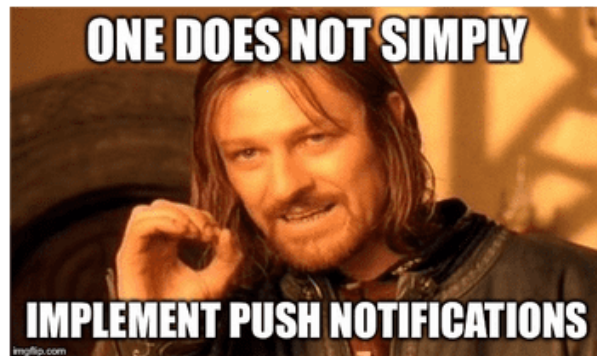


🚀🔔 Beginners guide to Web Push Notifications using Service Workers

Push notifications are very common in the native mobile application platforms like Android & iOS. They are the most effective ways to re-engage users to your apps. In this post we will look at how to implement push notifications for the web.



Notifications can be of two types:

1. Local Notification: This is generated by your app itself.
2. Push Notification: This is generated by a server via a push event.

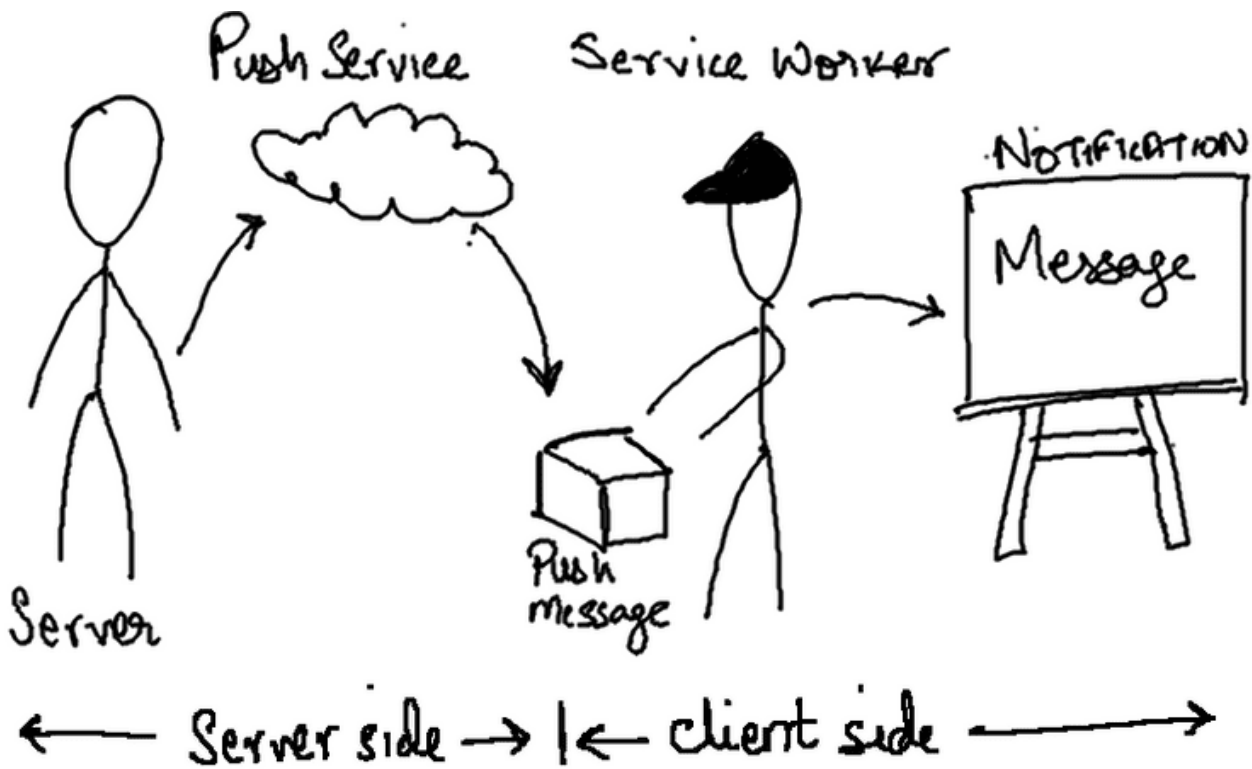
If you are here only for Push notifications, you can safely skip to **Step 3: Push Notification**

Overview

Components needed for Push Notification:

1. Service Workers

2. Notification API: This API is used to show a notification prompt to the user just like in case of mobile devices.
3. Push API: This API is used to get the push message from the server.



The steps involved in making a local push notification:

1. Ask for permission from the user using the Notification APIs **requestPermission** method.
2. If permission granted :
 - Brilliant! Now we use our service worker to subscribe to a Push Service using Push API.
 - Our Service Worker now listens for push events.
 - On arrival of a push event, service worker awakens and uses the information from the push message to show a notification using notification API.
3. If permission denied : There is nothing much you can do here. So make sure you handle this case in code as well.

Code

The entire code for this blog is available at <https://github.com/master-atul/web-push-demo>

Step 0: Boilerplate

Let's create a basic web app (no frameworks).

```
mkdir frontend
cd frontend
git init
touch index.html index.css index.js
```

Now we have a basic project structure. Let's add some basic code.

index.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <title>Push Demo</title>
  <meta name="viewport" content="width=device-width, initial-
scale=1">
  <link rel="stylesheet" type="text/css" media="screen"
href="index.css" />
  <script src="index.js"></script>
</head>
<body>
  <h1>Hello World</h1>
</body>
</html>
```

index.js

```
console.log('js is working')
```

index.css

```
body {  
  background-color: beige;  
}
```

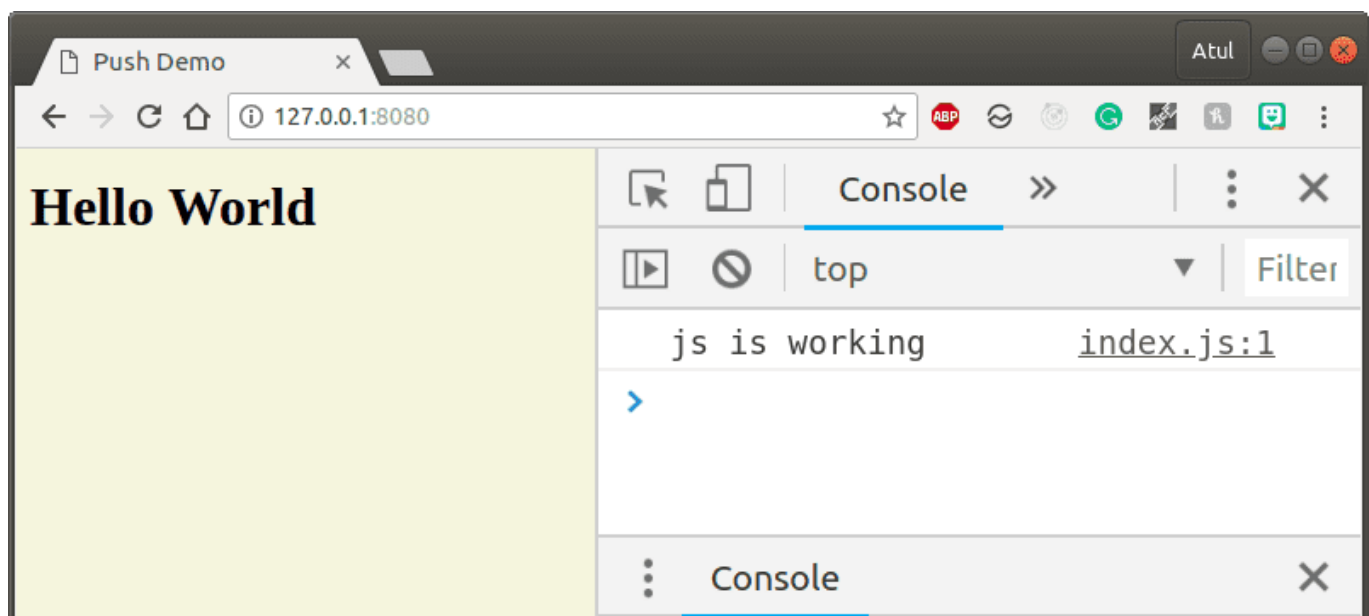
To run this project locally, let's use a simple development server. I will be using `http-server` npm module.

```
npm install -g http-server
```

Inside the frontend directory run

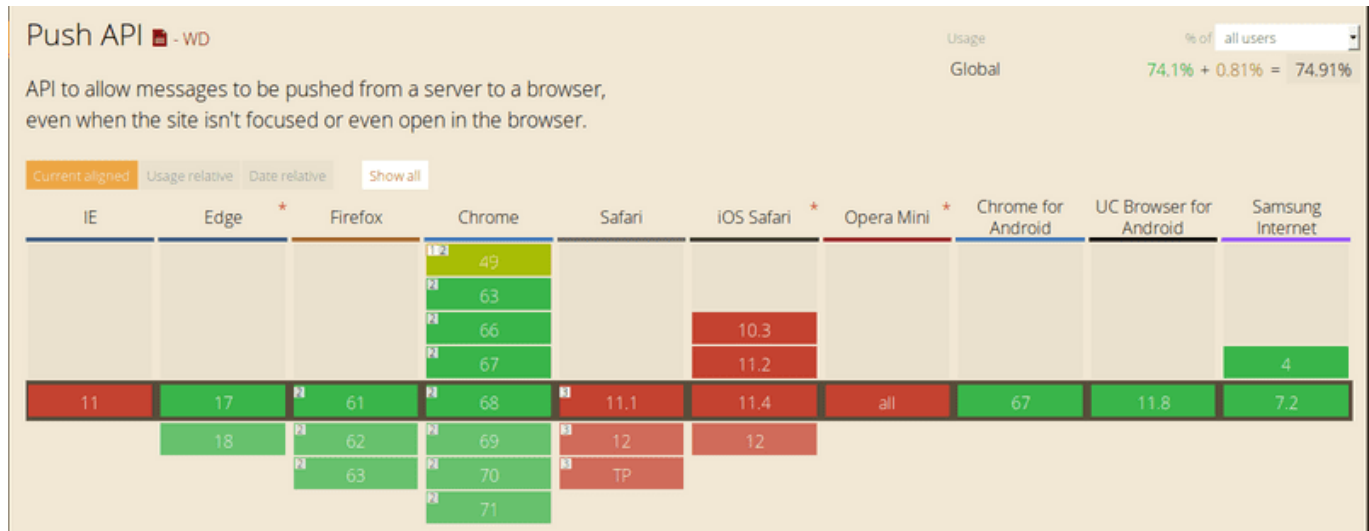
```
http-server
```

Now open up `http://127.0.0.1:8080/` on browser, you should get:



Support

To make push notifications work we need Browser Push API and Service Workers. Almost all browsers support Service Workers. Browser support for Push API is also good. Except Safari (*Has custom implementation of Push API*), all major browsers support it.



Let's check support in our code. Change the contents of **index.js** to:

index.js

```
const check = () => {
  if (!('serviceWorker' in navigator)) {
    throw new Error('No Service Worker support!')
  }
  if (!('PushManager' in window)) {
    throw new Error('No Push API Support!')
  }
}

const main = () => {
  check()
}

main()
```

If the output in the browser console doesn't show any errors at this point, you are good to go.

Step 1: Register a Service Worker and Get Permission for Notification

To register a service worker that runs in background:

First we add a new js file **service.js**: This will contain all our service worker code that will run on a separate thread.

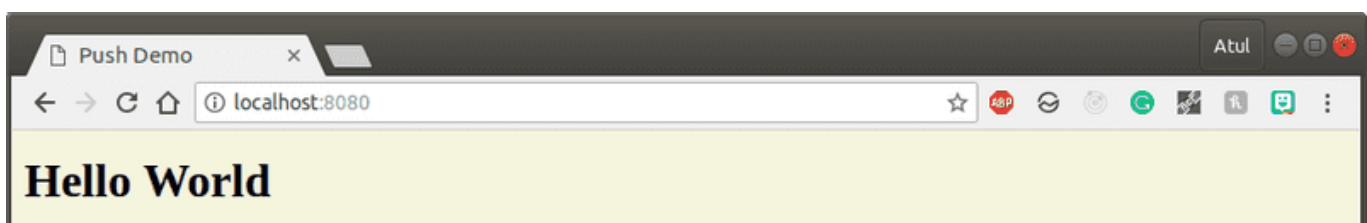
service.js

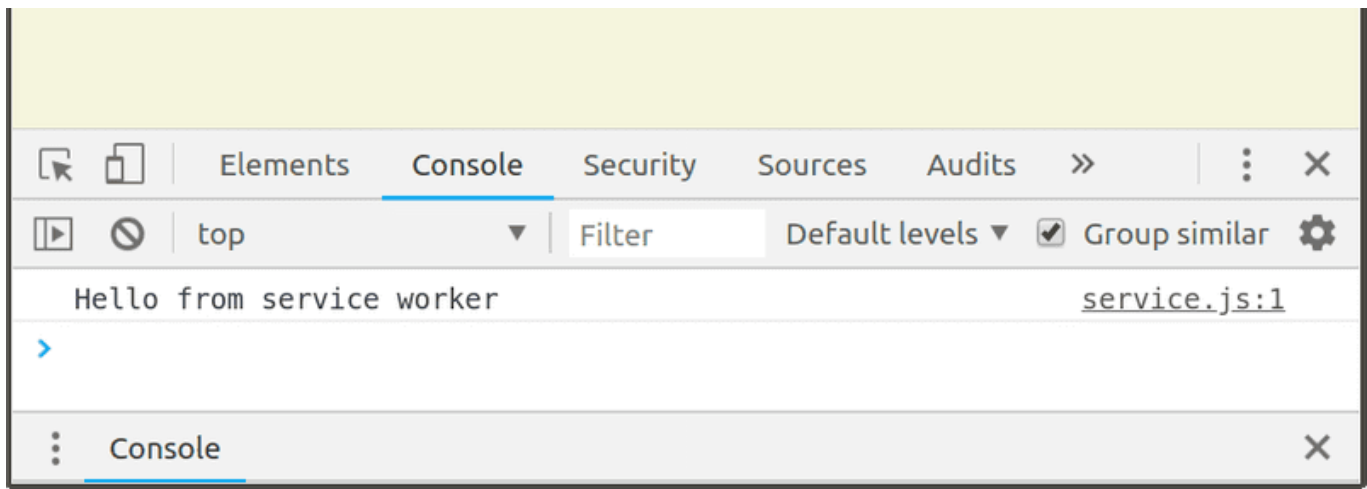
```
console.log('Hello from service worker')
```

Now modify the frontend **index.js** to:

```
const check = () => {  
  ....  
  ....  
}  
  
// I added a function that can be used to register a service worker.  
const registerServiceWorker = async () => {  
  const swRegistration = await  
navigator.serviceWorker.register('service.js'); //notice the file  
name  
  return swRegistration;  
}  
  
const main = async () => { //notice I changed main to async function  
so that I can use await for registerServiceWorker  
  check();  
  const swRegistration = await registerServiceWorker();  
}  
  
main();
```

Let's run and see:





Since service workers are registered only once, If you refresh the app using browser refresh button you will not see *Hello from service worker* again.

You can call `register()` every time a page loads without concern; the browser will figure out if the service worker is already registered or not and handle it accordingly.

As service workers are event oriented, if we need to do something useful with service workers we will need to listen to events inside it.

One subtlety with the `register()` method is the location of the service worker file. You'll notice in this case that the service worker file is at the root of the domain. This means that the service worker's scope will be the entire origin. In other words, this service worker will receive fetch events for everything on this domain. If we register the service worker file at `/example/sw.js`, then the service worker would only see fetch events for pages whose URL starts with `/example/` (i.e. `/example/page1/`, `/example/page2/`).

Debugging Tip for Service workers

- **If you are using Chrome dev tools:** You can go to **Application Tab > Service Workers**. Here you can unregister the service worker and refresh the app again. For debugging purposes, I would suggest you enable update on reload checkbox on the top to avoid manual unregister every time you change the service worker file. More detailed guide [here](#).
- **If you are using Firefox:** On the Menu bar go to **Tools > Web Developer > Service workers** or type in the URL bar: `about:debugging#workers`. This should show you list of all service workers.

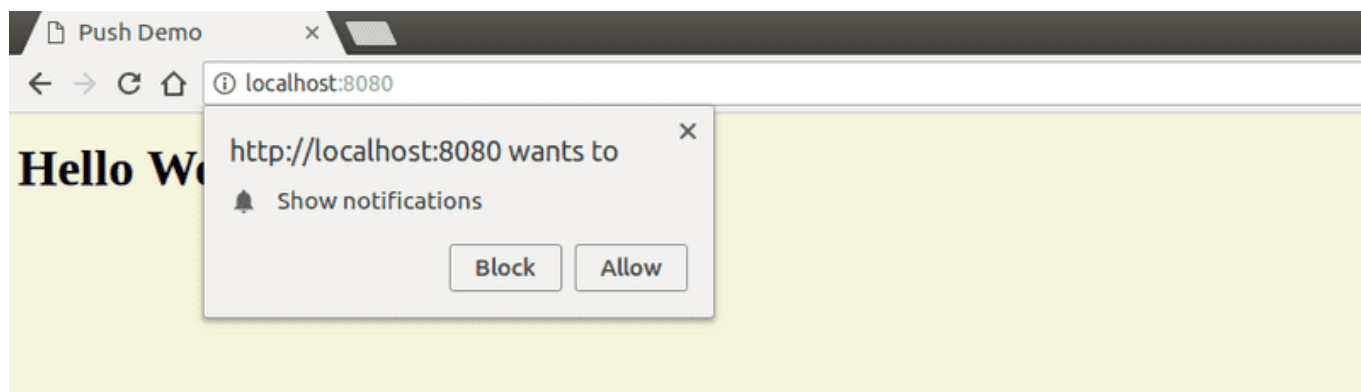
Permission for Notification

In order to show a web notification to the user, the web app needs to get permission from the user.

Modify the frontend index.js to:

```
....  
....  
  
const registerServiceWorker = async () => {  
    ....  
    ....  
}  
  
const requestNotificationPermission = async () => {  
    const permission = await window.Notification.requestPermission();  
    // value of permission can be 'granted', 'default', 'denied'  
    // granted: user has accepted the request  
    // default: user has dismissed the notification permission popup  
    // by clicking on x  
    // denied: user has denied the request.  
    if(permission !== 'granted'){  
        throw new Error('Permission not granted for Notification');  
    }  
}  
  
const main = async () => {  
    check();  
    const swRegistration = await registerServiceWorker();  
    const permission = await requestNotificationPermission();  
}  
  
main();
```

Let's refresh and see what happens.



Note: We are asking for notification permission in the main function since this is a demo app. But ideally, we should not be doing it here as it accounts for bad UX. More details on where you should be calling it in a production app here.

An alternative approach here would be to add a button asking the user to subscribe to notifications and then on click of that button we show the notification prompt.

You can also get the value of permission via

```
console.log(Notification.permission)
// This will output: granted, default or denied
```

This can be used to hide the button if the user has already answered the notification prompt (if the value is **not** 'default').

Step 2: Local Notification

Now that we have the permission from the user. We can go ahead and try out the notification popup.

To do that modify frontend index.js

```
....
....

const requestNotificationPermission = async () => {
  ....
  ....
}

const showLocalNotification = (title, body, swRegistration) => {
  const options = {
    body,
    // here you can add more properties like icon, image,
    vibrate, etc.
  }
}
```

```

    };
    swRegistration.showNotification(title, options);

}

const main = async () => {
    check();
    const swRegistration = await registerServiceWorker();
    const permission = await requestNotificationPermission();
    showLocalNotification('This is title', 'this is the message',
swRegistration);
}

main();

```

Possible values of Notification Option:

```

const options = {
  "///": "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>",

  "///": "Information Option. No visual affect.",
  "timestamp": "<Long>"
}

```

Details of each and every property is listed here

Main thread and Service Worker thread

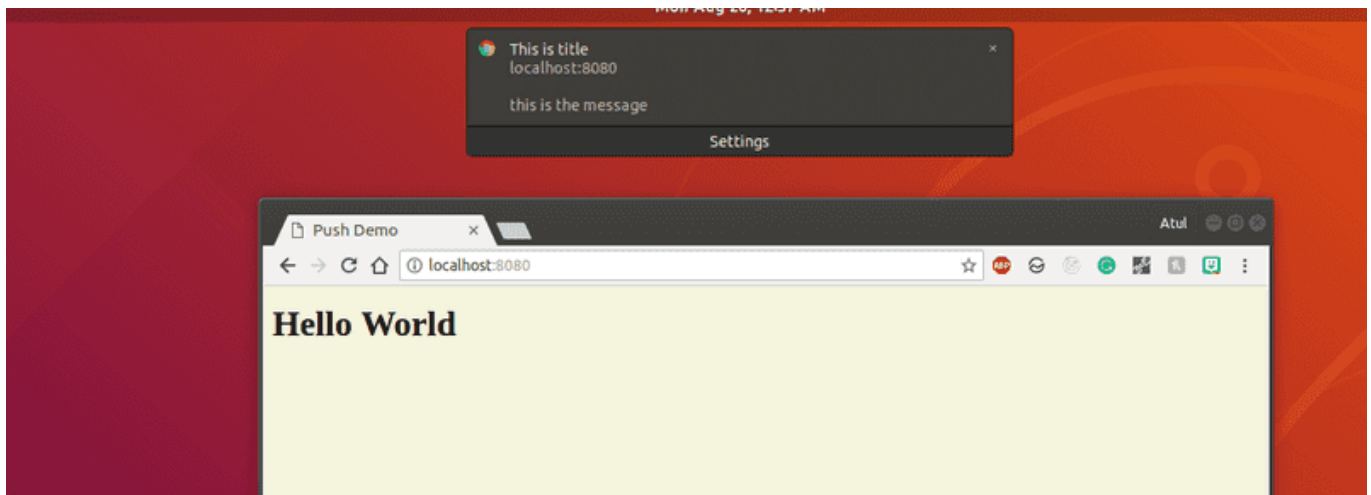
Main thread: The main JS thread that runs when we are browsing a web page with javascript.

Service Worker thread: This is an independent javascript thread which runs on the background and can run even when the page has been closed.

If you notice we haven't used our service worker till now for any purpose other than just registering it. With respect to Push Notifications, the JS main thread should only be used for all UI related stuff like changing DOM elements or asking a user, the permissions for showing notifications. Notice here that I am displaying the notification from the main thread (index.js). Ideally we will not do this in the main thread. Keep in mind that notifications are only useful for re engaging the user back to your app (ie when the page is not open). If you show notification when the user is on your page, then the user actually gets distracted.

The worker thread (service worker) should be used for waiting for push messages/events and showing the notifications. This is because it is not necessary that the actual page would be open on the browser at the time a push event arrives (meaning main js thread doesn't exist in this case).

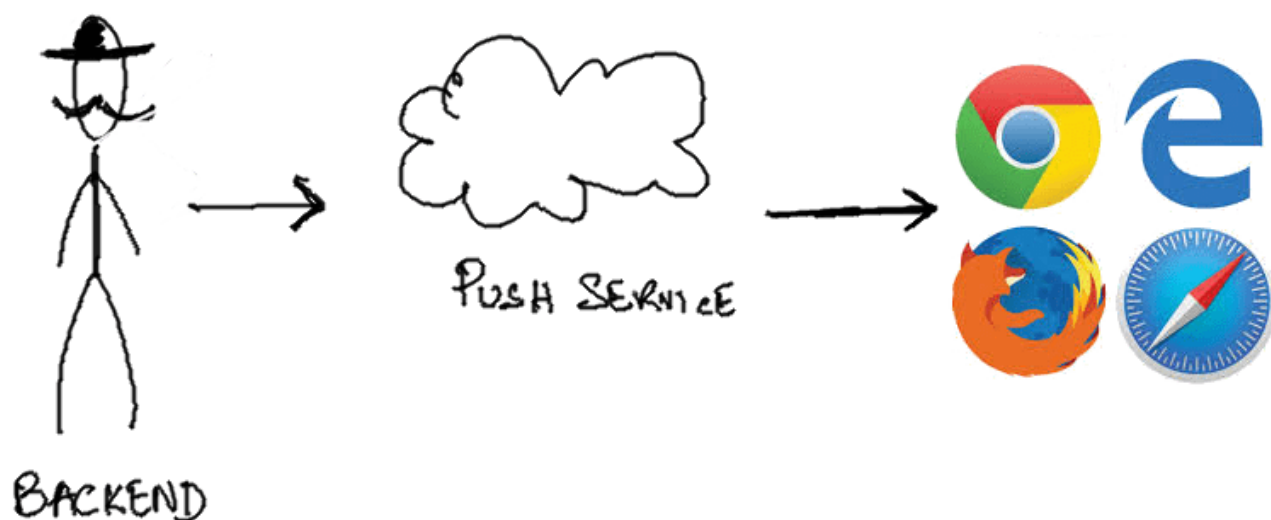
Hence, In practical use cases we will call **showNotification** from the **service.js** file. We will do that when we implement remote notifications. Here I just wanted to show you how the **showNotification** call looks like.



Step 3: Push Notification

TLDR

- We subscribe our service worker to push events from the Push Service of browser.
- We will send a message to the push service from our backend.
- Push event from push service containing the message from our backend will arrive at the browser.
- We use message from the push service to show the notification.



So What is a Push Service ?

A push service receives a network request, validates it and delivers a push message to the appropriate browser. If the browser is offline, the message is queued until the browser comes online.

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 know more: [Read up here](#)

Before we proceed

Let's clean up our code a bit as per recommendations above. Make sure our files look like this now:

index.html

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <title>Push Demo</title>
  <meta name="viewport" content="width=device-width, initial-
scale=1">
  <link rel="stylesheet" type="text/css" media="screen"
href="index.css" />
  <script src="index.js"></script>
</head>

<body>
  <h1>Hello World</h1>
  <button id="permission-btn" onclick="main()">Ask
Permission</button>
</body>

</html>
```

index.js

```
const check = () => {
  if (!('serviceWorker' in navigator)) {
    throw new Error('No Service Worker support!')
  }
  if (!('PushManager' in window)) {
    throw new Error('No Push API Support!')
  }
}

const registerServiceWorker = async () => {
  const swRegistration = await
navigator.serviceWorker.register('service.js')
  return swRegistration
}
```

```

const requestNotificationPermission = async () => {
  const permission = await window.Notification.requestPermission()
  // value of permission can be 'granted', 'default', 'denied'
  // granted: user has accepted the request
  // default: user has dismissed the notification permission popup by
  clicking on x
  // denied: user has denied the request.
  if (permission !== 'granted') {
    throw new Error('Permission not granted for Notification')
  }
}

const main = async () => {
  check()
  const swRegistration = await registerServiceWorker()
  const permission = await requestNotificationPermission()
}
// main(); we will not call main in the beginning.

```

service.js

```

self.addEventListener('activate', async () => {
  // This will be called only once when the service worker is
  activated.
  console.log('service worker activate')
})

```

Let's run this: You should now see a button on the page saying 'Ask Permission'. Clicking on it would call our main function.

Listen to Remote Notification

We will now only focus on the service worker file from now on unless specified otherwise.

To listen/subscribe to remote push messages we need the browser web app to register to the push service.

service.js

```

self.addEventListener('activate', async () => {
  // This will be called only once when the service worker is
  activated.

```

```

    try {
      const options = {}
      const subscription = await
self.registration.pushManager.subscribe(options)
      console.log(JSON.stringify(subscription))
    } catch (err) {
      console.log('Error', err)
    }
  })

```

Now, unregister the service workers and refresh the web app. Click on Ask for permission button.

In Firefox console you should see:

```

{
  "endpoint": "https://updates.push.services.mozilla.com:443/wpush/v1/<some_id>",
  "keys": {
    "auth": "<some key>",
    "p256dh": "<some key>"
  }
}

```

The value of subscription will be null if the web app has not subscribed to the push service.

In Chrome console you would see an error:

```

Error DOMException: Registration failed - missing
applicationServerKey, and gcm_sender_id not found in manifest

```

This is because, there are two option parameters **applicationServerKey** (also known as **VAPID public key**) and **userVisibleOnly** which are optional in Firefox but are required parameters in chrome.

userVisibleOnly: You need to send userVisibleOnly as true always in chrome. This parameter restricts the developer to use push messages for notifications. That is the

developer will not be able use the push messages to send data to server silently without showing a notification. Currently it has to be set to true otherwise you get a permission denied error. In essence, silent push messages are not supported at the moment due to privacy and security concerns.

VAPID: *Voluntary Application Server Identification for Web Push* is a spec that allows a backend server(your application server) to identify itself to the Push Service(browser specific service). It is a security measure that prevents anyone else from sending messages to an application's users.

How does VAPID keys enable security ?

In short:

1. You generate a set of private and public keys (VAPID keys) at the application server(for example:your backend NodeJS server).
2. Public key will be sent to the push service when the frontend app tries to subscribe to the push service. Now the push service knows the public key from your application server(backend). The subscribe method will return a unique endpoint for you frontend web app. You will store this unique endpoint at the backend application server.
3. From the application server, you will hit an API request to the unique push service endpoint that you just saved. In this API request you will sign some information in the Authorization header with the private key. This allows the push service to verify that the request came from the correct application server.
4. After validating the header, the push service will send the message to the frontend web app.

PS: To get the value of push subscription you can also call (in your service worker file)

```
const subscription = await  
self.registration.pushManager.getSubscription()
```

Generating VAPID Keys

To generate a set of private and public VAPID keys:

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

This should print out something like this:

=====

Public Key:

*BJ5IxJBWdeqFDJTvrZ4wNRu7UY2XigDXjgiUBYEYVXDudxhEs0ReOJRBcBHsPYgZ5dyV8
VjyqzbQKS8V7bUAglk*

Private Key:

ERIZmc5T5uWGeRxedxu92k3HnpVwy_RCNQfgek1x2Y4

=====

DO NOT USE THE ABOVE KEYS, GENERATE YOUR OWN.

You need to only generate this once. Keep your private key safe. Public key can be stored on frontend web app.

Now let's get back to **service.js**

```
// urlB64ToUint8Array is a magic function that will encode the base64
public key
// to Array buffer which is needed by the subscription option
const urlB64ToUint8Array = base64String => {
  const padding = '='.repeat((4 - (base64String.length % 4)) % 4)
  const base64 = (base64String + padding).replace(/\-/g,
'+').replace(/_/g, '/')
  const rawData = atob(base64)
  const outputArray = new Uint8Array(rawData.length)
  for (let i = 0; i < rawData.length; ++i) {
    outputArray[i] = rawData.charCodeAt(i)
  }
  return outputArray
}

self.addEventListener('activate', async () => {
  // This will be called only once when the service worker is
```

```

activated.
    try {
        const applicationServerKey = urlB64ToUint8Array(

'BJ5IxBWdeqFDJTvrZ4wNRu7UY2XigDXjgiUBYEYVXDudxhEs0ReOJRBcBHsPYgZ5dyV
8VjyqzbQKS8V7bUAglk'
        )
        const options = { applicationServerKey, userVisibleOnly: true }
        const subscription = await
self.registration.pushManager.subscribe(options)
        console.log(JSON.stringify(subscription))
    } catch (err) {
        console.log('Error', err)
    }
})

```

Now refresh and unregister/re-register the service worker. Click on “Ask Permission”. You should see in your console.

```

{
  "endpoint" : "https://fcm.googleapis.com/fcm/send/<some id>",
  "expirationTime" : null,
  "keys": {
    "p256dh" : "<some key>",
    "auth" : "<some key>"
  }
}

```

This is quite similar to the one we received from the Firefox subscription as well. Notice that **fcm.googleapis.com** (firebase) is the push service used by google chrome while Firefox uses **updates.push.services.mozilla.com**. The Web Push Protocol standardises the Push API. This enables each browser to use the push service it deems fit as long as it conforms to the same API as mentioned in the spec.

Now once we receive the value of subscription, we need to save it in our backend server (application server). To do so we should create an API end point on our application server that will take in the subscription and store it in the database/cache. Let’s for now consider that the API to hit is **POST** <http://localhost:3000/save-subscription> with request body containing the entire subscription object. We will make this API on the backend server soon.

So let's change the **service.js** file to add functionality to save the subscription.

```
// urlB64ToUint8Array is a magic function that will encode the base64
public key
// to Array buffer which is needed by the subscription option
const urlB64ToUint8Array = base64String => {
  const padding = '='.repeat((4 - (base64String.length % 4)) % 4)
  const base64 = (base64String + padding).replace(/\-/g,
'+').replace(/_/g, '/')
  const rawData = atob(base64)
  const outputArray = new Uint8Array(rawData.length)
  for (let i = 0; i < rawData.length; ++i) {
    outputArray[i] = rawData.charCodeAt(i)
  }
  return outputArray
}

// saveSubscription saves the subscription to the backend
const saveSubscription = async subscription => {
  const SERVER_URL = 'http://localhost:4000/save-subscription'
  const response = await fetch(SERVER_URL, {
    method: 'post',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(subscription),
  })
  return response.json()
}

self.addEventListener('activate', async () => {
  // This will be called only once when the service worker is
  activated.
  try {
    const applicationServerKey = urlB64ToUint8Array(
'BJ5IxJBWdeqFDJTvrZ4wNRu7UY2XigDXjgiUBYEYVXDudxhEs0ReOJRBcBHsPYgZ5dyV
8VjyqzbQKS8V7bUAglk'
)
    const options = { applicationServerKey, userVisibleOnly: true }
    const subscription = await
self.registration.pushManager.subscribe(options)
    const response = await saveSubscription(subscription)
    console.log(response)
  } catch (err) {
    console.log('Error', err)
  }
})
```

Next step is to save subscription at the backend and then use the subscription to send push messages to the frontend app via the push service. But before that, we will also need to listen to the push event from the backend on our frontend app.

Listen to Push event

In your **service.js** add

```
self.addEventListener('push', function(event) {
  if (event.data) {
    console.log('Push event!! ', event.data.text())
  } else {
    console.log('Push event but no data')
  }
})
```

Time for backend (NodeJS)

Let's create a basic express js project.

```
mkdir backend
cd backend
npm init
npm install --save express
```

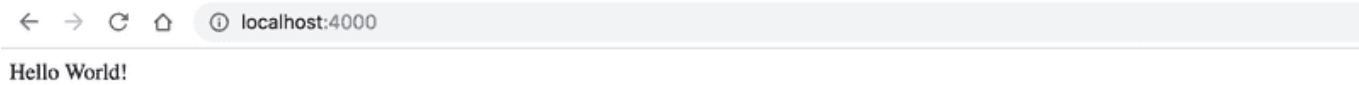
Now create a backend **index.js** file

```
const express = require('express')
const cors = require('cors')
const bodyParser = require('body-parser')
const app = express()
app.use(cors())
app.use(bodyParser.json())
const port = 4000
app.get('/', (req, res) => res.send('Hello World!'))
app.listen(port, () => console.log(`Example app listening on port ${port}!`))
```

Run the node server as shown below and open up `localhost:4000` on the browser

```
node index.js
```

You should see something like this:

A screenshot of a web browser window. The address bar shows 'localhost:4000'. The page content displays 'Hello World!' in a simple black font on a white background.

```
< > ↻ 🏠 localhost:4000  
Hello World!
```

Saving subscription on the backend: Let's create our save-subscription endpoint on the express server. All we need to do is retrieve the subscription from frontend and stored it somewhere safely in the backend. We will need this subscription later to send push messages to the user's browser.

Make changes to your backend nodejs **index.js** file :

```
const express = require('express')  
const cors = require('cors')  
const bodyParser = require('body-parser')  
  
const app = express()  
app.use(cors())  
app.use(bodyParser.json())  
  
const port = 4000  
  
app.get('/', (req, res) => res.send('Hello World!'))  
  
const dummyDb = { subscription: null } //dummy in memory store  
  
const saveToDatabase = async subscription => {  
  // Since this is a demo app, I am going to save this in a dummy in  
  memory store. Do not do this in your apps.  
  // Here you should be writing your db logic to save it.
```

```

    dummyDb.subscription = subscription
  }

  // The new /save-subscription endpoint
  app.post('/save-subscription', async (req, res) => {
    const subscription = req.body
    await saveToDatabase(subscription) //Method to save the
    subscription to Database
    res.json({ message: 'success' })
  })

  app.listen(port, () => console.log(`Example app listening on port
  ${port}!`))

```

Now we have saved the subscription at the backend, next step is to finally send a push message to the frontend.

Generate Remote Notification

To generate a push message we will use a npm library web-push which will help us encrypt the message, setting the correct parameters, legacy support for browsers etc. If you need to figure out how to do this in languages other than Javascript(NodeJS), you can take a look at the web-push source code.

In your backend project, install the npm module **web-push** locally.

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

In the backend **index.js** add the following:

```

...
...
...
const webpush = require('web-push') //requiring the web-push module
...
...
...
const app = express();
...
...
...

```

```

const vapidKeys = {
  publicKey:

'BJ5IxJBWdeqFDJTvrZ4wNRu7UY2XigDXjgiUBYEYVXDudxhEs0ReOJRBCBHsPYgZ5dyV
8VjyqzbQKS8V7bUAglk',
  privateKey: 'ERIZmc5T5uWGeRxedxu92k3HnpVwy_RCNQfgek1x2Y4',
}

//setting our previously generated VAPID keys
webpush.setVapidDetails(
  'mailto:myuserid@email.com',
  vapidKeys.publicKey,
  vapidKeys.privateKey
)

//function to send the notification to the subscribed device
const sendNotification = (subscription, dataToSend='') => {
  webpush.sendNotification(subscription, dataToSend)
}

```

Explanation

webpush.setVapidDetails takes three arguments.

- First argument needs to be either a URL or a mailto email address.
- Second argument is the VAPID public key we generated earlier.
- Third argument is the VAPID private key.

To test this out let's add another route **/send-notification** to our NodeJS express server. The complete backend **index.js** file would look something like this.

```

const express = require('express')
const cors = require('cors')
const bodyParser = require('body-parser')
const webpush = require('web-push')

const app = express()
app.use(cors())
app.use(bodyParser.json())

const port = 4000

app.get('/', (req, res) => res.send('Hello World!'))

const dummyDb = { subscription: null } //dummy in memory store

```

```

const saveToDatabase = async subscription => {
  // Since this is a demo app, I am going to save this in a dummy in
  memory store. Do not do this in your apps.
  // Here you should be writing your db logic to save it.
  dummyDb.subscription = subscription
}

// The new /save-subscription endpoint
app.post('/save-subscription', async (req, res) => {
  const subscription = req.body
  await saveToDatabase(subscription) //Method to save the
  subscription to Database
  res.json({ message: 'success' })
})

const vapidKeys = {
  publicKey:

'BJ5IxJBWdegFDJTvrZ4wNRu7UY2XigDXjgiUBYEYVXDudxhEs0ReOJRBcBHsPYgZ5dyV
8VjyqzbQKS8V7bUAglk',
  privateKey: 'ERIZmc5T5uWGeRxedxu92k3HnpVwy_RcNqfgek1x2Y4',
}

//setting our previously generated VAPID keys
webpush.setVapidDetails(
  'mailto:myuserid@email.com',
  vapidKeys.publicKey,
  vapidKeys.privateKey
)

//function to send the notification to the subscribed device
const sendNotification = (subscription, dataToSend) => {
  webpush.sendNotification(subscription, dataToSend)
}

//route to test send notification
app.get('/send-notification', (req, res) => {
  const subscription = dummyDb.subscription //get subscription from
  your databse here.
  const message = 'Hello World'
  sendNotification(subscription, message)
  res.json({ message: 'message sent' })
})

app.listen(port, () => console.log(`Example app listening on port
${port}!`))

```

Let's try it out!



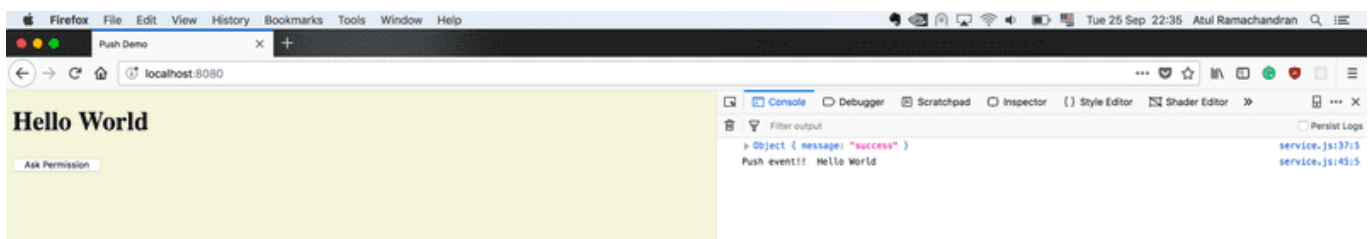
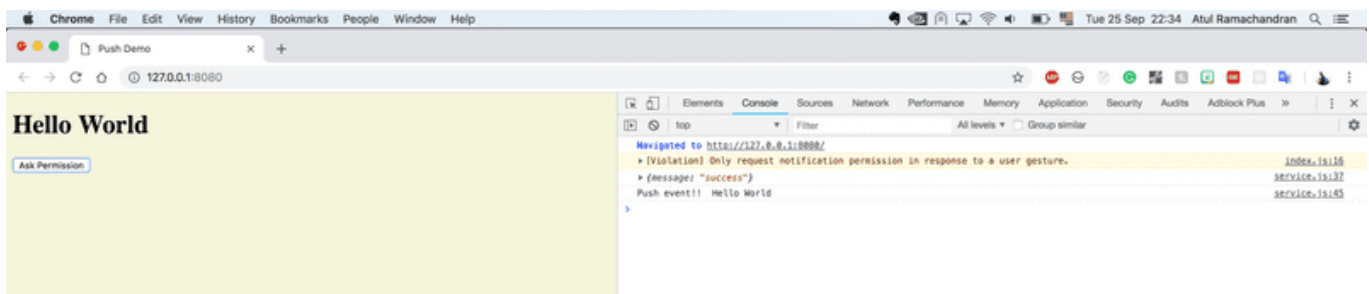

```
Example app listening on port 4000!  
[...]  
Starting up http-server, serving ./  
Available on:  
  http://127.0.0.1:8080  
  http://172.25.9.70:8080  
Hit CTRL-C to stop the server  
[...]
```

Let's open up `http://127.0.0.1:8080` in the browser

Make sure you start from clean slate. Unregister service workers, clear up cache, etc.

Now hit **ask permission** and you should see a console message success. Which means our push subscription has been saved in backend.

Now open up the backend route `http://localhost:4000/send-notification` in another tab in your browser (since it is a GET API). You should see hello world push message on the console of the frontend app.



Adding final touches: Show push messages as a notification

Open up your frontend `service.js` and add the following:

```
...  
...  
...  
self.addEventListener("push", function(event) {
```

```

    if (event.data) {
      console.log("Push event!! ", event.data.text());
      showLocalNotification("Yolo", event.data.text(),
self.registration);
    } else {
      console.log("Push event but no data");
    }
  });

const showLocalNotification = (title, body, swRegistration) => {
  const options = {
    body
    // here you can add more properties like icon, image, vibrate,
    etc.
  };
  swRegistration.showNotification(title, options);
};

```

The final **service.js** file looks like this:

```

// urlB64ToUint8Array is a magic function that will encode the base64
public key
// to Array buffer which is needed by the subscription option
const urlB64ToUint8Array = base64String => {
  const padding = '='.repeat((4 - (base64String.length % 4)) % 4)
  const base64 = (base64String + padding).replace(/\-/g,
'+').replace(/_/g, '/')
  const rawData = atob(base64)
  const outputArray = new Uint8Array(rawData.length)
  for (let i = 0; i < rawData.length; ++i) {
    outputArray[i] = rawData.charCodeAt(i)
  }
  return outputArray
}

const saveSubscription = async subscription => {
  const SERVER_URL = 'http://localhost:4000/save-subscription'
  const response = await fetch(SERVER_URL, {
    method: 'post',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(subscription),
  })
  return response.json()
}

self.addEventListener('activate', async () => {
  // This will be called only once when the service worker is

```

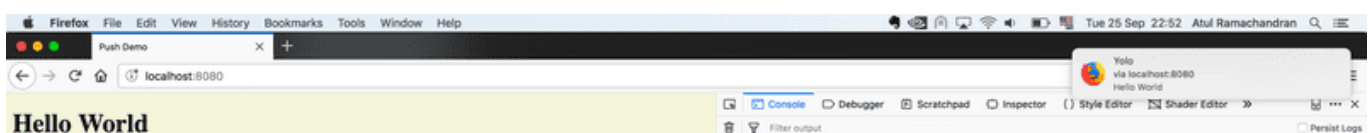
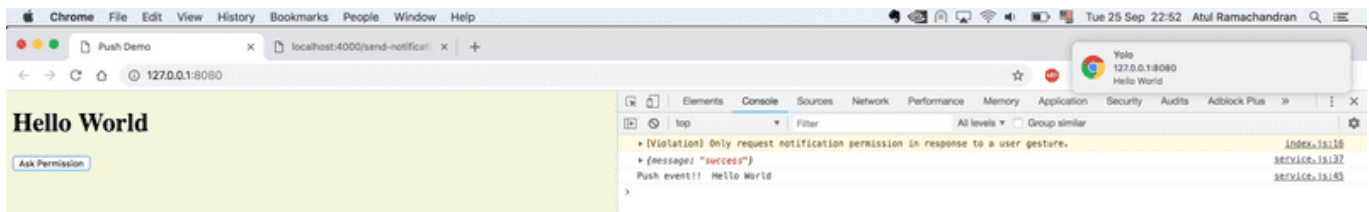
installed for first time.

```
    try {
      const applicationServerKey = urlB64ToUint8Array(
        'BJ5IxJBWdeqFDJTvrZ4wNRu7UY2XigDXjgiUBYEYVXDudxhEs0ReOJRBcBHsPYgZ5dyV
        8VjyqzbQKS8V7bUAglk'
      )
      const options = { applicationServerKey, userVisibleOnly: true }
      const subscription = await
self.registration.pushManager.subscribe(options)
      const response = await saveSubscription(subscription)
      console.log(response)
    } catch (err) {
      console.log('Error', err)
    }
  })

self.addEventListener('push', function(event) {
  if (event.data) {
    console.log('Push event!! ', event.data.text())
    showLocalNotification('Yolo', event.data.text(),
self.registration)
  } else {
    console.log('Push event but no data')
  }
})

const showLocalNotification = (title, body, swRegistration) => {
  const options = {
    body,
    // here you can add more properties like icon, image, vibrate,
    etc.
  }
  swRegistration.showNotification(title, options)
}
```

Now let's run it one last time. Again, unregister everything, clean up cache and reopen the browser tab. Do the same procedure as before.





Wohoo !! 🚀 Now you can create web apps that can spam people too 🗯️

Caveats

- Great article by Rich Harris: *Stuff I wish I'd known sooner about service workers*
<https://gist.github.com/Rich-Harris/fd6c3c73e6e707e312d7c5d7d0f3b2f9>

References

- <https://developers.google.com/web/fundamentals/codelabs/push-notifications/>
- https://developer.mozilla.org/en-US/docs/Web/API/Push_API
- <https://developers.google.com/web/fundamentals/primers/service-workers/>
- <https://blog.sessionstack.com/how-javascript-works-the-mechanics-of-web-push-notifications-290176c5c55d>
- <https://developers.google.com/web/fundamentals/primers/service-workers/>
- <https://developers.google.com/web/fundamentals/codelabs/debugging-service-workers/>
- <https://developers.google.com/web/fundamentals/push-notifications/permission-ux>

- <https://developers.google.com/web/fundamentals/push-notifications/how-push-works>
- <https://gist.github.com/Rich-Harris/fd6c3c73e6e707e312d7c5d7d0f3b2f9>

The entire code for this blog is available at <https://github.com/master-atul/web-push-demo>

This story was originally published at <https://blog.atulr.com/web-notifications> on 25th September 2018

[JavaScript](#) [Push Notification](#) [Service Worker](#)

[About](#) [Help](#) [Legal](#)

Get the Medium app

