Sending Messages with Web Push Libraries

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

One of the pain points when working with web push is that triggering a push message is extremely "fiddly". To trigger a push message an application needs to make a POST request to a push service following the web push protocol. To use push across all browsers you need to use VAPID (a.k.a application server keys) which basically requires setting a header with a value proving your application can message a user. To send data with a push message, the data needs to be encrypted and specific headers added so the browser can decrypt the message correctly.

The main issue with triggering push is that if you hit a problem, it's difficult to diagnose the issue. This is improving with time and wider browser support but it's far from easy. For this reason, I strongly recommend using a library to handle the encryption, formatting and triggering of your push message.

If you really want to learn about what the libraries are doing, we'll cover it in the next section. For now, we are going to look at managing subscriptions and using an existing web push library to make the push requests.

In this section we'll be using the web-push Node library. Other languages will have differences, but they won't be too dissimilar. We are looking at Node since it's JavaScript and should be the most accessible for readers.

Note: If you want a library for a different language, checkout the web-push-libs organization on Github.

We'll go through the following steps:

1. Send a subscription to our backend and save it.
2. Retrieve saved subscriptions and trigger a push message.

Saving Subscriptions

Saving and querying `PushSubscriptions` from a database will vary depending on your server side language and database choice, but it might be useful to see an example of how it could be done.

In the demo web page the `PushSubscription` is sent to our backend by making a simple POST request:

```
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 Express server in our demo has a matching request listener for the `/api/save-subscription/` endpoint:

```
app.post('/api/save-subscription/', function (req, res) {
```

In this route we validate the subscription just to make sure the request is OK and not full of garbage:

```
const isValidSaveRequest = (req, res) => {
  // Check the request body has at least an endpoint.
  if (!req.body || !req.body.endpoint) {
    // Not a valid subscription.
    res.status(400);
    res.setHeader('Content-Type', 'application/json');
    res.send(JSON.stringify({
      error: {
        id: 'no-endpoint',
        message: 'Subscription must have an endpoint.'
      }
    }));
    return false;
  }
  return true;
};
```

Note: In this route we only check for an endpoint. If you **require** payload support, make sure you check for the auth and p256dh keys as well.

If the subscription is valid, we need to save it and return an appropriate JSON response:

```
  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
      }
    }));
  });
```

This demo uses nedb to store the subscriptions, it's a simple file based database, but you could use any database you chose. We are only using this as it requires zero set-up. For production you'd want to use something more reliable. (I tend to stick with good old MySQL.)

```
function saveSubscriptionToDatabase(subscription) {
  return new Promise(function(resolve, reject) {
    db.insert(subscription, function(err, newDoc) {
      if (err) {
        reject(err);
        return;
      }

      resolve(newDoc._id);
    });
  });
};
```

Sending Push Messages

When it comes to sending a push message we ultimately need some event to trigger the process of sending a message to users. A common approach is creating an admin page that let's you configure and trigger the push message. But you could create a program to run locally or any other approach that allows accessing the list of `PushSubscriptions` and running the code to trigger the push message.

Our demo has an "admin like" page that lets you trigger a push. Since it's just a demo it's a public page.

I'm going to go through each step involved in getting the demo working, these will be baby steps to everyone follow along, including anyone who is new to Node.

When we discussed subscribing a user we covered adding an `applicationServerKey` to the `subscribe()` options. It's on the back end that we'll need this private key.

In the demo these values are added to our Node app like so (boring code I know, but just want you to know there is no magic):

```
const vapidKeys = {
  publicKey:
'BEl62iUYgUivxIkv69yViEuiBIa-Ib9-SkvMeAtA3LFgDzkrxZJjSgSnfckjBJuBkr3qBUYIHBQFLXYp
  privateKey: 'UUxI4O8-FbRouAevSmBQ6o18hgE4nSG3qwvJTfKc-ls'
};
```

Next we need to install the `web-push` module for our Node server:

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

Then in our Node script we require in the `web-push` module like so:

```
const webpush = require('web-push');
```

Now we can start to use the `web-push` module. First we need to tell the `web-push` module about our application server keys. (Remember they are also known as VAPID keys because that's the name of the spec.)

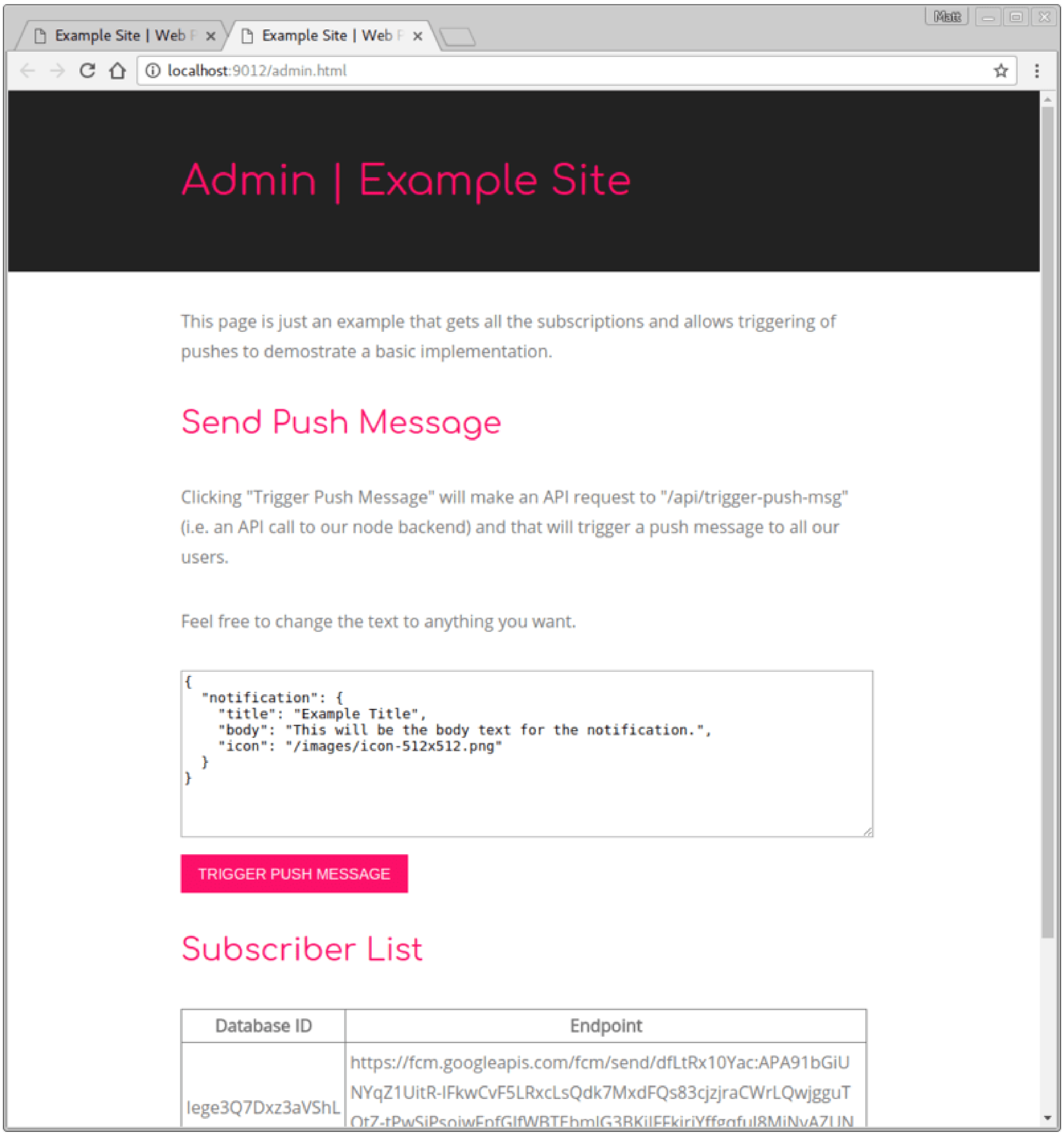
```
const vapidKeys = {
  publicKey:
'BEl62iUYgUivxIkv69yViEuiBIa-Ib9-SkvMeAtA3LFgDzkrxZJjSgSnfckjBJuBkr3qBUYIHBQFLXYp
  privateKey: 'UUxI4O8-FbRouAevSmBQ6o18hgE4nSG3qwvJTfKc-ls'
};
```

```
webpush.setVapidDetails(
  'mailto:web-push-book@gauntface.com',
  vapidKeys.publicKey,
  vapidKeys.privateKey
);
```

We also include a "mailto:" string as well. This string needs to be either a URL or a mailto email address. This piece of information will actually be sent to web push service as part of the request to trigger a push. The reason this is done is so that if a web push service needs to get in touch with the sender, they have some information that will enable them to.

With this, the `web-push` module is ready to use, the next step is to trigger a push message.

The demo uses the pretend admin panel to trigger push messages.


Example Site | Web F
Example Site | Web F
localhost:9012/admin.html

Admin | Example Site

This page is just an example that gets all the subscriptions and allows triggering of pushes to demostrate a basic implementation.

Send Push Message

Clicking "Trigger Push Message" will make an API request to "/api/trigger-push-msg" (i.e. an API call to our node backend) and that will trigger a push message to all our users.

Feel free to change the text to anything you want.

```
{
  "notification": {
    "title": "Example Title",
    "body": "This will be the body text for the notification.",
    "icon": "/images/icon-512x512.png"
  }
}
```

Subscriber List

Database ID	Endpoint
lege3Q7Dxz3aVShL	https://fcm.googleapis.com/fcm/send/dfLtRx10Yac:APA91bGiU NYqZ1UitR-IFkwCvF5LRxcLsQdk7MxdFQs83cjzjraCWrLQwjgguT Ot7-tPwSiPsoiwEnfGIfWBTEbmIG3BKiIFEkiriYffgqfuI8MiNvAZUN

Clicking the "Trigger Push Message" button will make a POST request to `/api/trigger-push-msg/` which is the signal for our backend to send push messages, so we create the route in express for this endpoint:

```
app.post('/api/trigger-push-msg/', function (req, res) {
```

When this request is received, we grab the subscriptions from the database and for each one, we trigger a push message.

```
  return getSubscriptionsFromDatabase()
  .then(function(subscriptions) {
    let promiseChain = Promise.resolve();

    for (let i = 0; i < subscriptions.length; i++) {
      const subscription = subscriptions[i];
      promiseChain = promiseChain.then(() => {
        return triggerPushMsg(subscription, dataToSend);
      });
    }

    return promiseChain;
  })
```

The function **`triggerPushMsg()`** can then use the web-push library to send a message to the provided subscription.

```
const triggerPushMsg = function(subscription, dataToSend) {
  return webpush.sendNotification(subscription, dataToSend)
  .catch((err) => {
    if (err.statusCode === 410) {
      return deleteSubscriptionFromDatabase(subscription._id);
    } else {
      console.log('Subscription is no longer valid: ', err);
    }
  });
};
```

The call to **`webpush.sendNotification()`** will return a promise. If the message was sent successfully the promise will resolve and there is nothing we need to do. If the promise rejects, you need to examine the error as it'll inform you as to whether the PushSubscription is still valid or not.

To determine the type of error from a push service it's best to look at the status code. Error messages vary between push services and some are more helpful than others.

In this example it checks for status codes '404' and '410', which are the HTTP status codes for 'Not Found' and 'Gone'. If we receive one of these, it means the subscription has expired or is no longer valid. In these scenarios we need remove the subscriptions from our database.

We'll cover some of the other status codes in the next section when we look at the web push protocol in more detail.

Note: If you hit problems at this stage, it's worth looking at the error logs from Firefox before Chrome. The Mozilla push service has much more helpful error messages compared to Chrome / FCM.

After looping through the subscriptions, we need to return a JSON response.

```
.then(() => {
  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-send-messages',
      message: `We were unable to send messages to all subscriptions : ` +
        `'${err.message}'`
    }
  }));
});
```

We've gone over the major implementation steps.

1. Create an API to send subscriptions from our web page to our back-end so it can save them to a database.
2. Create an API to trigger the sending of push messages (in this case an API called from the pretend admin panel).
3. Retrieve all the subscriptions from our backend and send a message to each subscription with one of the web-push libraries.

Regardless of your backend (Node, PHP, Python, ...) the steps for implementing push are going to be the same.

Next up, what exactly are these web-push libraries doing for us?

Previous
← Permission UX

Next
Web Push Protocol →