

PWA workshop

Part 4

Web Push Notifications

Web Push Notifications

1. Introduction

Most modern web apps need the ability to update and communicate with their users on a regular basis. Communication channels such as social media, emails, and in-app notifications are great, but they don't always grab the attention of the user, especially when the user navigates away from the website.

This is where push notifications come in. They're those helpful notifications that appear on your device that prompt you about information that could be useful to you. You can swipe or tap away to close them, or you can tap them and be instantly directed to a web page with the relevant information. Traditionally, only native applications had this amazing ability to tap into the operating system of a device and send push notifications. This is where PWAs are a game changer. They have the ability to receive push notifications that appear in the browser.

The best thing about push notifications is that the user receives them even when they aren't browsing your site. The experience looks and feels like a native app and works even if the browser isn't running. This makes it a perfect way to engage with users and draw them back to your web app even if they haven't opened the browser in a while. For example, if your website is a weather application, a push notification could provide your users with useful information such as warnings about approaching bad weather. You could even schedule weekly weather forecasts that can be sent as a push notification, depending on how your users subscribe. The possibilities are endless.

But what about malicious websites using this technology to send spammy push notifications? In order to send push messages to a user, the user first needs to opt-in, to your messages. Once a user has either accepted or blocked the push notification prompt, the prompt won't appear again. It's important to note that this prompt will only appear if the site is running over HTTPS, has a registered Service Worker, and you have written code for it.

What You'll Learn

- ✓ Subscribing to notifications
- ✓ VAPID protocol
- ✓ Sending notifications
- ✓ Receiving and interacting with notifications
- ✓ Unsubscribe from notifications

2. Getting set up

Project Set Up

In this code lab, we are building on top of the project started in the `Keeping your data synchronized with BackgroundSync API` code lab.

If you didn't do it already: **Fork** and then **Clone** the following repository: `https://github.com/The-Guide/fe-guild-2019-pwa.git`

```
$ git clone https://github.com/[YOUR GITHUB PROFILE]/fe-guild-2019-pwa.git
$ cd fe-guild-2019-pwa
```

If you want to start directly with `Web Push Notifications` checkout the following branch:

```
$ git checkout pwa-web-push-init
```

First install the dependencies

```
$ npm install
```

Then type in the terminal

```
$ npm start
```

and open Chrome at `localhost:8080fe-guild-2019-pwa/`

In this code lab, we are also using the server so in case you didn't do it already **fork** and then **clone** the following repository: `https://github.com/The-Guide/fe-guild-2019-pwa-server.git`

```
$ git clone https://github.com/[YOUR GITHUB PROFILE]/fe-guild-2019-pwa-server.git  
$ cd fe-guild-2019-pwa-server
```

Install dependencies

```
$ npm install
```

To start the project type in the terminal:

```
$ npm start
```

the server will be hosted at `localhost:3000`

3. User permissions

The browser displays a prompt asking a user if they'd like to opt-in to notifications. If they accept, you can save their subscription details on the server and use them to send notifications later. These subscription details are unique to each user, device, and browser, so if a user logs in to your site on multiple devices, they'll be prompted once per device.

Once they've accepted, you can use these stored subscription details to send messages to a user later with a scheduled task that updates users with timely information.

Before you can start sending notifications to a user, you need to ask their permission by displaying a prompt. This prompt functionality is built into the browser by default, but first, you need to add a little code to ensure that this prompt is initiated. If a user accepts the prompt, you'll be provided with a subscription object containing information about their subscription. But if a user denies the prompt, you won't be able to send them any messages, and they won't be prompted again. This ensures that you aren't able to annoyingly prompt users every time they visit your site.

User permissions in app.js

```
const enableNotificationsButtons = document.querySelectorAll('.enable-notifications');

const askForNotificationPermission = () => {

  Notification.requestPermission(result=> {

    console.log('User Choice', result);

    if (result !== 'granted') {

      console.log('No notification permission granted!');

    } else {

      console.log('Notification permission granted!');

    }

  });

};

if ('Notification' in window) {

  for (let i = 0; i < enableNotificationsButtons.length; i++) {

    enableNotificationsButtons[i].style.display = 'inline-block';

    enableNotificationsButtons[i].addEventListener('click', askForNotificationPermission);

  }

}
```

Explanation

We ask the user for permission when clicking any of the `Enable Notifications` buttons. The buttons should be visible only if the browser supports notifications.

Also in `app.css`

```
.enable-notifications {  
  display: none;  
}
```


4. Displaying a Notification to the User in App.js

```
const displayConfirmNotification = () => {  
  const options = {  
    body: 'You successfully subscribed to our Notification service!'  
  };  
  new Notification('Successfully subscribed!', options);  
};
```

```
const askForNotificationPermission = () => {  
  Notification.requestPermission(result=> {  
    console.log('User Choice', result);  
    if (result !== 'granted') {  
      console.log('No notification permission granted!');  
    } else {  
      displayConfirmNotification();  
    }  
  });  
};
```

Notifications options: Notifications can be sent from the Service Worker registration and can receive an options object that can control both the visual and behavioral aspect of the notification.

```
const displayConfirmNotification = () => {  
  if ('serviceWorker' in navigator) {  
    const options = {  
      body: 'You successfully subscribed to our Notification service!',  
      icon: 'src/images/icons/app-icon-96x96.png',  
      image: 'src/images/main-image-sm.jpg',  
      dir: 'ltr',  
      lang: 'en-US', // BCP 47,  
      vibrate: [100, 50, 200],  
      badge: 'src/images/icons/app-icon-96x96.png'  
    };  
  
    navigator.serviceWorker.ready  
      .then(sw => sw.showNotification('Successfully subscribed (from SW)!', options));  
  }  
};
```

And, for example in my case on Windows 10, you should be able to see something like this

Push

Test push message from DevTools.

Push

Sync

test-tag-fro

localhost/fe-guild-2019-pwa

Source [sw.js](#)

Received 1/1/19

Status ● #2090 activat


Clients http://localhost:


Push

Test push me

Sync

test-tag-fro





Successfully subscribed (from SW)!

You successfully subscribed to our Notification service!

Google Chrome • localhost:8080

→

ENG

21:52

29/01/2019

2

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>",

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

TODO: INSERT MORE EXPLANATIONS HERE

Let's add additional options like `tag`, `renotify` and actions

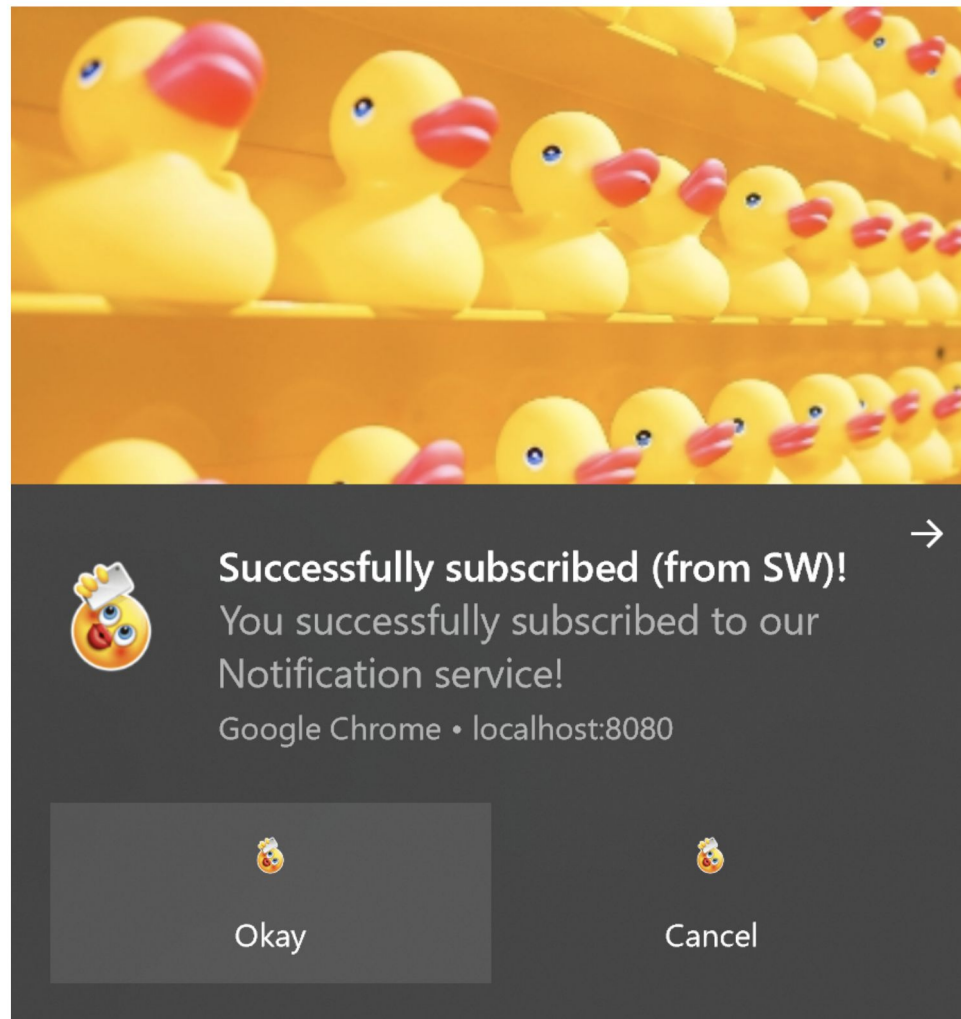
```
const displayConfirmNotification = () => {
  if ('serviceWorker' in navigator) {
    const options = {
      body: 'You successfully subscribed to our Notification service!',
      icon: 'src/images/icons/app-icon-96x96.png',
      image: 'src/images/main-image-sm.jpg',
      dir: 'ltr',
      lang: 'en-US', // BCP 47,
      vibrate: [100, 50, 200],
      badge: 'src/images/icons/app-icon-96x96.png',
      tag: 'confirm-notification',
      renotify: true,
      actions: [
        {
          action: 'confirm',
          title: 'Okay',
          icon: 'src/images/icons/app-icon-96x96.png'
        },
        {
          action: 'cancel',
          title: 'Cancel',
          icon: 'src/images/icons/app-icon-96x96.png'
        }
      ]
    };

    navigator.serviceWorker.ready
      .then(sw => sw.showNotification('Successfully subscribed (from SW)!', options))
  }
};
```

where

- `tag` is the **ID** of the notification
- `renotify` specifies whether the user should be notified after a new notification replaces an old one.

and the result



5. Interacting with notifications

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

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

- Clicking on the notification does nothing.
- Each new notification is shown one after the other. The browser will not collapse the notifications in any way.
- The platform may play a sound or vibrate the user's devices (depending on the platform).
- 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. (For example, compare your notifications on Android and Desktop.)

In this step, 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 or removes 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 whenever a notification is clicked.

In sw-template.js:

```
self.addEventListener('notificationclick', event => {  
  
    const notification = event.notification;  
  
    const action = event.action;  
  
    console.log(notification);  
  
    if (action === 'confirm') {  
  
        console.log('Confirm was chosen');  
  
        notification.close();  
  
    } else {  
  
        console.log(action);  
  
        notification.close();  
  
    }  
  
});  
  
self.addEventListener('notificationclose', event => console.log('Notification was closed', event));
```

6. Subscribing to notifications

Subscribing a user requires two things. First, getting permission (**DONE**) 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 an ID for that user's device.

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

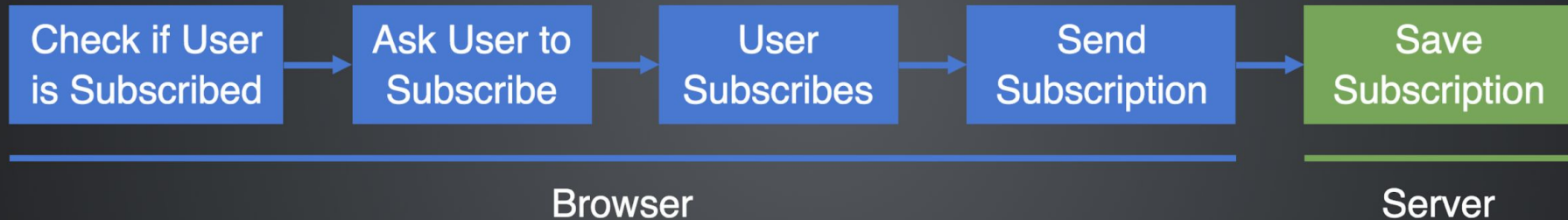
Before subscribing a user, you'll need to generate a set of "application server keys", which we'll cover later on.

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.

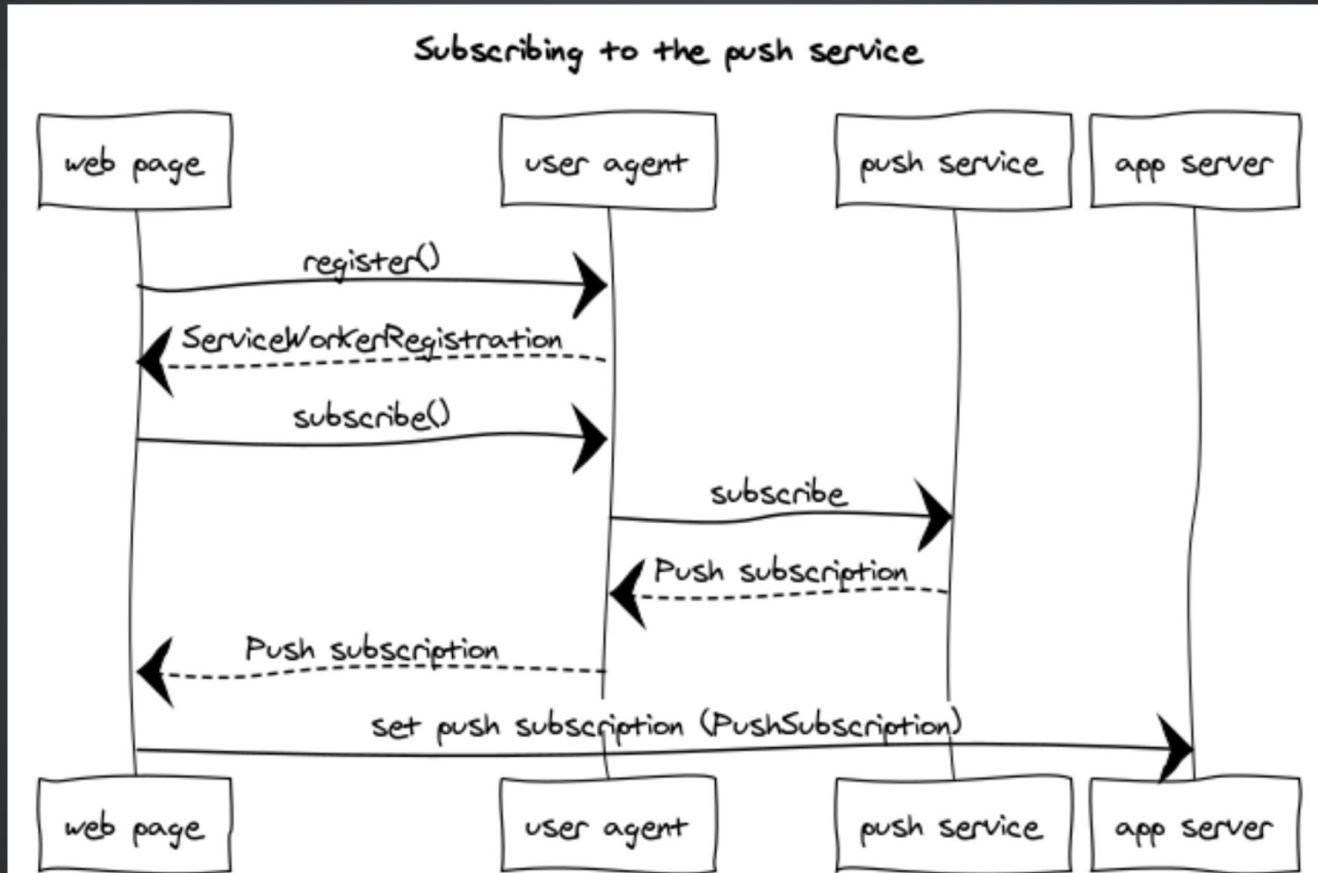
Once we have our service worker registered and we've got permission, we can subscribe a user by calling `registration.pushManager.subscribe()`.

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.

WebPush Notifications



Subscribing to the push service



In app.js just above askForNotificationPermission:

```
const configurePushSubscription = () => {  
  
  if ('serviceWorker' in navigator) {  
  
    let serviceWorkerRegistration;  
  
    navigator.serviceWorker.ready  
  
      .then(registration => {  
  
        serviceWorkerRegistration = registration;  
  
        return registration.pushManager.getSubscription();  
  
      })  
  
      .then(subscription => {  
  
        if (subscription === null) {      // Create a new subscription  
  
          return serviceWorkerRegistration.pushManager.subscribe({  
  
            userVisibleOnly: true,  
  
            applicationServerKey: urlBase64ToUint8Array('PUBLIC KEY HERE')  
  
          });  
  
        }  
  
        .then(pushSubscription => {  
  
          console.log('Received PushSubscription: ', JSON.stringify(pushSubscription));  
  
          return pushSubscription;  
  
        });  
  
      }  
  
    };  
};
```

In utility.js

```
const urlBase64ToUint8Array = base64String => {  
  const padding = '='.repeat((4 - base64String.length % 4) % 4);  
  const base64 = (base64String + padding)  
    .replace(/-/g, '+')  
    .replace(/_/g, '/');  
  
  const rawData = window.atob(base64);  
  const outputArray = new Uint8Array(rawData.length);  
  
  for (let i = 0; i < rawData.length; ++i) {  
    outputArray[i] = rawData.charCodeAt(i);  
  }  
  return outputArray;  
};
```

You may notice code we've used frequently throughout this code lab. You register the Service Worker and, if it's successful, you can then use the registration object.

We first check if you have a subscription by calling `registration.pushManager.getSubscription()` and if it is `null` we then call `subscribe()`

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

Let's look at all the options we can pass in.

userVisibleOnly

At the moment you **must** pass in a value of `true` in order to prevent the sending of a push message without showing 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.

applicationServerKey

We briefly mentioned "application server keys" in the previous section. "Application server keys" are used by a push service to identify the application subscribing a user and ensure 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 application's public key. The browser passes this onto a push service when subscribing the user, meaning the push service can tie your application's public key to the user's `PushSubscription`.

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 application's users.

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.

How to Create Application Server Keys

You can create a public and private set of application server keys from the `fe-guild-2019-pwa-server` root folder by running `npm run web-push`. This uses the [web-push command line](#) to generate keys.

After generating the keys make sure you replace the values inside the `configurePushSubscription` function in `app.js`. Also on the server side inside the `routes.js` file

What is a PushSubscription?

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

```
{
  'endpoint': 'https://SOME.PUSHSERVICE.COM/SOMETHING-UNIQUE',
  'expirationTime': null,
  'keys': {
    'p256dh': 'BGhFV5qx5cd0aD_XF293OqMdYSUIrMrzj2-RuzGwOTIhdW8vFm_zN2VtwMOq9PRlyjaJ3',
    'auth': 'HA1JEiRAp2HLuVH6390umw'
  }
};
```

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

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

Send a Subscription to the 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.

Adapt the `configurePushSubscription` function in `app.js`

```
const configurePushSubscription = () => {  
  
  if ('serviceWorker' in navigator) {  
  
    let serviceWorkerRegistration;  
  
    navigator.serviceWorker.ready  
  
      .then(registration => {  
  
        serviceWorkerRegistration = registration;  
  
        return registration.pushManager.getSubscription();  
  
      })  
  
      .then(subscription => {  
  
        if (subscription === null) {  
  
          // Create a new subscription  
  
          return serviceWorkerRegistration.pushManager.subscribe({  
  
            userVisibleOnly: true,  
  
            applicationServerKey: urlBase64ToUint8Array("PUBLIC KEY HERE")  
  
          });  
  
        }  
  
      });  
  
  }  
  
};
```

next:

```
    })

    .then(pushSubscription => {

      return fetch(`${SERVER_URL}/subscriptions`, {

        method: 'POST',

        headers: {

          'Content-Type': 'application/json',

          'Accept': 'application/json'

        },

        body: JSON.stringify(pushSubscription)

      });

    })

    .then(response => {

      if (response.ok) {

        displayConfirmNotification();

      }

    })

    .catch(error => console.log(error));

  }

}
```

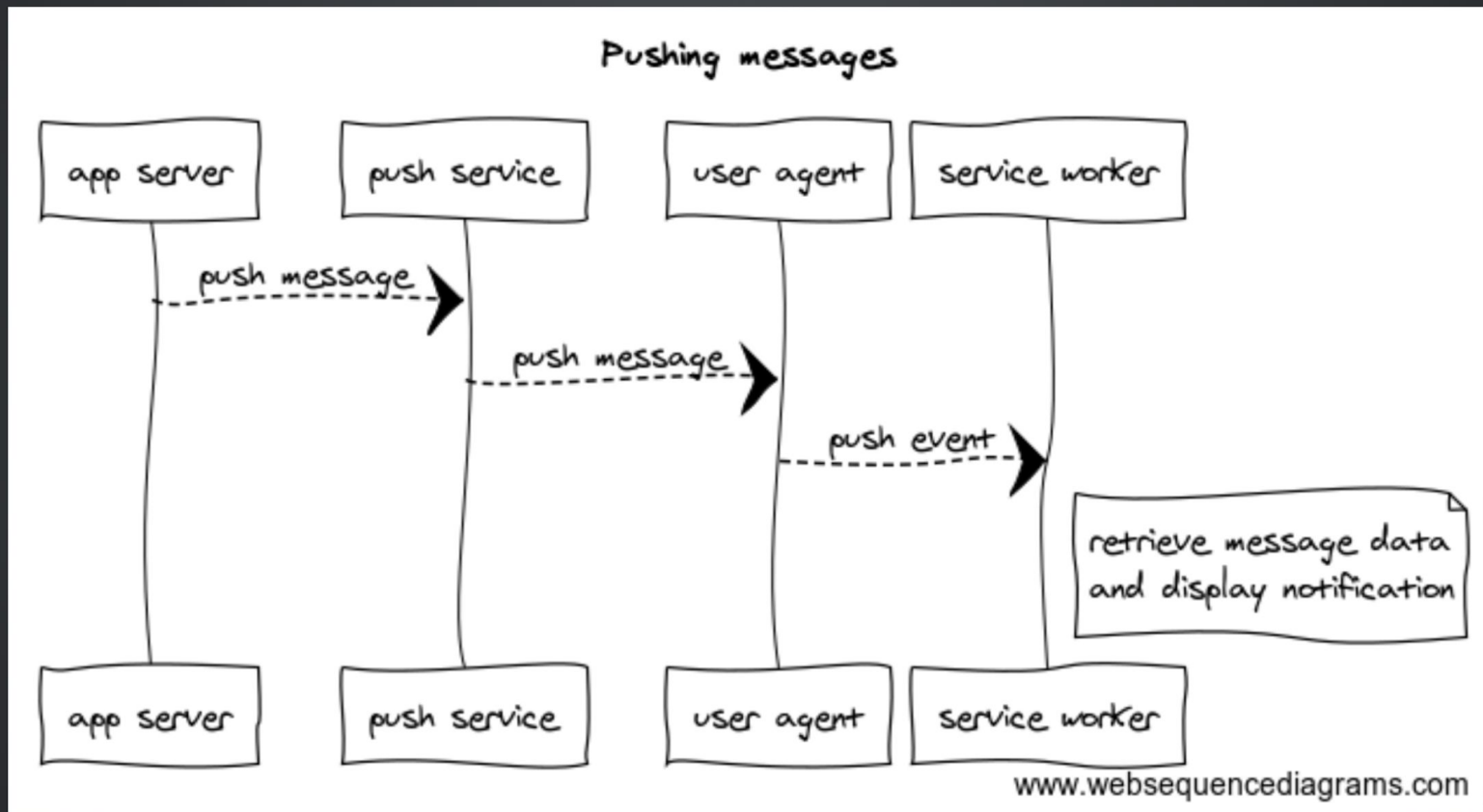
With the `PushSubscription` details on our server, we are good to send our user a message whenever we want. In the `fe-guild-2019-pwa-server\routes\routes.js` the code is already there waiting for us to receive the notifications.

7. Receiving notifications

Now that we've stored the user's unique subscription details, we can start sending push messages to them and provide them with timely updates on important notifications.

On the front-end code, we need to add some code to our Service Worker. The code in the next listing shows how to listen for the push event and display a push notification accordingly.

Pushing messages



In sw-template.js

```
self.addEventListener('push', event => {  
  console.log('Push Notification received', event);  
  let data = {title: 'New!', content: 'Something new happened!', openUrl: '/'};  
  
  if (event.data) {  
    data = JSON.parse(event.data.text());  
  }  
  
  const options = {  
    body: data.content,  
    image: data.imageUrl,  
    icon: 'src/images/icons/app-icon-96x96.png',  
    badge: 'src/images/icons/app-icon-96x96.png',  
    data: {  
      url: data.openUrl  
    }  
  };  
  event.waitUntil(  
    self.registration.showNotification(data.title, options)  
  );  
});
```

The code in the listing above listens to the push event and reads the payload of the data sent from the server. With this payload data, you can then display a notification using the `showNotification` function.

And it looks like this

Hooray! We've sent our first web push notification. You should now notice this appear in the browser. But there's one more step. For the user to interact with the push notification, we need to handle the click event of the notification. We already do that in the `notificationclick`, but the event handler needs a little update.

In sw-template.js replace the notificationclick event handler:

```
self.addEventListener('notificationclick', event => {
  const notification = event.notification;
  const action = event.action;

  console.log(notification);

  if (action === 'confirm') {
    console.log('Confirm was chosen');
    notification.close();
  } else {
    event.waitUntil(
      self.clients.matchAll()
        .then(clients => {
          let visibleClient = clients.find(client => client.visibilityState === 'visible');

          if (visibleClient && 'navigate' in visibleClient) {
            visibleClient.navigate(notification.data.url);
            visibleClient.focus();
          }
        })
    );
  }
});
```


Next:

```
else {  
    self.clients.openWindow(`fe-guild-2019-pwa/${notification.data.url}`);  
}  
notification.close();  
})  
);  
console.log(action);  
notification.close();  
}  
});
```

8. Unsubscribe from notifications

Users can unsubscribe themselves by changing a few settings in their browser, but there may come a time when you want to unsubscribe a user programmatically. For example, you could add a simple button to your web page that would allow users to unsubscribe at the tap of a button instead of digging around in their browser settings. The code in the following listing shows this in action.

```
const unsubscribe = () => {  
  if ('serviceWorker' in navigator) {  
    navigator.serviceWorker.ready  
      .then(serviceWorkerRegistration => {  
        return serviceWorkerRegistration.pushManager.getSubscription();  
      })  
      .then(subscription => {  
        if (!subscription) {  
          console.log("Not subscribed, nothing to do.");  
          return;  
        }  
        return subscription.unsubscribe();  
      })  
      .then(() => console.log("Successfully unsubscribed!"))  
      .catch(error => console.error('Error thrown while trying to unsubscribe from push messaging', error));  
  }  
};
```

The code in the listing above is a basic example that shows how you can unsubscribe a user. The listing contains code that will first check to see if the user is already subscribed using the `pushManager.getSubscription()` function. If the user is subscribed, you then unsubscribe them using the `subscription.unsubscribe()` function.

Exercises

1. Implement the unsubscribe functionality.
2. Delete the subscription from the server as well.

9. Third-party push notifications

As you can imagine, the business of sending push notifications to the many different browsers out there can be tricky. If you'd prefer not to build your own push notification server and instead use a SaaS product, there are many third-party solutions out there.

Services such as OneSignal, Roost, and Aimtell all offer a solution that can target multiple browsers and provide you with enhanced functionality. Many of these services have created libraries to deal with all the major browsers, which means you'll get full coverage and engagement regardless of browser. These services also have a lot of functionality built into them that allows you to schedule messages for a later date, and some have complex reporting charts that give insight into how your users are interacting with your notifications.

Solution

Source code

```
$ git checkout pwa-web-push-final
```