Subscribing a User

- Feature detection
- Register a service worker
  - PushManager API browser compatibility
- Requesting permission
- Subscribe a user with PushManager
  - userVisibleOnly options
  - applicationServerKey option
  - How to create application server keys
  - Permissions and subscribe()
  - What is a PushSubscription?
- Send a subscription to Your Server
  - FAQs
    - Can I change the push service a browser uses?
    - Each browser uses a different Push Service, don't they have different API's?
    - If I subscribe a user on their desktop, are they subscribed on their phone as well?
- Where to go next
  - Code labs

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

1. Check for *serviceWorker* on *navigator*.
2. Check for *PushManager* on *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, 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 are 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 function tells the browser that we have a service worker file and where it's 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 runs correctly and there are 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 the PushManager API.

PushManager API browser compatibility

{% BrowserCompat 'api.PushManager' %}

Requesting permission

We've registered our service worker and are 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:

{% Img src="image/C47gYyWYVMMhDmtYSLOWazuyePF2/VpF84naCF2Ri7mF3MlhW.png", alt="Permission prompt on desktop and mobile Chrome.", width="800", height="251" %}

Once the user interacted with the permission prompt by pressing Allow, Block, or just closing it, 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. They'll have to manually "unblock" your app by changing its permission state, 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 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 navigator.serviceWorker
    .register('/service-worker.js')
    .then(function (registration) {
      const subscribeOptions = {
        userVisibleOnly: true,
        applicationServerKey: urlBase64ToUint8Array(
```

```
          'BEl62iUYgUivxIkv69yViEuiBIa-Ib9-
SkvMeAtA3LFgDzkrxZJjSgSnfckjBJuBkr3qBUYIHBQFLXYp5Nksh8U',
        ),
      };

      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.

applicationServerKey option

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`.

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.

{% Img src="image/C47gYyWYVMMhDmtYSLOWazuyePF2/nHEwbmGFjtttom6DTFAw.svg", alt="Illustration of the public application server key is used in subscribe method.", width="800", height="264" %}

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.

{% Img src="image/C47gYyWYVMMhDmtYSLOWazuyePF2/A8NWXbSlEadr12WXrer6.svg", alt="How the private application server key is used when sending a message", width="800", height="378" %}

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.

How to create application server keys

You can create a public and private set of application server keys by visiting web-push-codelab.glitch.me or you can use the web-push command line 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, and in return we get a promise that resolves to a `PushSubscription` resulting in some code like so:

```
function subscribeUserToPush() {
  return navigator.serviceWorker
    .register('/service-worker.js')
    .then(function (registration) {
      const subscribeOptions = {
        userVisibleOnly: true,
        applicationServerKey: urlBase64ToUint8Array(
          'BEl62iUYgUivxIkv69yViEuiBIa-Ib9-
SkvMeAtA3LFgDzkrxZJjSgSnfckjBJuBkr3qBUYIHBQFLXYp5Nksh8U',
        ),
      };
```

```
      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":

"BIPUL12DLfytvTajnryr2PRdAgXS3HGKiLqndGcJGabyhHheJYlNGCeXl1dn18gSJ1WAkAPIxr4gK0_dQds4yiI=",
        "auth":"FPssNDTKnInHVndSTdbKFw=="
      }
    }
```

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 section).

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. Alternatively you can piece together the same result manually like so:

```
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);
```

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 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.

Where to go next

- Web Push Notification Overview
- How Push Works
- Subscribing a User
- Permission UX
- Sending Messages with Web Push Libraries
- Web Push Protocol
- Handling Push Events
- Displaying a Notification
- Notification Behavior
- Common Notification Patterns
- Push Notifications FAQ
- Common Issues and Reporting Bugs

Code labs

- Build a push notification client
- Build a push notification server

□□□ #2
□ □□□ □□□ 2026-10-31 22:12:02 UTC
□ □□□ □□□ 2026-10-31 22:12:04 UTC