Web Push Interoperability Wins

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

By Joseph Medley
Technical Writer

When Chrome first supported the Web Push API, it relied on the Firebase Cloud Messaging (FCM),formerly known as Google Cloud Messaging (GCM), push service. This required using it's proprietary API. This allowed the Chrome to make the Web Push API available to developers at a time when the Web Push Protocol spec was still being written and later provided authentication (meaning the message sender is who they say they are) at a time when the Web Push Protocol lacked it. Good news: neither of these are true anymore.

FCM / GCM and Chrome now support the standard Web Push Protocol, while sender authentication can be achieved by implementing VAPID, meaning your web app no longer needs a 'gcm_sender_id'.

In this article, I'm going to first describe how to convert your existing server code to use the Web Push Protocol with FCM. Next, I'll show you how to implement VAPID in both your client and server code.

FCM supports Web Push Protocol

Let's start with a little context. When your web application registers for a push subscription it's given the URL of a push service. Your server will use this endpoint to send data to your user via your web app. In Chrome you'll be given a FCM endpoint if you subscribe a user without VAPID. (We'll cover VAPID later). Before FCM supported Web Push Protocol you had to extract the FCM registration ID from the end of the URL and and put it in the header before making a FCM API request. For example, an FCM endpoint of `https://android.googleapis.com/gcm/send/ABCD1234`, would have a registration ID of 'ABCD1234'.

Now that FCM supports Web Push Protocol you can leave the endpoint intact and use the URL as a Web Push Protocol endpoint. (This brings it in line with Firefox and hopefully every other future browser.)

Before we dive into VAPID, we need to make sure our server code correctly handles the FCM endpoint. Below is an example of making a request to a push service in Node. Notice that for FCM we're adding the API key to the request headers. For other push service endpoints this won't be needed. For Chrome prior to version 52, Opera Android and the Samsung

Browser, you're also still required to include a 'gcm_sender_id' in your web app's manifest.json. The API key and sender ID are used to check whether the server making the requests is actually allowed to send messages to the receiving user.

```
const headers = new Headers();
// 12-hour notification time to live.
headers.append('TTL', 12 * 60 * 60);
// Assuming no data is going to be sent
headers.append('Content-Length', 0);

// Assuming you're not using VAPID (read on), this
// proprietary header is needed
if(subscription.endpoint
  .indexOf('https://android.googleapis.com/gcm/send/') === 0) {
  headers.append('Authorization', 'GCM_API_KEY');
}

fetch(subscription.endpoint, {
  method: 'POST',
  headers: headers
})
.then(response => {
  if (response.status !== 201) {
    throw new Error('Unable to send push message');
  }
});
```

Remember, this is a change to FCM / GCM's API, so you don't need to update your subscriptions, just change your server code to define the headers as shown above.

Introducing VAPID for server identification

VAPID is the cool new short name for "Voluntary Application Server Identification". This new spec essentially defines a handshake between your app server and the push service and allows the push service to confirm which site is sending messages. With VAPID you can avoid the FCM-specific steps for sending a push message. You no longer need a Firebase project, a `gcm_sender_id`, or an `Authorization` header.

The process is pretty simple:

1. Your application server creates a public/private key pair. The public key is given to your web app.

2. When the user elects to receive pushes, add the public key to the subscribe() call's options object.
3. When your app server sends a push message, include a signed JSON Web Token along with the public key.

Let's look at these steps in detail.

Create a public/private key pair

I'm terrible at encryption, so here's the relevant section from the spec regarding the format of the VAPID public/private keys:

> Application servers SHOULD generate and maintain a signing key pair usable with elliptic curve digital signature (ECDSA) over the P-256 curve.

You can see how to do this in the web-push node library:

```
function generateVAPIDKeys() {
  var curve = crypto.createECDH('prime256v1');
  curve.generateKeys();

  return {
    publicKey: curve.getPublicKey(),
    privateKey: curve.getPrivateKey(),
  };
}
```

Subscribing with the public key

To subscribe a Chrome user for push with the VAPID public key, you need to pass the public key as a Uint8Array using the **`applicationServerKey`** parameter of the subscribe() method.

```
const publicKey = new Uint8Array([0x4, 0x37, 0x77, 0xfe, …. ]);
serviceWorkerRegistration.pushManager.subscribe(
  {
    userVisibleOnly: true,
    applicationServerKey: publicKey
  }
);
```

You'll know if it has worked by examining the endpoint in the resulting subscription object, if the origin is **`fcm.googleapis.com`**, it's working.

```
https://fcm.googleapis.com/fcm/send/ABCD1234
```

Note: Even though this is an FCM URL, use the Web Push Protocol **not** the FCM protocol, this way your server side code will work for any push service.

Sending a push message

To send a message using VAPID, you need to make a normal Web Push Protocol request with two additional HTTP headers: an Authorization header and a Crypto-Key header.

Authorization header

The `Authorization` header is a signed JSON Web Token (JWT) with 'WebPush ' in front of it.

A JWT is a way of sharing a JSON object with a second party in such a way that the sending party can sign it and the receiving party can verify the signature is from the expected sender. The structure of a JWT is three encrypted strings, joined with a single dot between them.

```
<JWTHeader>.<Payload>.<Signature>
```

JWT header

The JWT Header contains the algorithm name used for signing and the type of token. For VAPID this must be:

```
{
  "typ": "JWT",
  "alg": "ES256"
}
```

This is then base64 url encoded and forms the first part of the JWT.

Payload

The Payload is another JSON object containing the following:

- Audience ("aud")

    - This is the origin of the push service (**NOT** the origin of your site). In JavaScript, you could do the following to get the audience: `const audience = new URL(subscription.endpoint).origin`
- Expiration Time ("exp")
    - This is the number of seconds until the request should be regarded as expired. This **MUST** be within 24 hours of the request being made, in UTC.
- Subject ("sub")
    - The subject needs to be a URL or a `mailto:` URL. This provides a point of contact in case the push service needs to contact the message sender.

An example payload could look like the following:

```
{
    "aud": "http://push-service.example.com",
    "exp": Math.floor((Date.now() / 1000) + (12 * 60 * 60)),
    "sub": "mailto: my-email@some-url.com"
}
```

This JSON object is base64 url encoded and forms the second part of the JWT.

Signature

The Signature is the result of joining the encoded header and payload with a dot then encrypting the result using the VAPID private key you created earlier. The result itself should be appended to the header with a dot.

I'm not going to show a code sample for this as there are a number of libraries that will take the header and payload JSON objects and generate this signature for you.

The signed JWT is used as the Authorization header with 'WebPush ' prepended to it and will look something like the following:

```
WebPush eyJ0eXAiOiJKV1QiLCJhbGciOiJFUzI1NiJ9.eyJhdWQiOiJodHRwczovL2ZjbS5nb2
```

Notice a few things about this. First, the Authorization header literally contains the word 'WebPush' and should be followed by a space then the JWT. Also notice the dots separating the JWT header, payload, and signature.

Crypto-Key header

As well as the Authorization header, you must add your VAPID public key to the `Crypto-Key` header as a base64 url encoded string with `p256ecdsa=` prepended to it.

```
p256ecdsa=BDd3_hVL9fZi9Ybo2UUzA284WG5FZR30_95YeZJsiApwXKpNcF1rRPF3foIiBHXRd
```

When you are sending a notification with encrypted data, you will already be using the `Crypto-Key` header, so to add the application server key, you just need to add a semicolon before adding the above content, resulting in:

```
dh=BGEw2wsHgLwzerjvnMTkbKrFRxdmwJ5S_k7zi7A1coR_sVjHmGrlvzYpAT1n4NPbioFlQkIrTNL8EH
p256ecdsa=BDd3_hVL9fZi9Ybo2UUzA284WG5FZR30_95YeZJsiApwXKpNcF1rRPF3foIiBHXRdJI2Qhu
```

Note: The separating semicolon should actually be a comma but there is a bug in Chrome prior to version 52 which prevents push from working if a comma is sent. This is fixed in Chrome 53, so you should be able to change to a comma once this hits stable.

Reality of these changes

With VAPID you no longer need to sign up for an account with GCM to use push in Chrome and you can use the same code path for subscribing a user and sending a message to a user in both Chrome and Firefox. Both are following the standards.

What you need to bear in mind is that in Chrome 51 and before, Opera for Android and Samsung browser you'll still need to define the `gcm_sender_id` in your web app manifest and you'll need to add the Authorization header to the FCM endpoint that will be returned.

VAPID provides an off ramp from these proprietary requirements. If you implement VAPID it'll work in all browsers that support web push. As more browsers support VAPID you can decide when to drop the `gcm_sender_id` from your manifest.

Note: Be sure to check out the full documentation including best practices for using Web Push Notifications