Common Notification Patterns

By Matt Gaunt
Matt is a contributor to Web**Fundamentals**

We're going to look at some common implementation patterns for web push.

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

Notification close event

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

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

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

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

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

Adding data to a notification

When a push message is received it's common to have data that is only useful if the user has clicked the notification. For example, the URL that should be opened when a notification is clicked.

The easiest way to take data from a push event and attach it to a notification is to add a **`data`** parameter to the options object passed into **`showNotification()`**, like so:

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

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

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

Open a window

One of the most common responses to a notification is to open a window / tab to a specific URL. We can do this with the clients.openWindow() API.

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

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

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

Focus an existing window

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

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

Taking the previous example, we'll alter the code to see if '/demos/notification-examples /example-page.html' is already open.

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

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

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

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

event.waitUntil(promiseChain);
```

Let's step through the code.

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

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

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

Then we get a list of the `WindowClient` objects, which are the list of currently open tabs and windows. (Remember these are tabs for your origin only.)

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

The options passed into `matchAll` inform the browser that we only want to search for "window" type clients (i.e. just look for tabs and windows and exclude web workers). `includeUncontrolled` allows us to search for all tabs from your origin that are not controlled by the current service worker, i.e. the service worker running this code. Generally, you'll always want `includeUncontrolled` to be true when calling `matchAll()`.

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

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

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

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

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

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

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

Merging notifications

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

You can however get more sophisticated with the collapsing of notifications using the Notifications API. Consider a chat app, where the developer might want a new notification to show a message similar to "You have two messages from Matt" rather than just showing the latest message.

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

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

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

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

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

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

  return currentNotification;
})
```

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

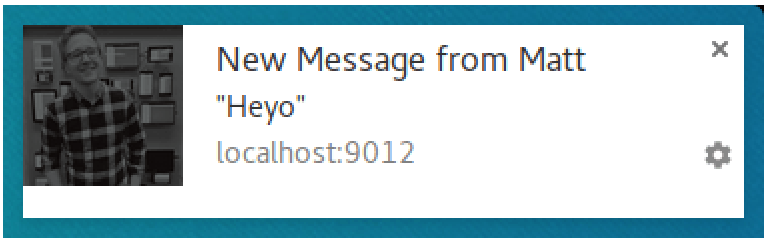
In this fake message app, we'll track the number of new messages by adding a count to our new notifications data and increment it with each new notification.

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

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

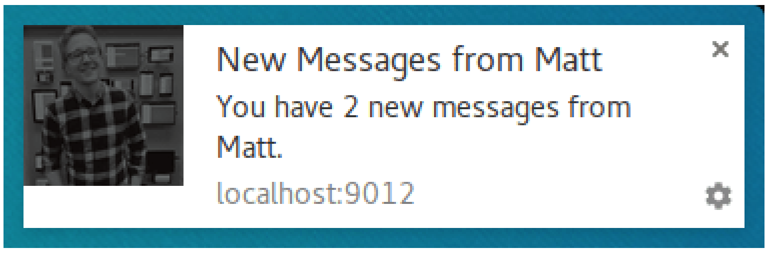
```
        options.body = `You have ${messageCount} new messages from ${userName}.`;
        options.data = {
          userName: userName,
          newMessageCount: messageCount
        };
        notificationTitle = `New Messages from ${userName}`;

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

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

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

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


New Message from Matt
"Heyo"
localhost:9012

A second notification would collapse the notifications into this:


New Messages from Matt
You have 2 new messages from Matt.
localhost:9012

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

The exception to the rule

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

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

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

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

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

    return clientIsFocused;
  });
}
```

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

Inside our push event we'd use this function to decide if we need to show a notification:

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

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

  event.waitUntil(promiseChain);
```

Message a page from a push event

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

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

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

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

event.waitUntil(promiseChain);
```

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

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

In this message listener you could do anything you want, show a custom UI on your page or completely ignore the message.

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

Cache a page and open a window

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

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

For more information check out this introduction to service workers post.

Previous
← Notification Behavior

Next
FAQ →