AN EASY GUIDE TO GET STARTED WITH PUSH NOTIFICATION IN JAVASCRIPT WITH REACT.JS

React Push notifications (with hooks)

How Push notifications work and how to in React

Lorenzo Spyna Follow
Aug 30, 2019 · 10 min read

This article is a guide to create *Push notifications* with *React* using *Hooks*. By the end of it, you will create a working Push notifications system. In the meantime, you can play with the demo or browse the source code of the final result.

I recently wrote an article about Web Push Notifications in vanilla JavaScript, and this is the React version. I suggest you read that article if you want to know the theory and flows of the *Push Notifications* so that we can jump straight to the code.

This article is divided into two parts:

1. Handling the *Push Notification*
2. Handling the *React app*

Handling Push notifications

Photo by Tim Mossholder on Unsplash

To use Push notifications you have to:

- Check if notifications are supported by the browser
- Register a service worker
- Ask the user permission
- (Optionally) Create a *notification subscription*
- (Optionally) Send the subscription to a *Push Server*
- Display the *Push notification*

It seems a lot of things, let's start with the first one.

Check if Notifications are supported by the browser

You want to use notifications only if the browser supports them. Create a function that does this check.

```
function isPushNotificationSupported() {
  return "serviceWorker" in navigator && "PushManager" in window;
}
```

This function checks if the `PushManager` and the `serviceWorker` do exist. It returns true or false;

We need a `serviceWorker` to handle notifications, so the next thing to do is register one.

Register a service worker

This function registers a file `sw.js` which is supposed to stay in the root of

our project, so in our react-app in the folder `public`. Just for now, it will be an empty file.

```
function registerServiceWorker() {
  return navigator.serviceWorker.register("/sw.js");
}
```

The function returns a promise that resolves a `serviceWorkerRegistration`.

Ask the user permission

Notification can't be sent without the user's consent, let's create a function that asks the permission.

```
async function askUserPermission() {
  return await Notification.requestPermission();
}
```

This function returns the result of the user consent that can be one of: `default`, `denied` and `granted`. `default` means the user didn't give any response.

If the result of this function is `granted` we can move to the next step.

To push or not to push?

At this point, we are at a crossroads! We have to decide if we want to:

- (Just) display notifications
- (Send) Push notifications

Photo by Jon Tyson on Unsplash

Push(ed) notifications are the ones you may have seen on Twitter, Facebook, WhatsApp, and similar apps. When your friends write you a message, you get the notification, even if Facebook or WhatsApp is not open.

If you need this kind of notification, continue reading, otherwise, you can skip the next sections and read directly the part about "sending notifications".

Create the notification subscription

To receive push notifications you need:

- a ***push service***: that manages and receives the notification (the browsers offers a push service)
- a ***push subscription***: that provides a subscription URL endpoint and allows unsubscription from a push service.
- a ***push server***: a server that sends push messages to the push subscription endpoint, which is handled by the browser *push service*.

To create the push subscription we use the service worker registered earlier.

```
async function createNotificationSubscription() {
  //wait for service worker installation to be ready
  const serviceWorker = await navigator.serviceWorker.ready;
  // subscribe and return the subscription
  return await serviceWorker.pushManager.subscribe({
    userVisibleOnly: true,
    applicationServerKey: pushServerPublicKey
  });
}
```

The instruction `navigator.service.worker.ready` waits for the service worker to be ready to be used, because after we register it, could be in other statuses like `waiting`, `installing`. If the registration fails, this promise will never be resolved.

We used the `PushManager` interface to create a subscription and passed two parameters to the method `subscribe`:

- `userVisibleOnly`: A boolean indicating that the returned push subscription will only be used for messages whose effect is made visible to the user.
- `applicationServerKey`: an ECDSA (Elliptic Curve Digital Signature Algorithm) P-256 public key the push server will use to authenticate your application. If specified, all messages from your application server must use the VAPID authentication scheme and include a JWT signed with the corresponding private key. This key ***IS NOT*** the same key that you use to encrypt the data.

I'll show you later how to create a Key pair (private and public) for the push server (the private one) and the application (the public one).

The function returns the `PushSubscription`, an object that contains the unique endpoint for the push server/service, and some other information. Each browser uses different push services that generate different endpoints. In chrome the endpoint will be something like:

```
https://fcm.googleapis.com/fcm/send/fXjr1iIPn00:.....
```

In firefox

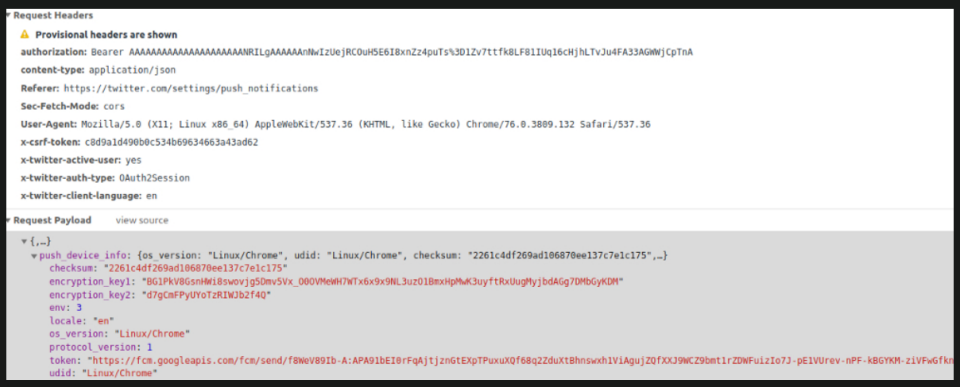
```
https://updates.push.services.mozilla.com/wpush/v2/gAAAAABdZ80ZW5...
```

If we send a push message to that endpoint, we receive a push notification, so don't give it to strangers. The endpoint and the information of the push subscriptions need to be sent to *Push Server* so that it can use it to send messages.

Send the subscription to the Push Server

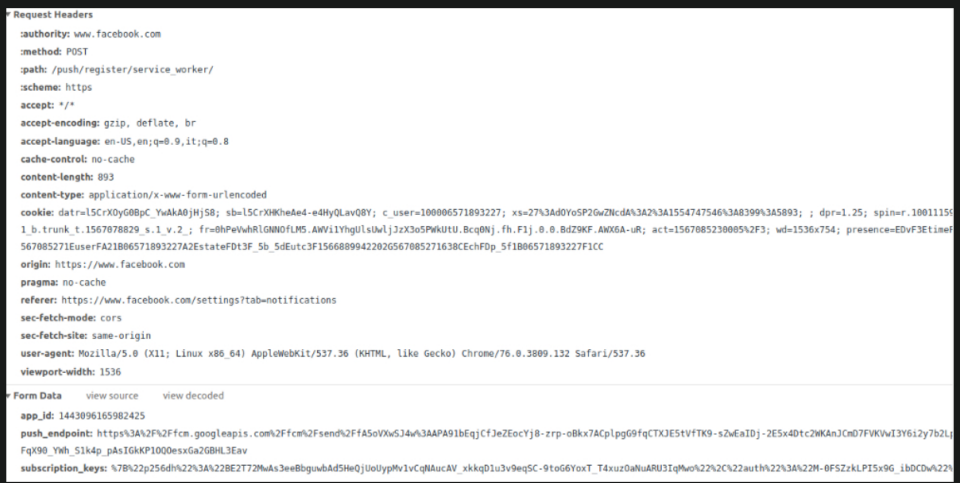
There is no standard way to send the push subscription to the *Push Server*.

Twitter uses its API passing a JSON, like this:



Twitter passes this data to subscribe to the push server. Notice the Authentication header

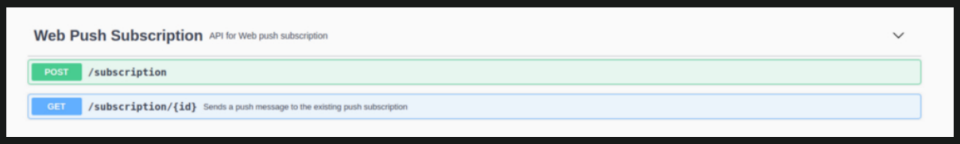
Facebook uses a form:



The network request that Facebook uses to send the subscription to the push server

The thing to notice is that both of them **send the subscription endpoint**, and the submission is secured.

In this tutorial, we will use a demo push server that I build and deployed on the internet. My push server is very dummy, does not have authentication, and does not save your subscription data. The source code is public and you can find it here.



The API of the demo Push server

You can use it at this address, and it exposes two endpoints:

- `POST /subscription` : to receive the subscription.
- `GET /subscription/{id}` : to trigger a push notification for the requested subscription.

The demo push server is made in node with a lib called web-push, I used that lib to create the public key of the subscription. To know how to create one by yourself take a look at that lib.

Let's create a function that calls the Push Server.

```
async function postSubscription(subscription) {
  const response = await fetch(`https://push-notification-demo-server.herokuapp.com/subscription`, {
    credentials: "omit",
    headers: { "content-type": "application/json;charset=UTF-8", "sec-fetch-mode": "cors" },
    body: JSON.stringify(subscription),
    method: "POST",
    mode: "cors"
  });
  return await response.json();
}
```

The parameter `subscription` is exactly the response of the function `createNotificationSubscription` we created earlier.

Once you sent the subscription to the *Push server*, it will send *push messages* to the *Push service* (of the browser).

(Sending the push message)

The Push server exposes the `GET /subscription/{id}` endpoint, which triggers a notification and sends a push message. The server is code is this:

```
...
sendNotification(
      pushSubscription,
      JSON.stringify({
        title: "New Product Available ",
        text: "HEY! Take a look at this brand new t-shirt!",
        image: "/images/jason-leung-HM6TMmevbZQ-unsplash.jpg",
        tag: "new-product",
        url: "/new-product-jason-leung-HM6TMmevbZQ-unsplash.html"
      })
    )
...
```

This code snippet sends a push message to the endpoint of the push subscription, the payload of the message contains the string representation of an object that has: `title`, `text`, `image`, `url`, `tag` and `url`. All this information is inserted in the push message and sent. This means that when the push service (the app) receives the message it can access them.

Receiving the push message

When a *push message* is sent from the *Push server*, the service worker can read it. To make this happen you have to create a listener in the service worker for the push events, let's do this in the service worker file.

```
function receivePushNotification(event) {
  console.log("[Service Worker] Push Received.");

const { image, tag, url, title, text } = event.data.json();

const options = {
    data: url,
    body: text,
    icon: image,
    vibrate: [200, 100, 200],
    tag: tag,
    image: image,
    badge: "/favicon.ico",
    actions: [{ action: "Detail", title: "View", icon:
```

```
"https://via.placeholder.com/128/ff0000" }]
  };
  event.waitUntil(self.registration.showNotification(title,
options));
}
self.addEventListener("push", receivePushNotification);
```

This snippet adds an `eventListener` to the `push` event. The listener is a function that consumes the push events. The method `event.data.json()` converts the message payload in an Object. Next, the method `selft.registration.showNotification` displays the notification to the user device. The options are self explicative, and you can find them documented here.

The user can click the notification. To receive the user click you need to add a listener in the service worker:

```
function openPushNotification(event) {
  console.log("Notification click Received.",
event.notification.data);
   event.notification.close();
  //do something
}
self.addEventListener("notificationclick", openPushNotification);
```

Putting things together

We have learned how to:

- check if the browser supports *Push Notifications*
- register a service worker
- ask user consent to display *Push Notifications*
- create a *Push notification subscription*
- send a *Push notification subscription* to a *Push Server*
- receive a *push message* in the *service worker* and display it
- handle the *notification click* event

Here's the code with all these functions:

This script contains also the function `getUserSubscription` that returns the current Push notification subscription if present, and we will use it in the second part of the article.

And this is the code of the service worker `sw.js`

That's all for the push notification part, let's start the React part.

Create the React app

Second and last step: create a React app.

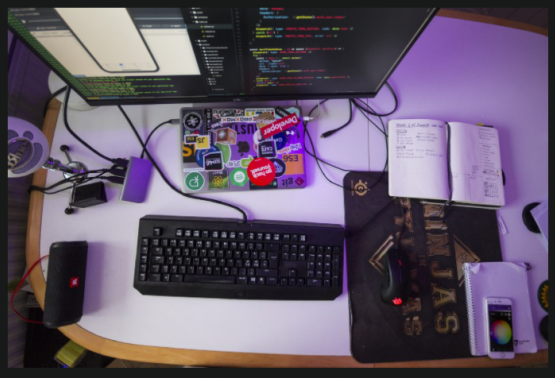


Photo by Oskar Yildiz on Unsplash

The fastest way to create a React app is by using `create-react-app`, running:

```
npm install -g create-react-app
# alternatively use npx

create-react-app push-notifications
# npx create-react-app push-notifications

cd push-notifications
npm run eject
```

Optionally you can run the `eject` script, which allows you to have more control over the app.

Next, we are going to create:

- a custom *React Hook* that uses the functions defined in `push-notification.js`
- a *React* presentation *Component* that uses the custom Hook.

The Hook

Create a file called `usePushNotifications.js` and place it in the source folder.

This is the full commented code:

What it does:

- Checks if the push notifications are supported by the browser
- If the push notifications are supported, registers the service worker
- Retrieves if there is any push notification subscription for the registered service worker
- Defines a click handler `onClickAskUserPermission` to ask the user permission, and exports it, so that you can be used in the component
- Defines and exports a click handler `onClickSubscribeToPushNotification` to create a push notification subscription.
- Defines and exports a click handler `onClickSendSubscriptionToPushServer` that sends the *push subscription* to the *push server.*
- Defines and exports a click handler `onclickSendNotification` that requests the *push server* to send a *push message*.

This Hook is a function that takes no parameters and returns an object containing all the things needed by a Component to use push notifications.

Using destructuring you can use it this way:

```
const {
    userConsent,
    pushNotificationSupported,
    userSubscription,
    onClickAskUserPermission,
    onClickSusbribeToPushNotification,
    onClickSendSubscriptionToPushServer,
    pushServerSubscriptionId,
    onClickSendNotification,
    error,
    loading
  } = usePushNotifications();
```

Which is equivalent to `const usePushNotifications = usePushNotifications()` `const userConsent = usePushNotifications.userConsent`.

React Component

Now that we have all the “business logic” we can create a React Component for the presentation.

Let’s start by importing all the required stuff:

```
import React from "react";
import usePushNotifications from "./usePushNotifications";

export default function PushNotificationDemo() {
  const {
    userConsent,
    pushNotificationSupported,
    userSubscription,
    onClickAskUserPermission,
    onClickSusbribeToPushNotification,
    onClickSendSubscriptionToPushServer,
    pushServerSubscriptionId,
    onClickSendNotification,
    error,
    loading
  } = usePushNotifications();

return <div></div>;
}
```

Next, we want to show an error or a loading message, when things are loading and when there’s an error. Before `return <div></div>` add the following:

```
  if (error) {
    return  (
      <section className="app-error">
        <h2>{error.name}</h2>
        <p>Error message : {error.message}</p>
        <p>Error code : {error.code}</p>
      </section>)
  }

  if (loading) {
    return "Loading, please stand by";
  }
```

Next, we want to show is the consent is granted and if the push notifications are supported, and Button to ask the user consent. Replace `return <div> </div>` with the following:

```
  return (
      <div>
        <p>Push notification are {!pushNotificationSupported && "NOT"}
  supported by your device.</p>
        <p>
          User consent to recevie push notificaitons is <strong>
  {userConsent}</strong>.
        </p>
        <button onClick={onClickAskUserPermission}>Ask user
  permission</button>
      </div>
    );
```

Next, we add the Button to create the notification subscription:

```
  <button onClick={onClickSusbribeToPushNotification}>Create
  Notification subscription</button>
```

Next, is the turn for the Button to send the notification subscription

```
  <button onClick={onClickSendSubscriptionToPushServer}>Send
  subscription to push server</button>
```

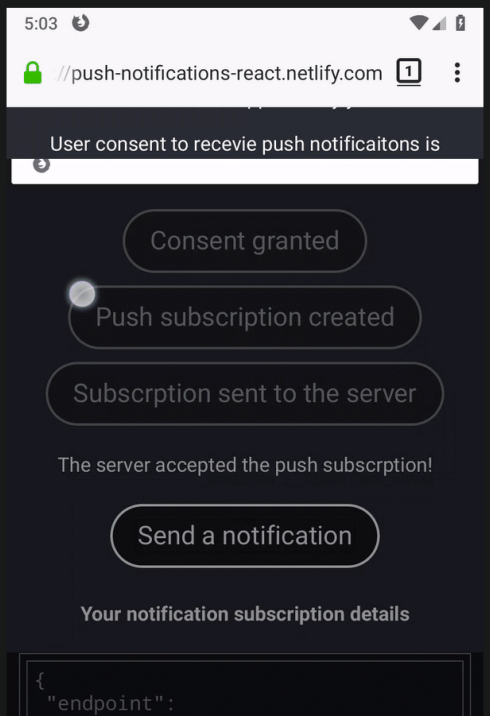
And finally, the Button to send a notification

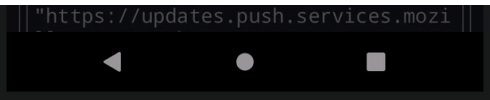
```
  <button onClick={onClickSendNotification}>Send a
  notification</button>
```

This is the full code:

In less than 40 lines of code, you have a clean Component that fully manages Push notifications.

Adding some style and some little improvements I came up with this demo, that you can play with at https://push-notifications-react.netlify.app/. Or if you prefer reading the code, it is available at this repo.





The final result: Push Notifications working

Congratulations!! 😎 You made it to the end. And if you liked 👌this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story.

More stories by Lorenzo Spyna:

An introduction to Web Push Notifications in JavaScript

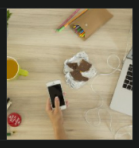
How to implement Web Push Notifications, a beginner introduction to make push notification easy to understand. With...

itnext.io

How to get a job at Amazon

We heard a lot of stories about the hiring process at Amazon, some are true some others are not. I want to share my...

medium.com



Thank you to Eddie Liu for the typo correction.

JavaScript | Development | React | Software Development | Technology

953 claps 5

WRITTEN BY

Lorenzo Spyna

I write Java and JavaScript code — https://spyna.it

Follow

ITNEXT

ITNEXT is a platform for IT developers & software engineers to share knowledge, connect, collaborate, learn and experience next-gen technologies.

Follow

More From Medium

How you can Control your Android Device with Python

Kush in ITNEXT

Your programming language does not matter

Lukasz Raczylo in ITNEXT

Tips for running Kubernetes cluster on Raspberry Pi

Lukasz Raczylo in ITNEXT

Yes, here's the best CSS framework in 2021

@maisonfutari in ITNEXT

Using Matplotlib to Plot a Live Graph of Benford's Law in Python

Kush in ITNEXT

Modularize Xcode Projects using local Swift Packages

Philip Niedertscheider in ITNEXT

Graphing Xcode project dependencies—Introducing XCGrapher

Max Chuquimia in ITNEXT

Simple tips for writing clean React components

Iskander Samatov in ITNEXT

Learn more.

Medium is an open platform where 170 million readers come to find insightful and dynamic thinking. Here, expert and undiscovered voices alike dive into the heart of any topic and bring new ideas to the surface. Learn more

Make Medium yours.

Follow the writers, publications, and topics that matter to you, and you'll see them on your homepage and in your inbox. Explore

Share your thinking.

If you have a story to tell, knowledge to share, or a perspective to offer — welcome home. It's easy and free to post your thinking on any topic. Write on Medium

Medium

About Help Legal