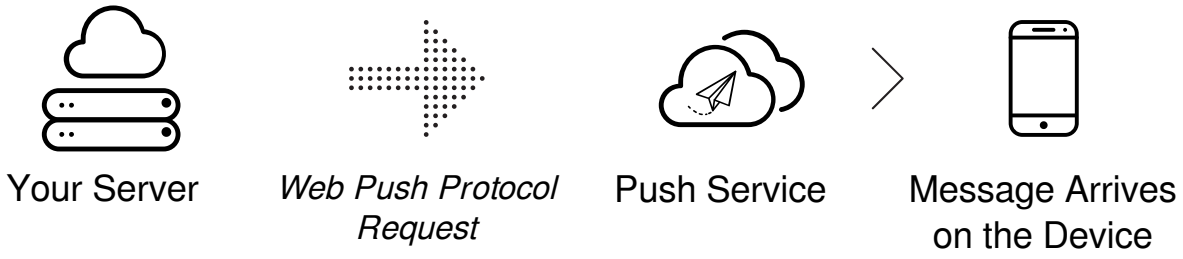
The Web Push Protocol

By Matt Gaunt

Matt is a contributor to Web**Fundamentals**

We've seen how a library can be used to trigger push messages, but what exactly are these libraries doing?

Well, they're making network requests while ensuring such requests are the right format. The spec that defines this network request is the Web Push Protocol.



This section outlines how the server can identify itself with application server keys and how the encrypted payload and associated data is sent.

This isn't a pretty side of web push and I'm no expert at encryption, but let's look through each piece since it's handy to know what these libraries are doing under the hood.

Application server keys

When we subscribe a user, we pass in an `applicationServerKey`. This key is passed to the push service and used to check that the application that subscribed the user is also the application that is triggering push messages.

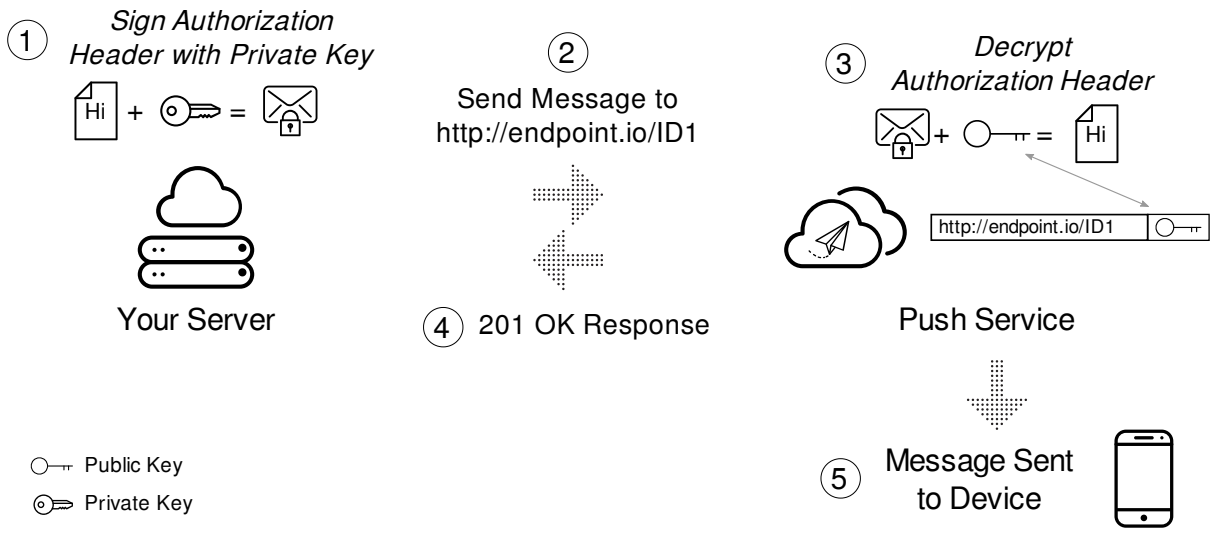
When we trigger a push message, there are a set of headers that we send that allow the push service to authenticate the application. (This is defined by the VAPID spec.)

What does all this actually mean and what exactly happens? Well these are the steps taken for application server authentication:

1. The application server signs some JSON information with it's **private application key**.

2. This signed information is sent to the push service as a header in a POST request.
3. The push service uses the stored public key it received from `pushManager.subscribe()` to check the received information is signed by the private key relating to the public key. *Remember*: The public key is the `applicationServerKey` passed into the subscribe call.
4. If the signed information is valid the push service sends the push message to the user.

An example of this flow of information is below. (Note the legend in the bottom left to indicate public and private keys.)



The "signed information" added to a header in the request is a JSON Web Token.

JSON web token

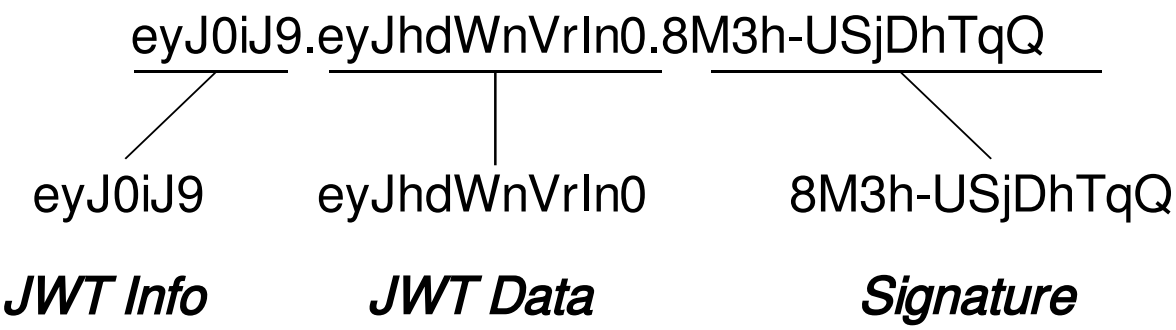
A JSON web token (or JWT for short) is a way of sending a message to a third party such that the receiver can validate who sent it.

When a third party receives a message, they need to get the senders public key and use it to validate the signature of the JWT. If the signature is valid then the JWT must have been signed with the matching private key so must be from the expected sender.

There are a host of libraries on https://jwt.io/ that can perform the signing for you and I'd recommend you do that where you can. For completeness, let's look at how to manually create a signed JWT.

Web push and signed JWTs

A signed JWT is just a string, though it can be thought of as three strings joined by dots.



The first and second strings (The JWT info and JWT data) are pieces of JSON that have been base64 encoded, meaning it's publicly readable.

The first string is information about the JWT itself, indicating which algorithm was used to create the signature.

The JWT info for web push must contain the following information:

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

The second string is the JWT Data. This provides information about the sender of the JWT, who it's intended for and how long it's valid.

For web push, the data would have this format:

```
{
  "aud": "https://some-push-service.org",
  "exp": "1469618703",
  "sub": "mailto:example@web-push-book.org"
}
```

The **aud** value is the "audience", i.e. who the JWT is for. For web push the audience is the push service, so we set it to the **origin of the push service**.

The **exp** value is the expiration of the JWT, this prevent snoopers from being able to re-use a JWT if they intercept it. The expiration is a timestamp in seconds and must be no longer 24 hours.

In Node.js the expiration is set using:

```
Math.floor(Date.now() / 1000) + (12 * 60 * 60)
```

It's 12 hours rather than 24 hours to avoid any issues with clock differences between the sending application and the push service.

Finally, the `sub` value needs to be either a URL or a `mailto` email address. This is so that if a push service needed to reach out to sender, it can find contact information from the JWT. (This is why the web-push library needed an email address).

Just like the JWT Info, the JWT Data is encoded as a URL safe base64 string.

The third string, the signature, is the result of taking the first two strings (the JWT Info and JWT Data), joining them with a dot character, which we'll call the "unsigned token", and signing it.

The signing process requires encrypting the "unsigned token" using ES256. According to the JWT spec, ES256 is short for "ECDSA using the P-256 curve and the SHA-256 hash algorithm". Using web crypto you can create the signature like so:

```
// Utility function for UTF-8 encoding a string to an ArrayBuffer.
const utf8Encoder = new TextEncoder('utf-8');

// The unsigned token is the concatenation of the URL-safe base64 encoded
// header and body.
const unsignedToken = .....;

// Sign the |unsignedToken| using ES256 (SHA-256 over ECDSA).
const key = {
  kty: 'EC',
  crv: 'P-256',
  x: window.uint8ArrayToBase64Url(
    applicationServerKeys.publicKey.subarray(1, 33)),
  y: window.uint8ArrayToBase64Url(
    applicationServerKeys.publicKey.subarray(33, 65)),
  d: window.uint8ArrayToBase64Url(applicationServerKeys.privateKey),
};

// Sign the |unsignedToken| with the server's private key to generate
// the signature.
return crypto.subtle.importKey('jwk', key, {
  name: 'ECDSA', namedCurve: 'P-256',
}, true, ['sign'])
.then((key) => {
  return crypto.subtle.sign({
    name: 'ECDSA',
    hash: {
      name: 'SHA-256',
    },
  }, key, utf8Encoder.encode(unsignedToken));
```

```
})
.then((signature) => {
  console.log('Signature: ', signature);
});
```

A push service can validate a JWT using the public application server key to decrypt the signature and make sure the decrypted string is the same as the "unsigned token" (i.e. the first two strings in the JWT).

The signed JWT (i.e. all three strings joined by dots), is sent to the web push service as the **`Authorization`** header with **`WebPush`** prepended, like so:

```
Authorization: 'WebPush <JWT Info>.<JWT Data>.<Signature>'
```

The Web Push Protocol also states the public application server key must be sent in the **`Crypto-Key`** header as a URL safe base64 encoded string with **`p256ecdsa=`** prepended to it.

```
Crypto-Key: p256ecdsa=<URL Safe Base64 Public Application Server Key>
```

The Payload Encryption

Next let's look at how we can send a payload with a push message so that when our web app receives a push message, it can access the data it receives.

A common question that arises from any who've used other push services is why does the web push payload need to be encrypted? With native apps, push messages can send data as plain text.

Part of the beauty of web push is that because all push services use the same API (the web push protocol), developers don't have to care who the push service is. We can make a request in the right format and expect a push message to be sent. The downside of this is that developers could conceivably send messages to a push service that isn't trustworthy. By encrypting the payload, a push service can't read the data that's sent. Only the browser can decrypt the information. This protects the user's data.

The encryption of the payload is defined in the Message Encryption spec.

Before we look at the specific steps to encrypt a push messages payload, we should cover some techniques that'll be used during the encryption process. (Massive hat tip to Mat Scales for his excellent article on push encryption.)

ECDH and HKDF

Both ECDH and HKDF are used throughout the encryption process and offer benefits for the purpose of encrypting information.

ECDH: Elliptic Curve Diffie-Hellman key exchange

Imagine you have two people who want to share information, Alice and Bob. Both Alice and Bob have their own public and private keys. Alice and Bob share their public keys with each other.

The useful property of keys generated with ECDH is that Alice can use her private key and Bob's public key to create secret value 'X'. Bob can do the same, taking his private key and Alice's public key to independently create the same value 'X'. This makes 'X' a shared secret and Alice and Bob only had to share their public key. Now Bob and Alice can use 'X' to encrypt and decrypt messages between them.

ECDH, to the best of my knowledge, defines the properties of curves which allow this "feature" of making a shared secret 'X'.

This is a high level explanation of ECDH, if you want to learn more I recommend checking out this video.

In terms of code; most languages / platforms come with libraries to make it easy to generate these keys.

In node we'd do the following:

```
const keyCurve = crypto.createECDH('prime256v1');
keyCurve.generateKeys();

const publicKey = keyCurve.getPublicKey();
const privateKey = keyCurve.getPrivateKey();
```

HKDF: HMAC based key derivation function

Wikipedia has a succinct description of HKDF:

> HKDF is an HMAC based key derivation function that transforms any weak key material into cryptographically strong key material. It can be used, for example, to convert Diffie Hellman exchanged shared secrets into key material suitable for use in encryption, integrity checking or authentication.

– Wikipedia

Essentially, HKDF will take input that is not particular secure and make it more secure.

The spec defining this encryption requires use of SHA-256 as our hash algorithm and the resulting keys for HKDF in web push should be no longer than 256 bits (32 bytes).

In node this could be implemented like so:

```
// Simplified HKDF, returning keys up to 32 bytes long
function hkdf(salt, ikm, info, length) {
  // Extract
  const keyHmac = crypto.createHmac('sha256', salt);
  keyHmac.update(ikm);
  const key = keyHmac.digest();

  // Expand
  const infoHmac = crypto.createHmac('sha256', key);
  infoHmac.update(info);

  // A one byte long buffer containing only 0x01
  const ONE_BUFFER = new Buffer(1).fill(1);
  infoHmac.update(ONE_BUFFER);

  return infoHmac.digest().slice(0, length);
}
```

Hat tip to Mat Scale's article for this example code.

This loosely covers ECDH and HKDF.

ECDH a secure way to share public keys and generate a shared secret. HKDF is a way to take insecure material and make it secure.

This will be used during the encryption of our payload. Next let's look at what we take as input and how that's encrypted.

Inputs

When we want to send a push message to a user with a payload, there are three inputs we need:

1. The payload itself.
2. The **auth** secret from the **PushSubscription**.

3. The **p256dh** key from the **PushSubscription**.

We've seen the **auth** and **p256dh** values being retrieved from a **PushSubscription** but for a quick reminder, given a subscription we'd need these values:

```
subscription.joJSON().keys.auth
subscription.joJSON().keys.p256dh

subscription.getKey('auth')
subscription.getKey('p256dh')
```

The **auth** value should be treated as a secret and not shared outside of your application.

The **p256dh** key is a public key, this is sometimes referred to as the client public key. Here we'll refer to **p256dh** as the subscription public key. The subscription public key is generated by the browser. The browser will keep the private key secret and use it for decrypting the payload.

These three values, **auth**, **p256dh** and **payload** are needed as inputs and the result of the encryption process will be the encrypted payload, a salt value and a public key used just for encrypting the data.

Salt

The salt needs to be 16 bytes of random data. In NodeJS, we'd do the following to create a salt:

```
const salt = crypto.randomBytes(16);
```

Public / Private Keys

The public and private keys should be generated using a P-256 elliptic curve, which we'd do in Node like so:

```
const localKeysCurve = crypto.createECDH('prime256v1');
localKeysCurve.generateKeys();

const localPublicKey = localKeysCurve.getPublicKey();
const localPrivateKey = localKeysCurve.getPrivateKey();
```

We'll refer to these keys as "local keys". They are used *just* for encryption and have *nothing* to do with application server keys.

With the payload, auth secret and subscription public key as inputs and with a newly generated salt and set of local keys, we are ready to actually do some encryption.

Shared secret

The first step is to create a shared secret using the subscription public key and our new private key (remember the ECDH explanation with Alice and Bob? Just like that).

```
const sharedSecret = localKeysCurve.computeSecret(
  subscription.keys.p256dh, 'base64');
```

This is used in the next step to calculate the Pseudo Random Key (PRK).

Pseudo random key

The Pseudo Random Key (PRK) is the combination of the push subscription's auth secret, and the shared secret we just created.

```
const authEncBuff = new Buffer('Content-Encoding: auth\0', 'utf8');
const prk = hkdf(subscription.keys.auth, sharedSecret, authEncBuff, 32);
```

You might be wondering what the **`Content-Encoding: auth\0`** string is for. In short, it doesn't have a clear purpose, although browsers could decrypt an incoming message and look for the expected content-encoding. The **`\0`** adds a byte with a value of 0 to end of the Buffer. This is expected by browsers decrypting the message who will expect so many bytes for the content encoding, followed a byte with value 0, followed by the encrypted data.

Our Pseudo Random Key is simply running the auth, shared secret and a piece of encoding info through HKDF (i.e. making it cryptographically stronger).

Context

The "context" is a set of bytes that is used to calculate two values later on in the encryption browser. It's essentially an array of bytes containing the subscription public key and the local public key.

```
const keyLabel = new Buffer('P-256\0', 'utf8');

// Convert subscription public key into a buffer.
const subscriptionPubKey = new Buffer(subscription.keys.p256dh, 'base64');
```

```
const subscriptionPubKeyLength = new Uint8Array(2);
subscriptionPubKeyLength[0] = 0;
subscriptionPubKeyLength[1] = subscriptionPubKey.length;

const localPublicKeyLength = new Uint8Array(2);
subscriptionPubKeyLength[0] = 0;
subscriptionPubKeyLength[1] = localPublicKey.length;

const contextBuffer = Buffer.concat([
  keyLabel,
  subscriptionPubKeyLength.buffer,
  subscriptionPubKey,
  localPublicKeyLength.buffer,
  localPublicKey,
]);
```

The final context buffer is a label, the number of bytes in the subscription public key, followed by the key itself, then the number of bytes local public key, followed by the key itself.

With this context value we can use it in the creation of a nonce and a content encryption key (CEK).

Content encryption key and nonce

A nonce is a value that prevents replay attacks as it should only be used once.

The content encryption key (CEK) is the key that will ultimately be used to encrypt our payload.

First we need to create the bytes of data for the nonce and CEK, which is simply a content encoding string followed by the context buffer we just calculated:

```
const nonceEncBuffer = new Buffer('Content-Encoding: nonce\0', 'utf8');
const nonceInfo = Buffer.concat([nonceEncBuffer, contextBuffer]);

const cekEncBuffer = new Buffer('Content-Encoding: aesgcm\0');
const cekInfo = Buffer.concat([cekEncBuffer, contextBuffer]);
```

This information is run through HKDF combining the salt and PRK with the nonceInfo and cekInfo:

```
// The nonce should be 12 bytes long
const nonce = hkdf(salt, prk, nonceInfo, 12);
```

```
// The CEK should be 16 bytes long
const contentEncryptionKey = hkdf(salt, prk, cekInfo, 16);
```

This gives us our nonce and content encryption key.

Perform the encryption

Now that we have our content encryption key, we can encrypt the payload.

We create an AES128 cipher using the content encryption key as the key and the nonce is an initialization vector.

In Node this is done like so:

```
const cipher = crypto.createCipheriv(
  'id-aes128-GCM', contentEncryptionKey, nonce);
```

Before we encrypt our payload, we need to define how much padding we wish to add to the front of the payload. The reason we'd want to add padding is that it prevents the risk of eavesdroppers being able to determine "types" of messages based on the payload size.

You must add two bytes of padding to indicate the length of any additional padding.

For example, if you added no padding, you'd have two bytes with value 0, i.e. no padding exists, after these two bytes you'll be reading the payload. If you added 5 bytes of padding, the first two bytes will have a value of 5, so the consumer will then read an additional five bytes and then start reading the payload.

```
const padding = new Buffer(2 + paddingLength);
// The buffer must be only zeros, except the length
padding.fill(0);
padding.writeUInt16BE(paddingLength, 0);
```

We then run our padding and payload through this cipher.

```
const result = cipher.update(Buffer.concat(padding, payload));
cipher.final();

// Append the auth tag to the result -
// https://nodejs.org/api/crypto.html#crypto_cipher_getauthtag
const encryptedPayload = Buffer.concat([result, cipher.getAuthTag()]);
```

We now have our encrypted payload. Yay!

All that remains is to determine how this payload is sent to the push service.

Encrypted payload headers & body

To send this encrypted payload to the push service we need to define a few different headers in our POST request.

Encryption header

The 'Encryption' header must contain the *salt* used for encrypting the payload.

The 16 byte salt should be base64 URL safe encoded and added to the Encryption header, like so:

```
Encryption: salt=<URL Safe Base64 Encoded Salt>
```

Crypto-Key header

We saw that the **`Crypto-Key`** header is used under the 'Application Server Keys' section to contain the public application server key.

This header is also used to share the local public key used to encrypt the payload.

The resulting header looks like this:

```
Crypto-Key: dh=<URL Safe Base64 Encoded Local Public Key String>; p256ecdsa
Encoded Public Application Server Key>
```

Content type, length & encoding headers

The **`Content-Length`** header is the number of bytes in the encrypted payload. 'Content-Type' and 'Content-Encoding' headers are fixed values. This is shown below.

```
Content-Length: <Number of Bytes in Encrypted Payload>
Content-Type: 'application/octet-stream'
Content-Encoding: 'aesgcm'
```

With these headers set, we need to send the encrypted payload as the body of our request. Notice that the **`Content-Type`** is set to **`application/octet-stream`**. This is because the encrypted payload must be sent as a stream of bytes.

In NodeJS we would do this like so:

```
const pushRequest = https.request(httpsOptions, function(pushResponse) {
pushRequest.write(encryptedPayload);
pushRequest.end();
```

More headers?

We've covered the headers used for JWT / Application Server Keys (i.e. how to identify the application with the push service) and we've covered the headers used to send an encrypted payload.

There are additional headers that push services use to alter the behavior of sent messages. Some of these headers are required, while others are optional.

TTL header

Required

TTL (or time to live) is an integer specifying the number of seconds you want your push message to live on the push service before it's delivered. When the TTL expires, the message will be removed from the push service queue and it won't be delivered.

```
TTL: <Time to live in seconds>
```

If you set a TTL of zero, the push service will attempt to deliver the message immediately, **but** if the device can't be reached, your message will be immediately dropped from the push service queue.

Technically a push service can reduce the TTL of a push message if it wants. You can tell if this has happened by examining the TTL header in the response from a push service.

Topic

Optional

Topics are strings that can be used to replace a pending messages with a new message if they have matching topic names.

This is useful in scenarios where multiple messages are sent while a device is offline, and you really only want a user to see the latest message when the device is turned on.

Urgency

Optional

Urgency indicates to the push service how important a message is to the user. This can be used by the push service to help conserve the battery life of a user's device by only waking up for important messages when battery is low.

The header value is defined as shown below. The default value is `normal`.

```
Urgency: <very-low | low | normal | high>
```

Everything together

If you have further questions about how this all works you can always see how libraries trigger push messages on the web-push-libs org.

Once you have an encrypted payload, and the headers above, you just need to make a POST request to the `endpoint` in a `PushSubscription`.

So what do we do with the response to this POST request?

Response from push service

Once you've made a request to a push service, you need to check the status code of the response as that'll tell you whether the request was successful or not.

Status Code	Description
201	Created. The request to send a push message was received and accepted.
429	Too many requests. Meaning your application server has reached a rate limit with a push service. The push service should include a 'Retry-After' header to indicate how long before another request can be made.
400	Invalid request. This generally means one of your headers is invalid or improperly formatted.
404	Not Found. This is an indication that the subscription is expired and can't be used. In this case you should delete the `PushSubscription` and wait for the client to resubscribe the user.
410	Gone. The subscription is no longer valid and should be removed from application server. This can be reproduced by calling `unsubscribe()` on a `PushSubscription`.
413	Payload size too large. The minimum size payload a push service must support is 4096 bytes (or

4kb).

Previous
← Sending Messages with Web Push Libraries

Next
Handling Push Events →

Last updated July 2, 2018.