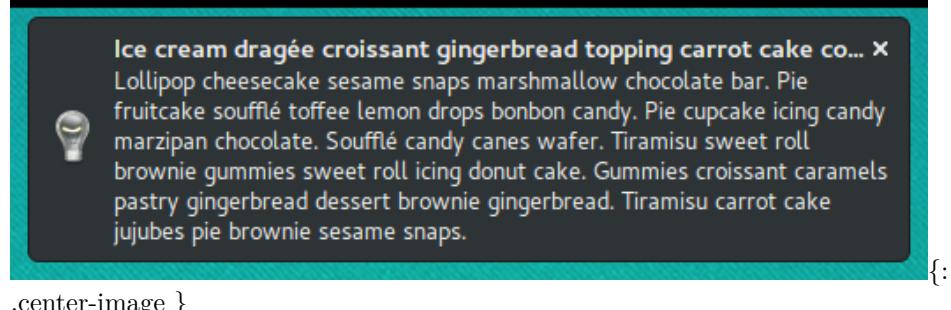
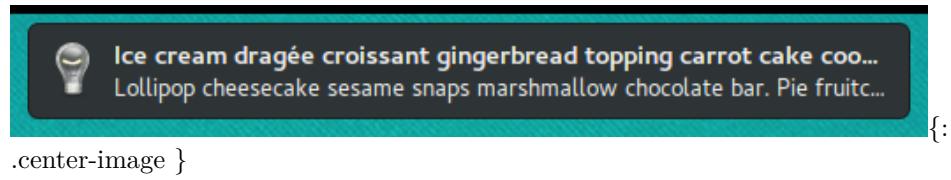
.center-image }

I was curious about what would happen if I added lots of text and this was the result.



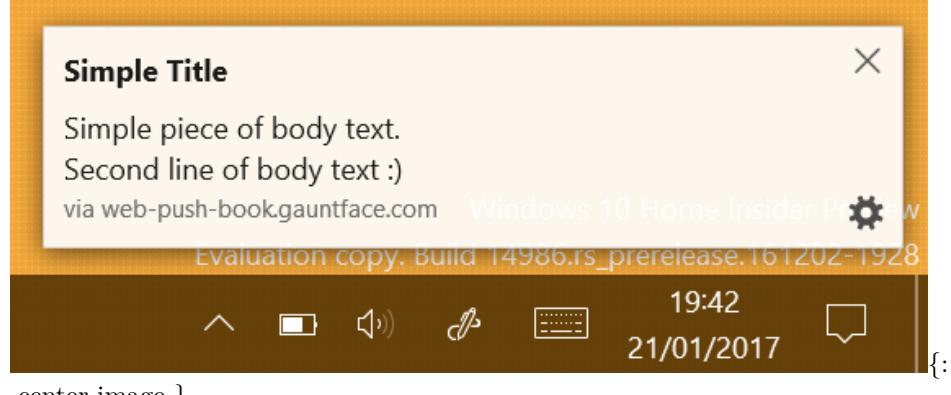
.center-image }

Interestingly, Firefox on Linux collapses the body text until you hover the notification, causing the notification to expand.

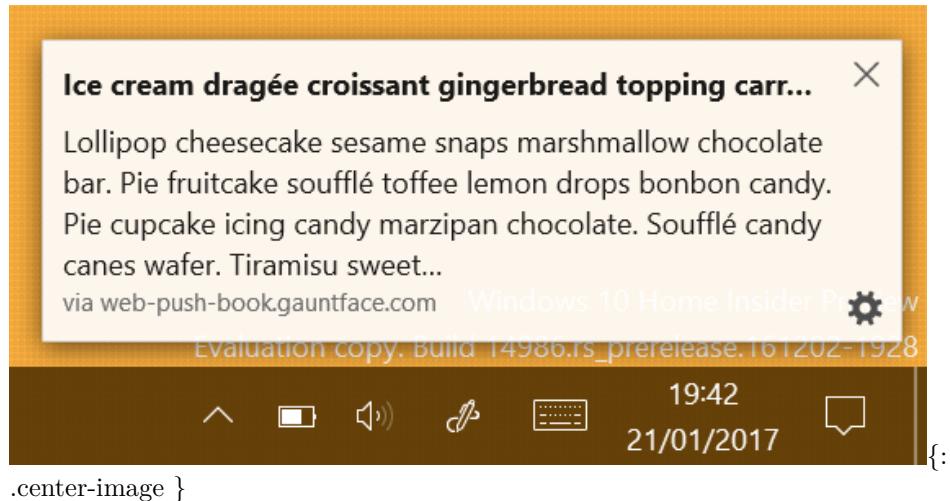


.center-image }

The reason I've included these examples is two fold. There will be differences between browsers, just looking at text, Firefox and Chrome look and act differently. Secondly there are differences across platforms. Chrome has a custom UI for all platforms whereas Firefox uses the system notifications on my Linux machine. The same notifications on Windows with Firefox look like this:



.center-image }



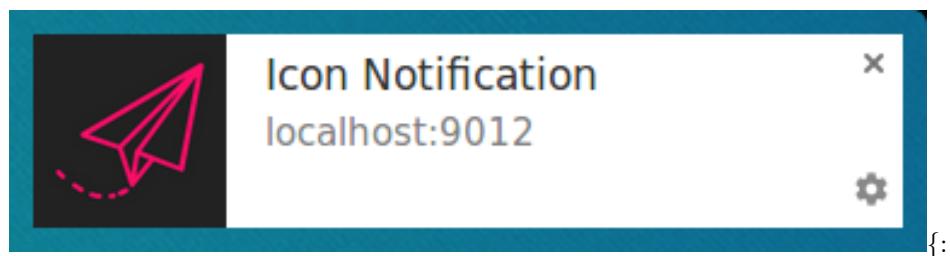
Icon

The `icon` option is essentially a small image you can show next to the title and body text.

In your code you just need to provide a URL to the image you'd like to load.

```
const title = 'Icon Notification';
const options = {
  icon: 'build/images/demos/icon-512x512.png'
};
registration.showNotification(title, options);
```

On Chrome we get this notification on Linux:



.center-image }

and on Firefox:



```
.center-image }
```

Sadly there aren't any solid guidelines for what size image to use for an icon.

[Android seems to want a 64dp image](#) (which is 64px multiples by the device pixel ratio).

If we assume the highest pixel ratio for a device will be 3, an icon size of $\geq 192\text{px}$ is a safe bet.

Note: Some browsers may require the image be served over HTTPS.

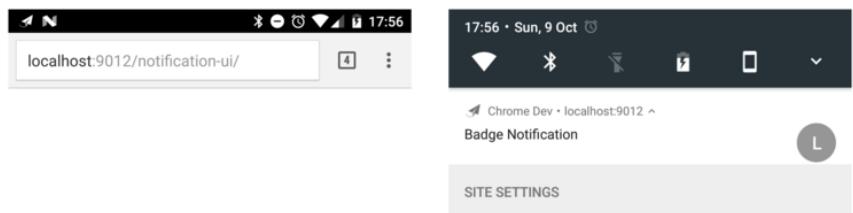
Just calling it out in case you intend to use a third party image over HTTP.

Badge

The **badge** is a small monochrome icon that is used to portray a little more information to the user about where the notification is from.

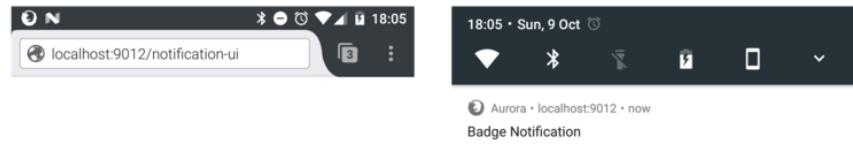
```
const title = 'Badge Notification';
const options = {
  badge: 'build/images/demos/badge-128x128.png'
};
registration.showNotification(title, options);
```

At the time of writing the badge is only used on Chrome for Android.



```
.center-image }
```

On other browsers (or Chrome without the badge), you'll see an icon of the browser.



```
.center-image {
```

```
{:
```

As with the `icon` option, there are no real guidelines on what size to use.

Digging through [Android guidelines](#) the recommended size is 24px multiplied by the device pixel ratio.

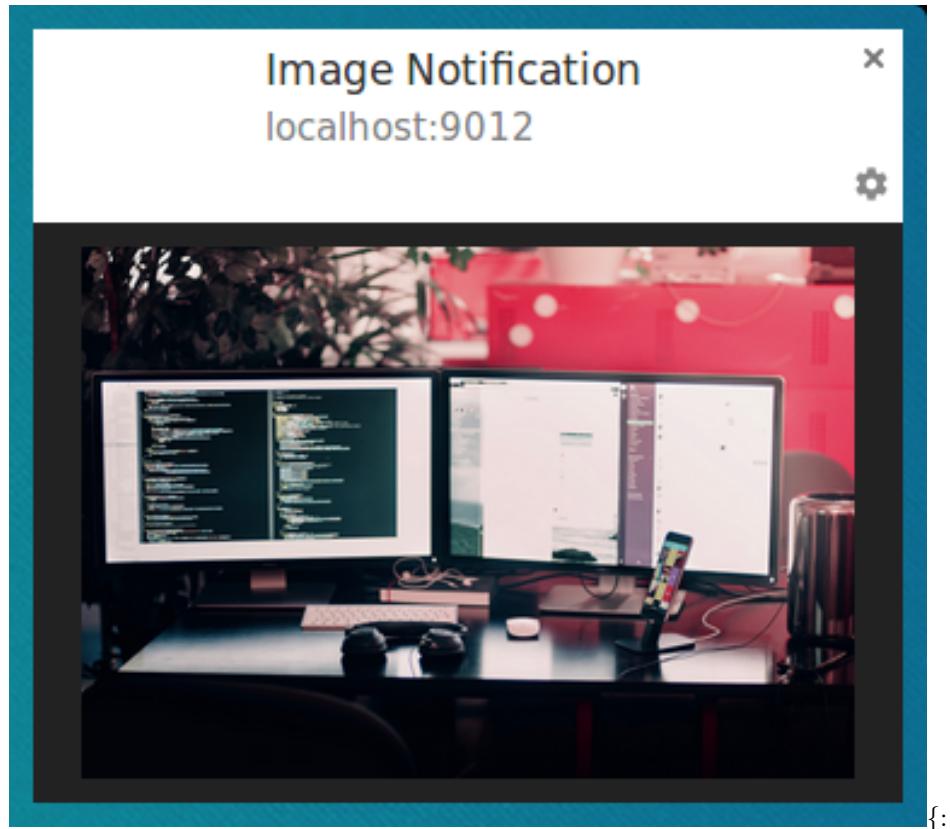
Meaning an image of $\geq 72\text{px}$ should be good (assuming a max device pixel ratio of 3).

Image

The `image` option can be used to display a larger image to the user. This is particularly useful to display a preview image to the user.

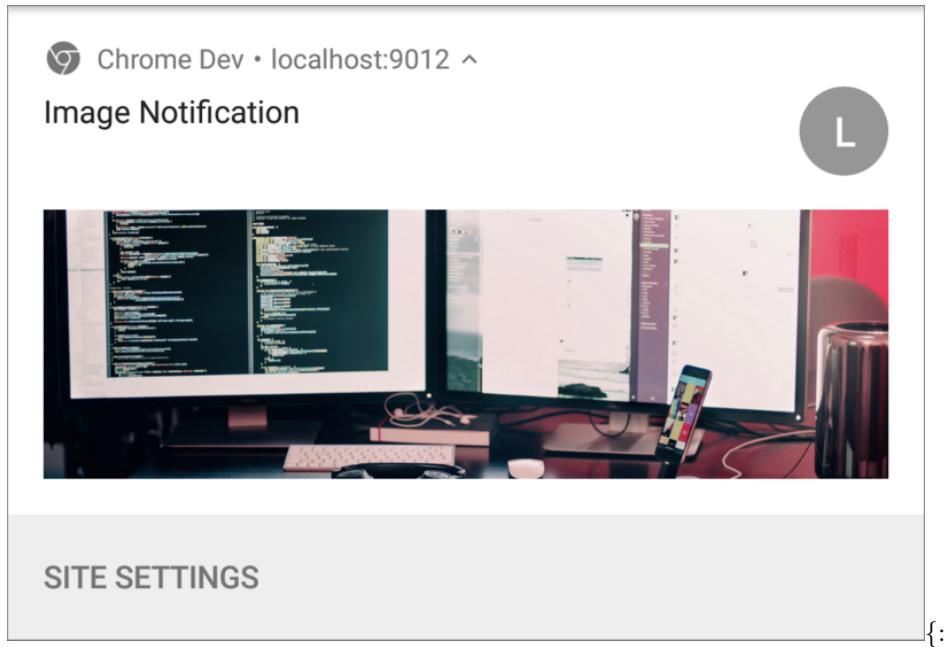
```
const title = 'Image Notification';
const options = {
  image: 'build/images/demos/unplash-farzad-nazifi-1600x1100.jpg'
};
registration.showNotification(title, options);
```

On desktop the notification will look like this:



.center-image {

On Android the cropping and ratio is different.



.center-image }

Given the differences in ratio between desktop and mobile it's extremely hard to suggest guidelines.

Since Chrome on desktop doesn't fill the available space and has a ratio of 4:3, perhaps the best approach is to serve an image with this ratio and allow Android to crop the image. That being said, the `image` option is still new and this behavior may change.

On Android, the only [guideline width](#) I could find was a width of 450dp.

Using this guideline, an image of width $\geq 1350\text{px}$ would be a good bet.

Actions

You can define actions to display buttons with a notification.

```
const title = 'Actions Notification';
const options = {
  actions: [
    {
      action: 'coffee-action',
      title: 'Coffee',
      icon: 'build/images/demos/action-1-128x128.png'
    },
    {
```

```

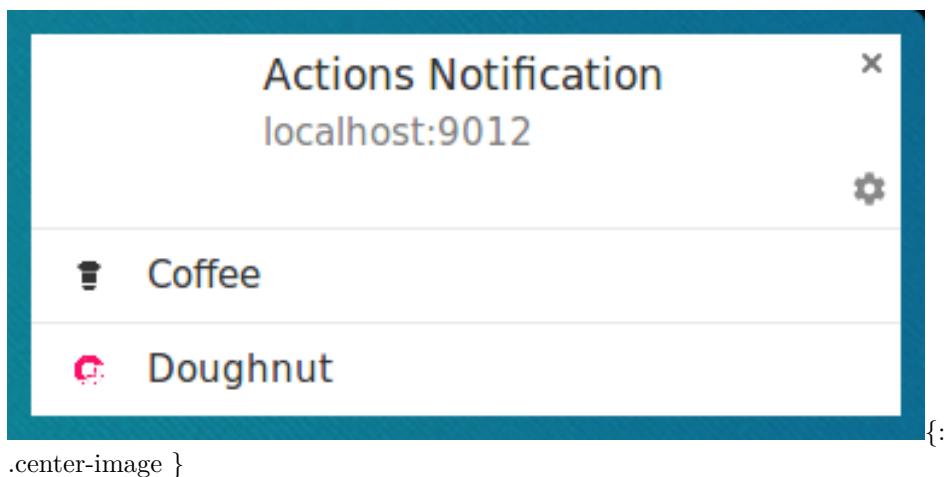
        action: 'doughnut-action',
        title: 'Doughnut',
        icon: 'build/images/demos/action-2-128x128.png'
    },
    {
        action: 'gramophone-action',
        title: 'gramophone',
        icon: 'build/images/demos/action-3-128x128.png'
    },
    {
        action: 'atom-action',
        title: 'Atom',
        icon: 'build/images/demos/action-4-128x128.png'
    }
]
};

const maxVisibleActions = Notification.maxActions;
if (maxVisibleActions < 4) {
    options.body = `This notification will only display ` +
        `${maxVisibleActions} actions.`;
} else {
    options.body = `This notification can display up to ` +
        `${maxVisibleActions} actions.`;
}

registration.showNotification(title, options);

```

At the time of writing only Chrome and Opera for Android support actions.



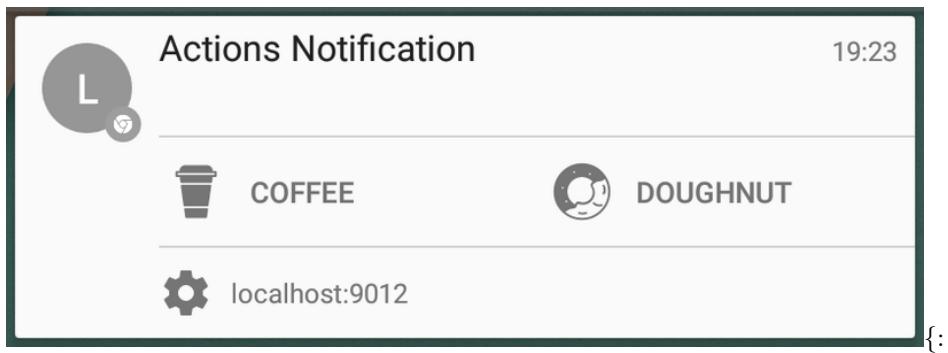
.center-image {

For each action you can define a title, an “action” (which is essentially an ID) and an icon. The title and icon is what you can see in the notification. The ID is used when detecting that the action button had been clicked (We’ll look into this more in the next section).

In the example above I’ve defined 4 actions to illustrate that you can define more actions than will be displayed. If you want to know the number actions that will be displayed by the browser you can check `Notification.maxActions`, which is used in the body text in the demo.

On desktop the action button icons display their colors (See the pink doughnut above).

On Android Marshmallow the icons are colored to match the system color scheme:



.center-image {

Chrome will hopefully change it’s behavior on desktop to match android (i.e. apply the appropriate color scheme to make the icons match the system look and feel). In the mean you can match Chrome’s text color by making your icons have a color of “#333333”..

On Android Nougat the action icons aren’t shown at all.

It’s also worth calling out that that icons look crisp on Android but **not** on desktop.

The best size I could get to work on desktop Chrome was 24px x 24px. This sadly looks out of place on Android.

The best practice we can draw from these differences:

- Stick to a consistent color scheme for your icons so at least all your icons are consistently displayed to the user.
- Make sure they work in monochrome as some platforms may display them in such a way.
- Test the size and see what works for you. 128px x 128px works well on Android for me but was poor quality on desktop.

- Expect your action icons not to be displayed at all.

The Notification spec is exploring a way to define multiple sizes of icons, but it looks like it'll be some time before anything is agreed upon.

Direction

The “dir” parameter allows you to define which direction the text should be displayed, right-to-left or left-to-right.

In testing it seemed that the direction was largely determined by the text rather than this parameter. Reading through the spec it calls out this can be used to suggest to the browser how to layout options like actions, but I saw no difference.

Probably best to define if you can, otherwise the browser should do the right thing according to the text supplied.

```
const title = '';
const options = {
  body: '',
  dir: 'rtl',
  actions: [
    {
      title: '1',
      action: 'action-1'
    },
    {
      title: '2',
      action: 'action-2'
    }
];
registration.showNotification(title, options);
```

The parameter should be set to either `auto`, `ltr` or `rtl`.

A right-to-left language used on Chrome on Linux looks like this:



.center-image }

On Firefox (while hovering over it) you'll get this:



.center-image }

Vibrate

The vibrate option allows you to define a vibration pattern that'll run when a notification is displayed, assuming the user's current settings allow for vibrations (i.e. the device isn't in silent mode).

The format of the vibrate option should be an array of numbers that describe the number of milliseconds the device should vibrate followed by the number of milliseconds the device should *not* vibrate.

```
const title = 'Vibrate Notification';
const options = {
  // Star Wars shamelessly taken from the awesome Peter Beverloo
  // https://tests.peter.sh/notification-generator/
  vibrate: [500,110,500,110,450,110,200,110,170,40,450,110,200,110,170,40,500]
};
registration.showNotification(title, options);
```

This only affects devices that support vibration.

Sound

The sound parameter allows you to define a sounds to play when the notification is received.

Sadly at the time of writing no browser has support for this option.

```
const title = 'Sound Notification';
const options = {
  sound: '/demos/notification-examples/audio/notification-sound.mp3'
};
registration.showNotification(title, options);
```

Timestamp

Timestamp allows you to tell the platform the time the notification was intended for (i.e. when the event the notification relates to occurred).

The `timestamp` should be the number of milliseconds since 00:00:00 UTC on 1 January 1970 (i.e. the unix epoch).

```
const title = 'Timestamp Notification';
const options = {
  body: 'Timestamp is set to "01 Jan 2000 00:00:00".',
  timestamp: Date.parse('01 Jan 2000 00:00:00')
};
registration.showNotification(title, options);
```

UX Best Practices

The biggest UX fail I've seen with notifications is a lack of specificity in the information displayed by a notification.

You should consider why you sent the push message in the first place and make sure all of the notification options are used to help users understand why they are reading that notification.

To be honest, it's easy to see examples and think "I'll never make that mistake" but it's easier to fall into that trap than you might think.

Some common pitfalls to avoid:

1. Don't put your website in the title or the body. Browsers include your domain in the notification so **don't duplicate it**.

Here's an example of Facebook's fallback message (you can use the DevTools 'push' button to display it yourself). Even if there is no data in the push, the important information is that there's a new notification, not "Facebook".

! [Screenshot of Facebook's Default Notification] (build/images/notification-screenshots/default-notification.png)

1. Use all information you have available to you. If you send a push message because someone sent a message to a user, rather than using a title of 'New Message' and body of 'Click here to read it.' use a title of 'John just sent a new message' and set the body of the notification to part of the message.

Here's an example from a Facebook message.



Figure 23: Screenshot of a Facebook Message Notification

It contains information on who sent the message, the message content and the users profile photo, making the notification more relevant to the user.

Browsers and Feature Detection

At the time of writing there is a pretty big disparity between Chrome and Firefox in terms of feature support for notifications.

Luckily, you can feature detect support for notification features by looking at the Notification prototype.

Let's say we wanted to know if a notification has support action buttons, we'd do the following:

```
if ('actions' in Notification.prototype) {  
    // Action buttons are supported.  
} else {  
    // Action buttons are NOT supported.  
}
```

With this, we could change the notification we display to our users.

With the other options, just do the same as above, replacing 'actions' with the desired parameter name.

Notification Behaviour

So far we've looked at the options that alter the visual appearance of a notification. There are options that alter the behaviour of notifications.

By default, calling `showNotification()` with just visual options will have the following behaviours:

1. Clicking on the notification does nothing.
2. Each new notification is shown one after the other. The browser will not collapse the notifications in anyway.
3. The platform may play a sound or vibration the users devices (depending on the platform).
4. On some platforms the notification will disappear after a short period of time while others will show the notification unless the user interacts with it (i.e. compare Android and Desktop with your notifications).

In this section we are going to look at how we can alter these default behaviours using options alone. These are relatively easy to implement and take advantage of.

Notification Click Event

When a user clicks on a notification the default behaviour is for nothing to happen, it doesn't even close / remove the notification.

The common practice for a notification click is for it to close and perform some other logic (i.e. open a window or make some API call to the application).

To achieve this we need to add a 'notificationclick' event listener to our service worker. This will be called when ever a notification is clicked.

```
self.addEventListener('notificationclick', function(event) {  
  const clickedNotification = event.notification;  
  clickedNotification.close();  
  
  // Do something as the result of the notification click  
  const promiseChain = doSomething();  
  event.waitUntil(promiseChain);  
});
```

As you can see in this example, the notification was clicked can be accessed via the `event.notification` parameter. From this we can via the properties on the notification, in this case we call it's `close()` method and then we are free to perform any task we wish in the background like a normal event.

Remember: You still need to make use of event.waitUntil() to keep the service worker running while your code is busy.

Actions

Actions allow you to give users another level of interaction with your users over just clicking the notification.

In the previous section you saw how to define actions when calling `showNotification()`:

```
const title = 'Actions Notification';
const options = {
  actions: [
    {
      action: 'coffee-action',
      title: 'Coffee',
      icon: 'build/images/demos/action-1-128x128.png'
    },
    {
      action: 'doughnut-action',
      title: 'Doughnut',
      icon: 'build/images/demos/action-2-128x128.png'
    },
    {
      action: 'gramophone-action',
      title: 'gramophone',
      icon: 'build/images/demos/action-3-128x128.png'
    },
    {
      action: 'atom-action',
      title: 'Atom',
      icon: 'build/images/demos/action-4-128x128.png'
    }
  ]
};

const maxVisibleActions = Notification.maxActions;
if (maxVisibleActions < 4) {
  options.body = `This notification will only display ` +
    `${maxVisibleActions} actions.`;
} else {
  options.body = `This notification can display up to ` +
    `${maxVisibleActions} actions.`;
}
```

```
registration.showNotification(title, options);
```

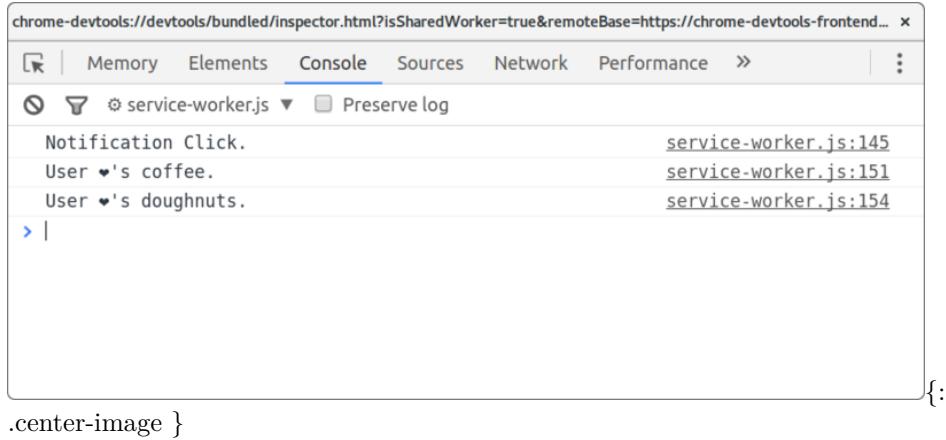
If / when the user clicks an action button, check the `event.action` value in the `notificationclick` event to tell if / which action button was clicked.

`event.action` will contain the `action` value set in the options. In the example about the `event.action` values would be one of the following: ‘coffee-action’, ‘doughnut-action’, ‘gramophone-action’ or ‘atom-action’.

With this we would detect notification clicks or action clicks like so:

```
self.addEventListener('notificationclick', function(event) {
  if (!event.action) {
    // Was a normal notification click
    console.log('Notification Click.');
    return;
  }

  switch (event.action) {
    case 'coffee-action':
      console.log('User \'s coffee.');
      break;
    case 'doughnut-action':
      console.log('User \'s doughnuts.');
      break;
    case 'gramophone-action':
      console.log('User \'s music.');
      break;
    case 'atom-action':
      console.log('User \'s science.');
      break;
    default:
      console.log(`Unknown action clicked: '${event.action}'`);
      break;
  }
});
```



The screenshot shows the Chrome DevTools Console tab with the URL `chrome-devtools://devtools/bundled/inspector.html?isSharedWorker=true&remoteBase=https://chrome-devtools-frontend...`. The console log shows three entries from `service-worker.js`:

```
Notification Click.  
User ❤'s coffee.  
User ❤'s doughnuts.
```

The log entries are timestamped with file names and line numbers: `service-worker.js:145`, `service-worker.js:151`, and `service-worker.js:154`.

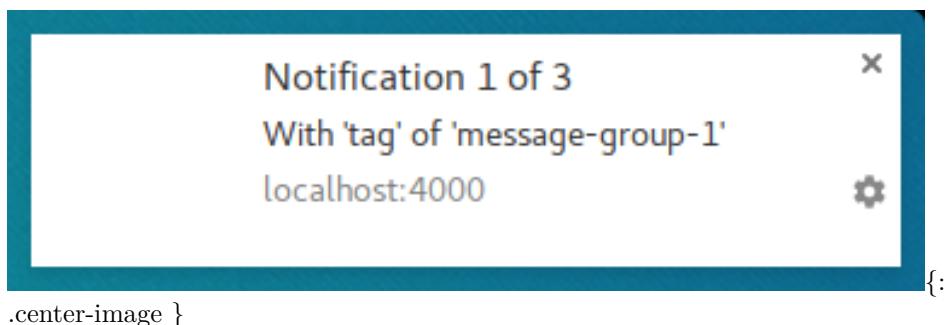
Tag

The `tag` option is essentially a String ID that “groups” notifications together, providing an easy way to determine how multiple notifications are displayed to the user. This is easiest to explain with an example.

Let's display a notification and give it a tag, of 'message-group-1'. We'd display the notification with this code:

```
const title = 'Notification 1 of 3';
const options = {
  body: 'With \'tag\' of \'message-group-1\'',
  tag: 'message-group-1'
};
registration.showNotification(title, options);
```

Which will show our first notification.



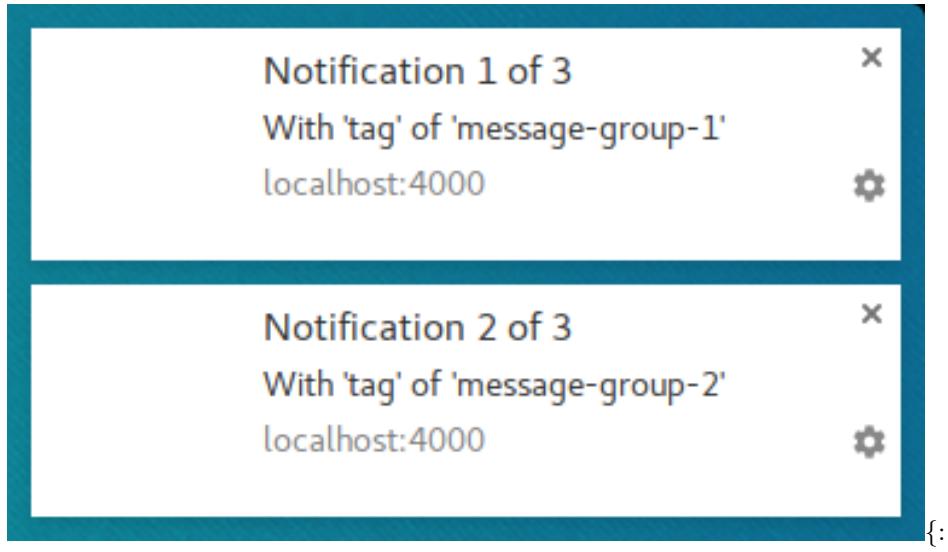
Let's display a second notification with a new tag of 'message-group-2', like so:

```

const title = 'Notification 2 of 3';
const options = {
  body: 'With \'tag\' of \'message-group-2\'',
  tag: 'message-group-2'
};
registration.showNotification(title, options);

```

This will display a second notification to the user.



.center-image } :

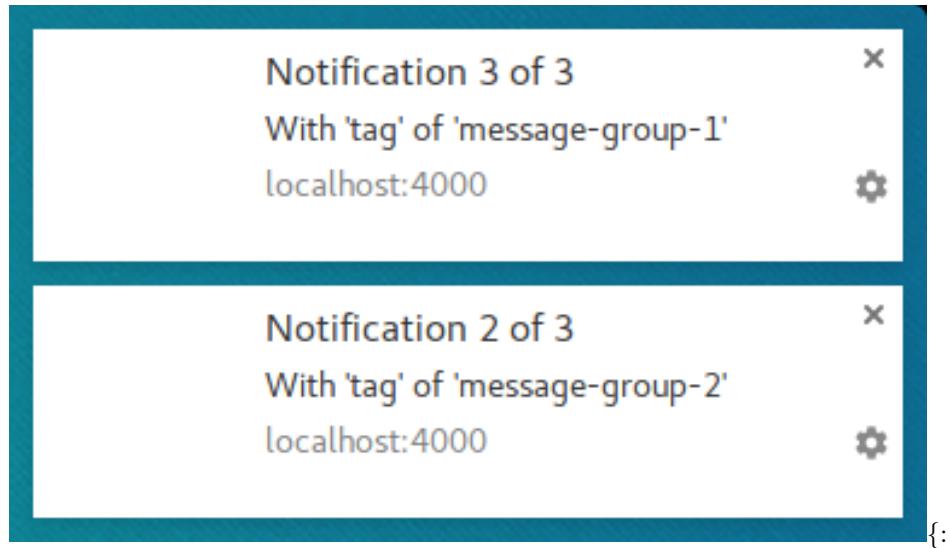
Now let's show a third notification but re-use the first tag of 'message-group-1'. Doing this will close the first notification and replace it with our new notification.

```

const title = 'Notification 3 of 3';
const options = {
  body: 'With \'tag\' of \'message-group-1\'',
  tag: 'message-group-1'
};
registration.showNotification(title, options);

```

Now we have 2 notifications even though `showNotification()` was 3 times.



```
.center-image {
```

The `tag` option is simply a way of grouping messages like this so that any old notifications that are currently displayed will be closed if they have the same tag as a new notification.

A subtlety to using `tag` is that the browser will replace any old notification without any sound and vibration that would normally be played for a new notification.

This is where the `renotify` option comes in.

Renotify

This largely applies to mobile devices at the time of writing. Setting this option makes new notifications vibrate and play a system sound.

There are scenarios where you might want a replacing notification to notify the user rather than silently update. Chat applications are a good example where you would want one notification but would want to inform the user a new message has been received. In this case you could use `tag` with `renotify` set to true.

```
const title = 'Notification 2 of 2';
const options = {
  tag: 'renotify',
  renotify: true
};
registration.showNotification(title, options);
```

Note: If you set `renotify: true` on a notification without a tag, you'll get the following error:

```
TypeError: Failed to execute 'showNotification' on 'ServiceWorkerRegistration': Notification has already been shown.
```

Silent

This option allows you to show a new notification but prevents the default behavior of vibration, sound and turning on the devices display.

This is ideal if your notifications that don't require immediate attention from the user.

```
const title = 'Silent Notification';
const options = {
  silent: true
};
registration.showNotification(title, options);
```

Note: If you define both `silent` and `renotify`, `silent` will take precedence.

Requires Interaction

Chrome on desktop will show notifications for a set time period before hiding it. Chrome on Android doesn't have this behaviour, notifications are displayed until the user interacts with it.

To force a notification to stay visible until the user has interacted with it you can define the `requireInteraction` option. This will show the notification permanently until the user dismisses / clicks your notification.

```
const title = 'Require Interaction Notification';
const options = {
  body: 'With "requireInteraction: \'true\'".',
  requireInteraction: true
};
registration.showNotification(title, options);
```

Please use this option with consideration. Showing a notification and forcing the user to stop what they are doing to dismiss your notification can be frustrating.

In the next section we are going to look at some of the common patterns used on the web for managing notifications and performing actions like opening pages when a notification is clicked.

Common Notification Patterns

There are a few common patterns that developers are implementing in their web app when it comes to web push and we're going to look at how we can implement them in our apps.

This will involve using a few different API's that are available in the service worker.

Notification Close Event

In the last section saw how we can listen for `notificationclick` events.

There is also a `notificationclose` event that is called if the user dismisses one of your notifications (i.e. rather than click the notification, the user clicks the cross or swipes the notification away).

This event is normally used for analytics to track user engagement with notifications.

```
self.addEventListener('notificationclose', function(event) {
  const dismissedNotification = event.notification;

  const promiseChain = notificationCloseAnalytics();
  event.waitUntil(promiseChain);
});
```

Adding Data to a Notification

When a push message is received it's common to have data that isn't useful the user has clicked the notification. For example, the URL that should be opened if the user clicks on a notification. The data can be passed in with the push event but it's not going to be used until the `notificationclick` event.

The easiest way to do this is to add data to a notification in the options for `showNotification()` like this:

```
const options = {
  body: 'This notification has data attached to it that is printed ' +
    'to the console when it\'s clicked.',
  tag: 'data-notification',
  data: {
    time: new Date(Date.now()).toString(),
    message: 'Hello, World!'
  }
}
```

```
};

registration.showNotification('Notification with Data', options);
```

Inside a click handler the data can be accessed with `event.notification.data`.

```
const notificationData = event.notification.data;
console.log('');
console.log('The data notification had the following parameters:');
Object.keys(notificationData).forEach((key) => {
  console.log(` ${key}: ${notificationData[key]}`);
});
console.log('');
```

Open a Window

One of the most common actions performance when a user clicks on a notification is to open a window / tab to a specific URL. We can do this with the `clients.openWindow()` API.

In our `notificationclick` event we'd run some kind like this:

```
const examplePage = '/demos/notification-examples/example-page.html';
const promiseChain = clients.openWindow(examplePage);
event.waitUntil(promiseChain);
```

In the next section we'll look at how to check if the page we want to direct the user to is already open or not. This way we can focus on the open tab rather than constantly opening new tabs.

Focus an Existing Window

When it's possible, we should focus a window rather than open a new window every time the user clicks a notification.

Before we look at how to achieve this, it's worth highlighting that this is **only possible for pages on your origin**. This is because we can only see what pages are open that belong to our site, this prevents developers from being able to see all the sites their users are viewing.

Taking the previous example where we opened '`/demos/notification-examples/example-page.html`', we'll alter it to see if it's already open or not.

```

const urlToOpen = new URL(examplePage, self.location.origin).href;

const promiseChain = clients.matchAll({
  type: 'window',
  includeUncontrolled: true
})
.then((windowClients) => {
  let matchingClient = null;

  for (let i = 0; i < windowClients.length; i++) {
    const windowClient = windowClients[i];
    if (windowClient.url === urlToOpen) {
      matchingClient = windowClient;
      break;
    }
  }

  if (matchingClient) {
    return matchingClient.focus();
  } else {
    return clients.openWindow(urlToOpen);
  }
});

event.waitUntil(promiseChain);

```

Let's step through the code.

First we parse our example page using the URL API. This is a neat trick I picked up from [Jeff Posnick](#). Calling `new URL()` with the `location` object will return an absolute URL if the string passed in is relative (i.e. `'/'` will become `'http:///'`).

We make the URL absolute so we can match it against the window URL's later on.

```
const urlToOpen = new URL(examplePage, self.location.origin).href;
```

Then we get a list of the “WindowClients”, which are the list of currently open tabs / windows (remember these are tabs for your origin only).

```

const promiseChain = clients.matchAll({
  type: 'window',
  includeUncontrolled: true
})

```

The options passed into `matchAll` just inform the browser that we only want to search for “window type” clients (i.e. just look for tabs and windows. Exclude web workers). `includeUncontrolled` is to search to tabs that aren’t controlled by the service worker this code is running in. Generally, you’ll always want this to be true when using the API for looking for windows to focus.

We capture the returned promise as `promiseChain` so that we can pass it into `event.waitUntil()` later on, keeping our service worker alive.

When the `matchAll()` promise resolves, we iterate through the returned window clients and compare the clients URL to the URL we want to open. If we find a match, we need to focus that client, which will bring that window to the users attention. Focusing is done with the `matchingClient.focus()` call.

If we can’t find a matching client, we open a new window, same as the previous section.

```
.then((windowClients) => {
  let matchingClient = null;

  for (let i = 0; i < windowClients.length; i++) {
    const windowClient = windowClients[i];
    if (windowClient.url === urlToOpen) {
      matchingClient = windowClient;
      break;
    }
  }

  if (matchingClient) {
    return matchingClient.focus();
  } else {
    return clients.openWindow(urlToOpen);
  }
});
```

Note: We are returning the promise for `matchingClient.focus()` and `clients.openWindow()` so that the promises are accounted for in our promise chain.

Merging notifications

We saw that adding a tag to a notification opts in to a behavior where any existing notification with the same tag is replaced.

You can however get more sophisticated with the collapsing of notifications using the Notifications API. Consider a chat app, where the developer might want a

new notification to show a message similar to “You have 2 messages from Matt” rather than just showing the latest message.

You can do this, or manipulate current notifications in other ways, using the `registration.getNotifications()` API which gives you access to all the currently visible notifications for your web app.

Let’s look at how we could use this API to implement the chat example.

In our chat app, let’s assume each notification has some data which includes a username.

First thing we’ll want to do is find any open notifications for a user with a specific username. We’ll get `registration.getNotifications()` and loop over them and check the `notification.data` for a specific username:

```
const promiseChain = registration.getNotifications()
.then(notifications => {
  let currentNotification;

  for(let i = 0; i < notifications.length; i++) {
    if (notifications[i].data &&
        notifications[i].data.userName === userName) {
      currentNotification = notifications[i];
    }
  }

  return currentNotification;
})
```

The next step is to replace this notification with a new notification.

In this fake message app, we’ll track the number of new messages by adding a count to our new notifications data and increment it with each new notification.

```
.then((currentNotification) => {
  let notificationTitle;
  const options = {
    icon: userIcon,
  }

  if (currentNotification) {
    // We have an open notification, let's do something with it.
    const messageCount = currentNotification.data.newMessageCount + 1;

    options.body = `You have ${messageCount} new messages from ${userName}.`;
    options.data = {
```

```

        userName: userName,
        newMessageCount: messageCount
    };
    notificationTitle = `New Messages from ${userName}`;

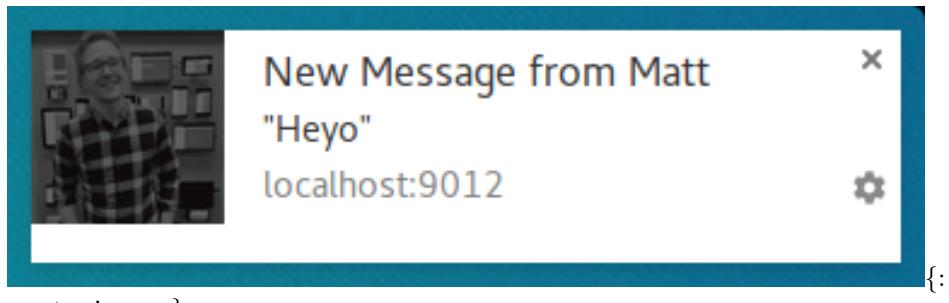
    // Remember to close the old notification.
    currentNotification.close();
} else {
    options.body = `${userMessage}`;
    options.data = {
        userName: userName,
        newMessageCount: 1
    };
    notificationTitle = `New Message from ${userName}`;
}

return registration.showNotification(
    notificationTitle,
    options
);
});
});

```

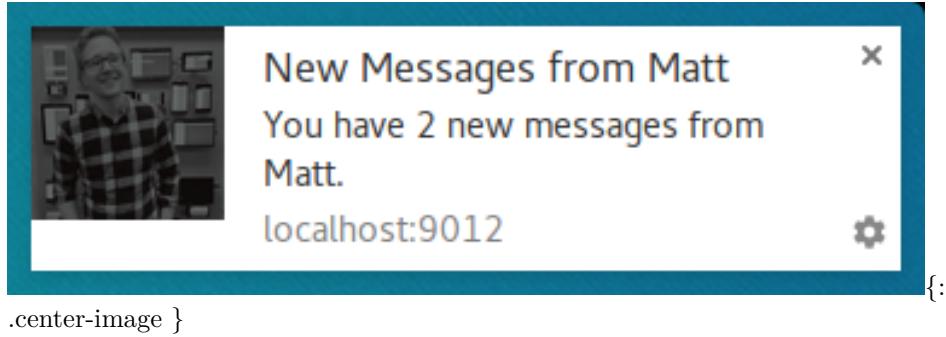
If there was a notification currently display we increment the message count and set the notification title and body message accordingly. If there were no notifications, we create a new notification with a `newMessageCount` of 1.

The end result is that the first message would look like this:



.center-image }

A second notification would collapse the notifications into this:



.center-image {

The nice thing with this approach is that if your user witnesses the notifications appearing one over the other, it'll look / feel more cohesive than just replacing with notification with the latest message.

The Exception to Rule

I've been stating that you **must** show a notification when you receive a push and this is true *most* of the time. The one scenario where you don't have to show a notification is when the user has your site open and focused.

Inside you're push event you can check whether you need to show a notification or not by examining the window clients and looking for a focused window.

The code to getting all the windows and looking for a focused window looks like this:

```
function isClientFocused() {
  return clients.matchAll({
    type: 'window',
    includeUncontrolled: true
  })
  .then((windowClients) => {
    let clientIsFocused = false;

    for (let i = 0; i < windowClients.length; i++) {
      const windowClient = windowClients[i];
      if (windowClient.focused) {
        clientIsFocused = true;
        break;
      }
    }

    return clientIsFocused;
  });
}
```

We use `clients.matchAll()` to get all of our window clients and then we loop over them checking the `focused` parameter.

Inside our push event we'd use this function to decide if we need to show a notification or not:

```
const promiseChain = isClientFocused()
.then((clientIsFocused) => {
  if (clientIsFocused) {
    console.log('Don\'t need to show a notification.');
    return;
  }

  // Client isn't focused, we need to show a notification.
  return self.registration.showNotification('Had to show a notification.');
});

event.waitUntil(promiseChain);
```

Message Page from a Push Event

We've seen that you can skip showing a notification if the user is currently on your site. But what if you still wanted to let the user know that an event has occurred, but a notification is too heavy handed?

One approach is to send a message from the service worker to the page, this way the web page can show a notification or update to the user informing them of the event. This is useful for scenarios when a subtle notification in the page is better / friendlier for the user.

Let's say we've received and pushed and checked that our web app is currently focused, we can "post a message" to each open page, like so:

```
const promiseChain = isClientFocused()
.then((clientIsFocused) => {
  if (clientIsFocused) {
    windowClients.forEach((windowClient) => {
      windowClient.postMessage({
        message: 'Received a push message.',
        time: new Date().toString()
      });
    });
  } else {
    return self.registration.showNotification('No focused windows', {
      body: 'Had to show a notification instead of messaging each page.'
    });
  }
});
```

```

        });
    }
});

event.waitUntil(promiseChain);

```

In each of pages, we listen for these messages by adding a message event listener:

```

navigator.serviceWorker.addEventListener('message', function(event) {
    console.log('Received a message from service worker: ', event.data);
});

```

In this message listener you could do anything you want, show a toast, a notification bubble or completely ignore the message.

It's also worth noting that if you don't define a message listener in your web page, the messages from the service worker will not do anything.

Cache a Page and Open Window

One scenario that is out of the scope of this book but worth just calling it is that you can improve the overall UX of your web app by caching web pages you expect users to visit after clicking on your notification.

This requires having your service worker set-up to handle `fetch` events, but if you do implement it, make sure you take advantage of it in your `push` event by caching the page and assets you'll need before showing your notification.

For more information check out this [introduction to service workers post](#).

Non-Standards Browsers

Throughout this book we've been using the application server key to identify our application with push services. This is the Web Standards approach of application identification.

In older versions of Chrome (version 51 and before), Opera for Android and the Samsung Internet Browser there was a non-standards approach for identifying your application.

In these browsers they required a `gcm_sender_id` parameter to be added to a [web app manifest](#) and the value have to be a Sender ID for a Google Developer Project.

This was completely proprietary and only required since the application server key / VAPID spec had not been defined.

In this section we are going to look at how we can add support for these browsers. Please note that this isn't recommended, but if you have a large audience on any of these browsers / browser versions, you should consider this.

What is gcm_sender_id?

The "gcm_sender_id" parameter came about in the early versions of Chrome when web push was first implemented. Google had a push service called "Google Cloud Messaging", which is now called "Firebase Cloud Messaging", both of which are commonly referred to as "GCM" and "FCM" for short.

For the Chrome team to implement push they decided to use "GCM" as it was a Google controller push service that they could work with, but it had two major drawbacks.

1. GCM needed a developer account to be set up and a "Sender ID" passed to Chrome.
2. GCM used a proprietary API to trigger messages. The API required a special "Authorization" header which is an API key the developer account, this way GCM can match the API key to the sender ID.

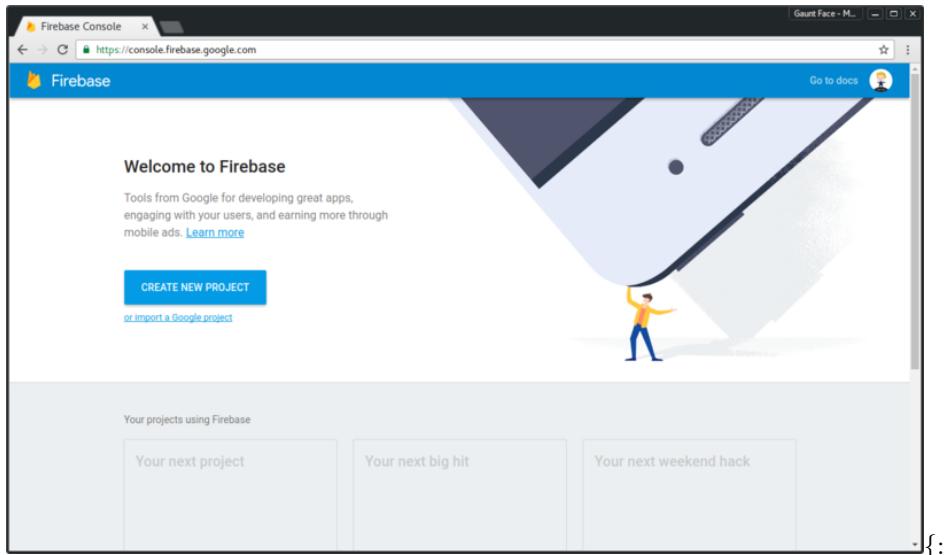
First we'll look at how to create a Firebase project to get the "Sender ID" and API key used for the "Authorization" header.

Then we'll look at adding the "gcm_sender_id" and what the API looks like to trigger a push.

Creating a Firebase Account

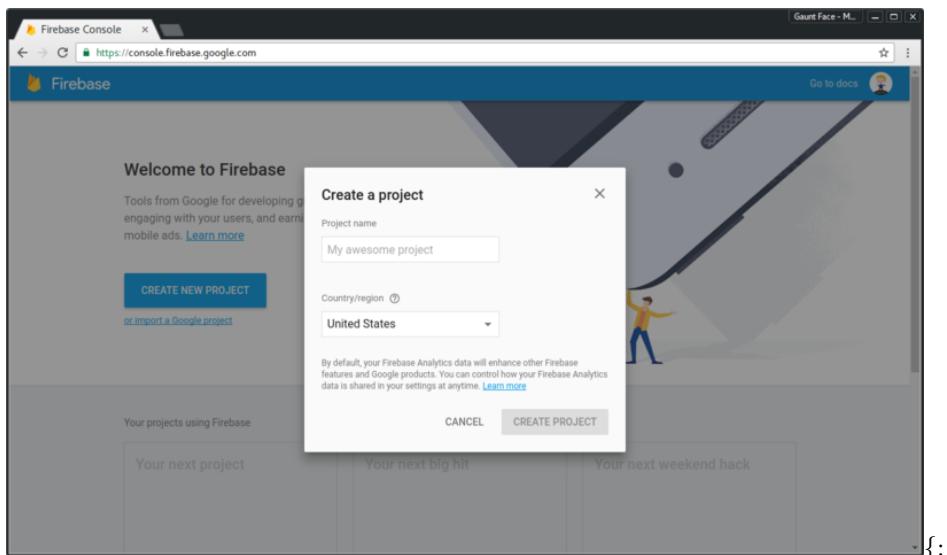
I briefly mentioned that Google Cloud Messaging was renamed to Firebase Cloud Messaging. For this reason we can use details from a Firebase project to work with the older GCM API. The reason we are doing this is that setting up a Firebase project is easier than an older Google Developer project.

The first step is to create a new Firebase project on <https://console.firebaseio.google.com>.



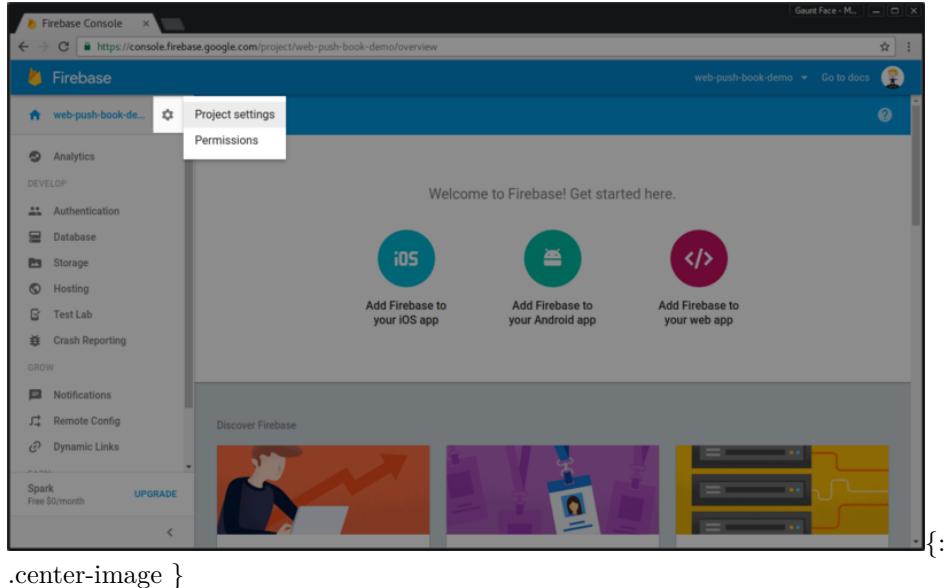
.center-image }

Creating a new project is simple, just fill in your project name and select your country.



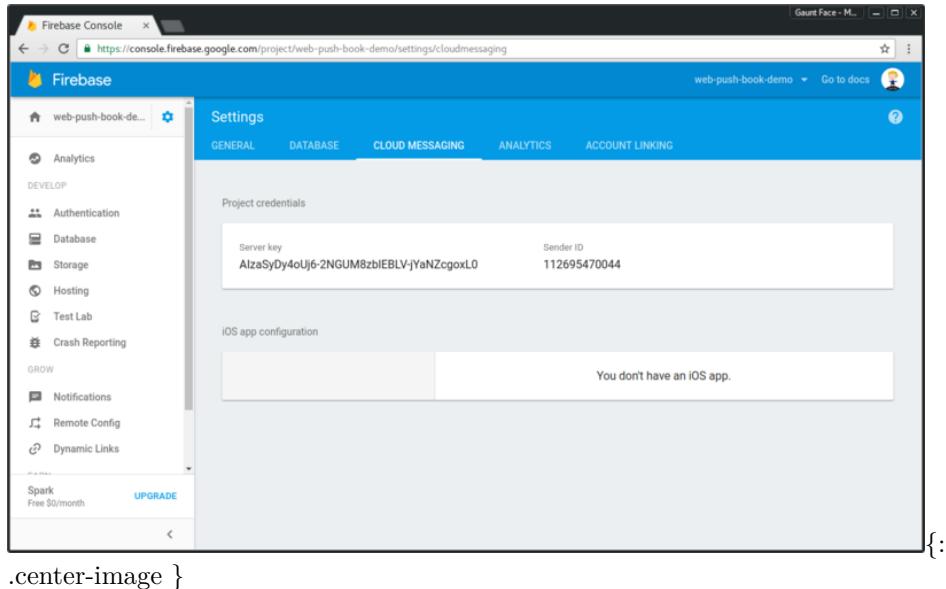
.center-image }

Once you've created your project, you'll find all the important push specific info in settings, which can be found by hovering / clicking the cog next to your projects name.



.center-image }

In settings, click on the “Cloud Messaging” tab and here you’ll find a “Server key” and a “Sender ID”. We’ll need these two pieces of information shortly.



.center-image }

Adding a Web App Manifest

The non-standards browsers will look for your sender ID in a web app manifest. For anyone who’s new to the web app manifest, it’s a JSON file that browsers

can use to gain extra information about their web app. This includes meta data like your web app's name, icon, theme color and other goodies, to learn more [check out the MDN docs](#).

For push, all we need is a JSON file with the field “gcm_sender_id” and we'll give it a value of the Sender ID from our Firebase project, like this:

```
{  
  "gcm_sender_id": "547903344792"  
}
```

Save this JSON as a file on your site, the demo for this site has a file called ‘manifest.json’ at the root of the site, i.e. ‘/manifest.json’.

Browsers will look for the manifest by looking for a “manifest” link tag in the head of our page.

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

With this set up, when `subscribe()` is called, browsers that require this will retrieve the web app manifest and use the `gcm_sender_id` value to subscribe the user to “GCM”.

If anything does wrong, you might receive an error like this:

```
Registration failed - no sender id provided
```

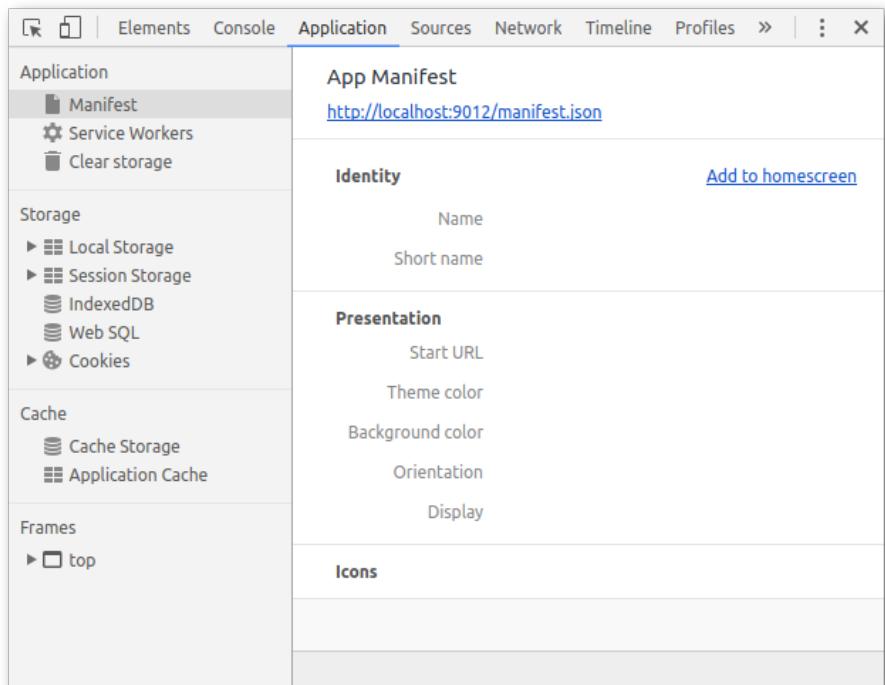
```
Registration failed - manifest empty or missing
```

```
Registration failed - gcm_sender_id not found in manifest
```

```
Failed to subscribe the user. DOMException: Registration failed - missing applicationServerKey
```

If this happens, make sure your manifest is valid JSON and make sure you have the write sender ID value.

If you are still stumped as to what the problem could be go into Chrome DevTools, select the **Application** pane, select the **Manifest** tab and click the ‘Add to homescreen’ link, this will force Chrome to get the manifest and parse it. This can often give a more helpful error message if there is a problem.



```
{:  
.center-image }
```

In Chrome, you can test whether your `gcm_sender_id` works or not by removing the `applicationServerKey` from your `subscribe()` options. This will result on Chrome using the `gcm_sender_id` as a fallback.

Using the Server Key

Once you've got a `PushSubscription`, you can tell if it's using the `gcm_sender_id` from your manifest because the endpoint for the subscription will start with `https://android.googleapis.com/gcm/send/`. This is the old endpoint for GCM.

For these `PushSubscriptions`, you can still send a Web Push Protocol request, but you need to set the **Authorization** header to `key=<Server Key from Firebase Project>`.

Remember, the 'Authorization' header would normally be the signed JWT using your application server keys, you can determine which to use based on the endpoint. Most (if not all) [Web Push libraries on Github](#) will manage this for you.

The code that does this for the Node Web Library is:

```

const isGCM = subscription.endpoint.indexOf(
  'https://android.googleapis.com/gcm/send') === 0;
if (isGCM) {
  requestDetails.headers.Authorization = 'key=' + currentGCMAPKey;
} else {
  // Add Application Server Key Details
  ...
}

```

Browser Specifics

Opera for Desktop

One thing to call out with Opera is that at the time of writing push is supported on their Android browser. On Desktop the API's are visible, but once you call subscribe, it will reject. There is no obvious way of feature detecting this sadly.

You'll need to either detect you are in Opera on desktop via user agent sniffing or simply let users go through your UI to enable push and fail at the last step.

No Payload

At the time of writing the Samsung Internet Browser doesn't support sending data with a push message (although it should do soon). This may also be the case with new browsers as they start to support web push.

This isn't necessarily a bad thing as you can make an API call when a push message is received, but for some this causes a great deal of complication. If you fall into the bracket of requiring payload support you can feature detect payload support by checking for the existence of `getKey()` on the `PushSubscription` prototype.

```

// payloadSupport is true when supported, false otherwise.
const payloadSupport = 'getKey' in PushSubscription.prototype;

```

Conclusion

Please just remember that when you see references to GCM, it's referring to a legacy, non-standard implementation of web push and will be phased out.

Only implement this *if* you have an audience from supporting browsers that warrant use of GCM.

FAQ

Why Doesn't Push Work when the Browser is Closed?

This question crops up quite a bit, largely because there are a few scenarios that make it difficult to reason with / understand.

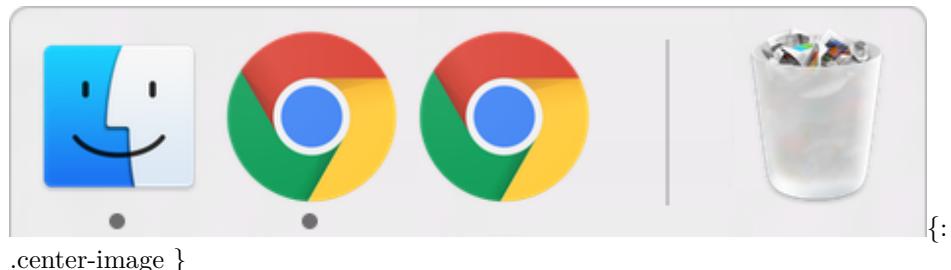
Let's start with Android. The Android OS is designed to listen for push messages and upon receiving a push message, wake up the appropriate Android app to handle the push message, regardless of whether the app is closed or not.

This is exactly the same with any browser on Android, the browser will be woken up when a push message is received and the browser will then wake up your service worker and dispatch the push event.

On desktop OS's, it's more nuanced and it's easiest to explain on Mac OS X because there is a visual indicator to help explain the different scenarios.

On Mac OS X, you can tell if a program is running or not by a marking under the app icon in the dock.

If you compare the two Chrome icons in the following dock, the one on the left is running, illustrated by the marking under the icon, whereas the Chrome on the right is **not running**, hence the lack of the marking underneath.



In the context of receiving push messages on desktop, you will receive messages when the browser is running, i.e. has the marking underneath the icon.

This means the browser can have no windows open, and you'll still receive the push message in your service worker, because the browser is running in the background.

The only time a push won't be received is if the browser is completely closed, i.e. not running at all - no marking. The same applies for Windows, although it's a little trickier to determine whether Chrome is running in the background or not.

How Do I Make My Homescreen Web App Open Fullscreen from a Push?

On Chrome for Android, a web app can be added to the home screen and when the web app is opened from the homescreen, it can launch in a fullscreen mode without the URL bar, as shown below.



{:
.center-image }

To keep this experience consistent, developers want their notifications clicked to open their web app in fullscreen as well.

Chrome “sort of” implemented support for this, although you may find it unreliable / hard to reason with. The relevant implementation detail is as follows:

Sites which have been added to homescreen on Android should be allowed to open in standalone mode in response to push notifications. As Chromium cannot detect what sites are on the homescreen after they have been added, the heuristic is sites which have been launched from homescreen within the last ten days will be opened in standalone from a tap on a notification. –[Chrome Issue](#)

What this means is that unless your user is visiting your site through the home screen icon fairly regularly, your notifications will open in the normal browser UI.

This issue will be worked on further.

Note: This is just the behavior of Chrome, other browsers may do different things as well. Feel free to [raise an issue](#) if you have anything you think should be mentioned with this question.

Why is this Any Better than Web Sockets?

The reason web push is preferable is that a service worker can be brought to life, when the browser window is closed. A web socket will only live as long as the browser and web page is kept open.

What is the deal with GCM, FCM, Web Push and Chrome?

This question has a number of facets to it and the easiest way to explain is to step through the history of web push and Chrome (Don't worry it's short).

December 2014 When Chrome first implemented web push, Chrome used Google Cloud Messaging (or GCM as it's known) to power the sending of push messages from the server to the browser.

This **was not web push**. There are a few reasons this early set-up of Chrome and GCM wasn't "real" web push.

1. GCM requires developers to set up an account on the Google Developers Console.
2. Chrome + GCM needed a special sender ID to be shared by a web app to be able to set up messaging correctly.
3. GCM's servers accepted a custom API request that wasn't a web standard.

July 2016 In July a new feature in web push landed - Application Server Keys (or VAPID as the spec is known). When Chrome added support for this new API, anyone who used it will get Chrome to use Firebase Cloud Messaging (also known as FCM) instead of GCM. This is important for a few reasons.

1. Chrome + Application Server Keys **do not** need any kind of project to be setup with Google or Firebase, it'll just work.
2. FCM supports the *web push protocol*, which is the API that all web push services will support. This means that regardless of what push service a browser uses, you just make the same kind of request and it'll send the message.

Why is it confusing today? There is a large amount of confusion now that content has been written on the topic of web push, much of which references GCM or FCM. If content references GCM, you should probably treat it as a sign that it's either old content OR it's focusing too much on Chrome (I'm guilty of doing this in a number of old posts).

Instead, think of web push as consisting of a browser, which uses a push service to manage sending and receiving message, where the push service will accept a "web push protocol" request. If you think in these terms, you can ignore which browser and which push service it's using and get to work.

This book has been written to focus on the standards approach of web push and purposefully ignores anything else.

The *only* time you should care about this back story is if and when you want to support older versions of Chrome, Samsung Internet browser and / or Opera for Android, all of whom use the older GCM trick / proprietary API's / hacks. If you want to support these browsers you'll need to implement this older GCM API which is [documented in the non-standards / legacy section of this book here](#).

Firebase has a JavaScript SDK. What and Why?

For those of you who have found the Firebase web SDK and noticed it has a messaging API for JavaScript, you may be wondering how it differs from web push.

The messaging SDK (known as Firebase Cloud Messaging JS SDK) does a few tricks behind the scenes to make it easier to implement web push.

1. Instead of worrying about PushSubscriptions and the various fields, you only need to worry about an FCM Token (A string).
2. Using the tokens for each user, you can use the proprietary FCM API to trigger push messages. This API doesn't require encrypting payloads, you can send plain payload in a POST request body.
3. FCM's proprietary API supports custom features, for example [FCM Topics](#) (It does work on web too - just poorly documented).
4. Finally FCM supports Android, iOS and web, so for some teams is easier to work with in existing projects.

This uses web push behind the scenes, but its goal is to abstract it away.

Like I said in the previous question, if you consider web push as just a browser and push service, then you can consider the Messaging SDK in Firebase as a library to try and simplify implementing web push.