Build a push notifications server

- Remix the sample app and view it in a new tab
- Get familiar with the starting app and its code
- Generate and load VAPID details
  - Background
  - Implementation
- Implement functionality to send notifications
  - Background
  - Implementation
- Handle new subscriptions
  - Background
  - Implementation
- Handle subscription cancellations
  - Background
  - Implementation

In this codelab, you'll build a push notifications server. The server will manage a list of push subscriptions and send notifications to them.

The client code is already complete–in this codelab, you'll work on the server-side functionality.

Remix the sample app and view it in a new tab

Notifications are automatically blocked from the embedded Glitch app, so you won't be able to preview the app on this page. Instead, here's what to do:

{% Instruction 'remix', 'ol' %} {% Instruction 'preview', 'ol' %}

The live app opens in a new Chrome tab. In the embedded Glitch, click **View Source** to show the code again.

As you work through this codelab, make changes to the code in the embedded Glitch on this page. Refresh the new tab with your live app to see the changes.

Get familiar with the starting app and its code

Start by taking a look at the app's client UI.

In the new Chrome tab:

1. {% Instruction 'devtools-console', 'none' %}
2. Try clicking buttons in the UI (check the Chrome dev console for output).
   - **Register service worker** registers a service worker for the scope of your Glitch project URL. **Unregister service worker** removes the service worker. If a push subscription is attached to it, the push subscription will also be de-activated.
   - **Subscribe to push** creates a push subscription. It is only available when a service worker has been registered and a `VAPID_PUBLIC_KEY` constant is present in the client code (more about this later), so you can't click it just yet.
   - When you have an active push subscription, **Notify current subscription** requests that the server send a notification to its endpoint.
   - **Notify all subscriptions** tells the server to send a notification to all of the subscription endpoints in its database.
     Note that some of these endpoints might be inactive. It's always possible that a subscription will disappear by the time the server sends a notification to it.

Let's look at what's going on server-side. To see messages from the server code, look at the Node.js log within the Glitch interface.

- In the Glitch app, click **Tools -> Logs**.
  You'll probably see a message like `Listening on port 3000`.
  If you tried clicking **Notify current subscription** or **Notify all subscriptions** in the live app UI, you'll also see the following message:

```
TODO: Implement sendNotifications()
Endpoints to send to:  []
```

Now let's look at some code.

- `public/index.js` contains the completed client code. It performs feature detection, registers and unregisters the service worker, and controls the user's subscription to push notifications. It also sends information about new and deleted subscriptions to the server. Since you're only going to be working on the server functionality, you won't be editing this file (apart from populating the `VAPID_PUBLIC_KEY` constant).
- `public/service-worker.js` is a simple service worker that captures push events and displays notifications.
- `/views/index.html` contains the app UI.
- `.env` contains the environment variables that Glitch loads into your app server when it starts up. You'll populate `.env` with authentication details for sending notifications.
- `server.js` is the file you'll be doing most of your work in during this codelab.

  The starting code creates a simple Express web server. There are four TODO items for you, marked in code comments with `TODO:`. You need to:

  - Load VAPID details from environment variables.
  - Implement functionality to send notifications.
  - Handle new subscriptions.
  - Handle subscription cancellations.

  In this codelab, you'll work through these TODO items one at a time.

Generate and load VAPID details

Your first TODO item is to generate VAPID details, add them to the Node.js environment variables, and update the client and server code with the new values.

Background

When users subscribe to notifications, they need to trust the identity of the app and its server. Users also need to be confident that, when they receive a notification, it's from the same app that set up the subscription. They also need to trust that nobody else can read the notification content.

The protocol that makes push notifications secure and private is called Voluntary Application Server Identification for Web Push (VAPID). VAPID uses public key cryptography to verify the identity of apps, servers, and subscription endpoints, and to encrypt notification content.

In this app, you'll use the web-push npm package to generate VAPID keys, and to encrypt and send notifications.

Implementation

In this step, generate a pair of VAPID keys for your app and add them to the environment variables. Load the environment variables in the server and add the public key as a constant in the client code.

1. Use the `generateVAPIDKeys` function of the `web-push` library to create a pair of VAPID keys. In **server.js**, remove the comments from around the following lines of code:
   server.js
   ```
   // Generate VAPID keys (only do this once).
   /*
    * const vapidKeys = webpush.generateVAPIDKeys();
    * console.log(vapidKeys);
    */
   const vapidKeys = webpush.generateVAPIDKeys();
   console.log(vapidKeys);
   ```

2. After Glitch restarts your app, it outputs the generated keys to the Node.js log within the Glitch interface (**not** to the Chrome console). To see the VAPID keys, select **Tools -> Logs** in the Glitch interface.
   Make sure that you copy your public and private keys from the same key pair!
   Glitch restarts your app every time you edit your code, so the first pair of keys you generate might scroll out of view as more output follows.
3. In **.env**, copy and paste the VAPID keys. Enclose the keys in double quotes (`"..."`). For `VAPID_SUBJECT`, you can enter `"mailto:test@test.test"`.
   .env
   ```
   # process.env.SECRET
   VAPID_PUBLIC_KEY=
   VAPID_PRIVATE_KEY=
   VAPID_SUBJECT=
   VAPID_PUBLIC_KEY="BN3tWzHp3L3rBh03lGLlLlsq..."
   VAPID_PRIVATE_KEY="I_lM7JMIXRhOk6HN..."
   VAPID_SUBJECT="mailto:test@test.test"
   ```

4. In **server.js**, comment out those two lines of code again, since you only need to generate VAPID keys once.
   server.js
   ```
   // Generate VAPID keys (only do this once).
   /*
   const vapidKeys = webpush.generateVAPIDKeys();
   console.log(vapidKeys);
   */
   const vapidKeys = webpush.generateVAPIDKeys();
   ```

```
console.log(vapidKeys);
```

5. In **server.js**, load the VAPID details from the environment variables.
`server.js`

```
const vapidDetails = {
  // TODO: Load VAPID details from environment variables.
  publicKey: process.env.VAPID_PUBLIC_KEY,
  privateKey: process.env.VAPID_PRIVATE_KEY,
  subject: process.env.VAPID_SUBJECT
}
```

6. Copy and paste the **public** key into the client code as well.
In **public/index.js**, enter the same value for `VAPID_PUBLIC_KEY` that you copied into the .env file:
`public/index.js`

```
// Copy from .env
const VAPID_PUBLIC_KEY = '';
const VAPID_PUBLIC_KEY = 'BN3tWzHp3L3rBh03lGLlLlsq...';
```

{% Aside %}

The values for the environment variables in **.env** are unique to a single Glitch project. If you remix your code, you'll need to repeat the steps above in your new Glitch project.

{% endAside %}

Implement functionality to send notifications

Background

In this app, you'll use the web-push npm package to send notifications.

This package automatically encrypts notifications when `webpush.sendNotification()` is called, so you don't need to worry about that.

web-push accepts multiple options for notifications–for example, you can attach headers to the message, and specify content encoding.

In this codelab, you'll only use two options, defined with the following lines of code:

```
let options = {
  TTL: 10000; // Time-to-live. Notifications expire after this.
  vapidDetails: vapidDetails; // VAPID keys from .env
};
```

The `TTL` (time-to-live) option sets an expiry timeout on a notification. This is a way for the server to avoid sending a notification to a user after it is no longer relevant.

The `vapidDetails` option contains the VAPID keys you loaded from the environment variables.

Implementation

In **server.js**, modify the `sendNotifications` function as follows:

`server.js`

```
function sendNotifications(database, endpoints) {
  // TODO: Implement functionality to send notifications.
  console.log('TODO: Implement sendNotifications()');
  console.log('Endpoints to send to: ', endpoints);
  let notification = JSON.stringify(createNotification());
  let options = {
    TTL: 10000, // Time-to-live. Notifications expire after this.
    vapidDetails: vapidDetails // VAPID keys from .env
  };
  endpoints.map(endpoint => {
    let subscription = database[endpoint];
    webpush.sendNotification(subscription, notification, options);
  });
}
```

Since `webpush.sendNotification()` returns a promise, you can easily add error handling.

In **server.js**, modify the `sendNotifications` function again:

`server.js`

```
function sendNotifications(database, endpoints) {
  let notification = JSON.stringify(createNotification());
```

```
  let options = {
    TTL: 10000; // Time-to-live. Notifications expire after this.
    vapidDetails: vapidDetails; // VAPID keys from .env
  };
  endpoints.map(endpoint => {
    let subscription = database[endpoint];
    webpush.sendNotification(subscription, notification, options);
    let id = endpoint.substr((endpoint.length - 8), endpoint.length);
    webpush.sendNotification(subscription, notification, options)
    .then(result => {
      console.log(`Endpoint ID: ${id}`);
      console.log(`Result: ${result.statusCode} `);
    })
    .catch(error => {
      console.log(`Endpoint ID: ${id}`);
      console.log(`Error: ${error.body} `);
    });
  });
}
```

Handle new subscriptions

Background

Here's what happens when the user subscribes to push notifications:

1. User clicks **Subscribe to push**.
2. Client uses the `VAPID_PUBLIC_KEY` constant (the server's public VAPID key) to generate a unique, server-specific `subscription` object. The `subscription` object looks like this:

```
{
  "endpoint": "https://fcm.googleapis.com/fcm/send/cpqAgzGzkzQ:APA9...",
  "expirationTime": null,
  "keys":
  {
    "p256dh": "BNYDjQL9d5PSoeBurHy2e4d4GY0sGJXBN...",
    "auth": "0IyyvUGNJ9RxJc83poo3bA"
  }
```

```
}
```

3. Client sends a `POST` request to the `/add-subscription` URL, including the subscription as stringified JSON in the body.
4. Server retrieves the stringified `subscription` from the body of the POST request, parses it back to JSON, and adds it to the subscriptions database.
   The database stores subscriptions using their own endpoints as a key:

```
{
  "https://fcm...1234": {
    endpoint: "https://fcm...1234",
    expirationTime: ...,
    keys: { ... }
  },
  "https://fcm...abcd": {
    endpoint: "https://fcm...abcd",
    expirationTime: ...,
    keys: { ... }
  },
  "https://fcm...zxcv": {
    endpoint: "https://fcm...zxcv",
    expirationTime: ...,
    keys: { ... }
  },
}
```

Now, the new subscription is available to the server for sending notifications.

Implementation

Requests for new subscriptions come to the `/add-subscription` route, which is a POST URL. You'll see a stub route handler in **server.js**:

`server.js`

```
app.post('/add-subscription', (request, response) => {
  // TODO: implement handler for /add-subscription
  console.log('TODO: Implement handler for /add-subscription');
  console.log('Request body: ', request.body);
  response.sendStatus(200);
});
```

In your implementation, this handler must:

- Retrieve the new subscription from the body of the request.
- Access the database of active subscriptions.
- Add the new subscription to the list of active subscriptions.

{% Aside %}

In this example, we use the `express-session` npm package to store a list of active subscriptions in a session variable.

This works for a demonstration, but it's not suitable for production. For compatible production packages, see the express-session documentation.

{% endAside %}

To handle new subscriptions:

- In **server.js**, modify the route handler for `/add-subscription` as follows:
  server.js

```
app.post('/add-subscription', (request, response) => {
  // TODO: implement handler for /add-subscription
  console.log('TODO: Implement handler for /add-subscription');
  console.log('Request body: ', request.body);
  let subscriptions = Object.assign({}, request.session.subscriptions);
  subscriptions[request.body.endpoint] = request.body;
  request.session.subscriptions = subscriptions;
  response.sendStatus(200);
});
```

Handle subscription cancellations

Background

The server won't always know when a subscription becomes inactive–for example, a subscription could be wiped when the browser shuts down the service worker.

The server can, however, find out about subscriptions that are cancelled through the app UI. In this step, you'll implement functionality to remove a subscription from the database.

This way, the server avoids sending out a bunch of notifications to non-existent endpoints. Obviously this doesn't really matter with a simple test app, but it becomes important at a larger scale.

Implementation

Requests to cancel subscriptions come to the `/remove-subscription` POST URL.

The stub route handler in **server.js** looks like this:

server.js

```
app.post('/remove-subscription', (request, response) => {
  // TODO: implement handler for /remove-subscription
  console.log('TODO: Implement handler for /remove-subscription');
  console.log('Request body: ', request.body);
  response.sendStatus(200);
});
```

In your implementation, this handler must:

- Retrieve the endpoint of the cancelled subscription from the body of the request.
- Access the database of active subscriptions.
- Remove the cancelled subscription from the list of active subscriptions.

The body of the POST request from the client contains the endpoint that you need to remove:

```
{
  "endpoint": "https://fcm.googleapis.com/fcm/send/cpqAgzGzkzQ:APA9..."
}
```

To handle subscription cancellations:

- In **server.js**, modify the route handler for `/remove-subscription` as follows:
  server.js

  ```
  app.post('/remove-subscription', (request, response) => {
    // TODO: implement handler for /remove-subscription
    console.log('TODO: Implement handler for /remove-subscription');
    console.log('Request body: ', request.body);
    let subscriptions = Object.assign({}, request.session.subscriptions);
    delete subscriptions[request.body.endpoint];
  ```

```
  request.session.subscriptions = subscriptions;
  response.sendStatus(200);
});
```

{% Aside 'success' %} You have implemented a push notifications server. Try it out in the live app UI! Register a service worker, subscribe to push, and send yourself some test notifications. {% endAside %}

□□□ #2
□ □□□ □□□ 2026-10-31 23:00:16 UTC
□ □□□ □□□ 2026-10-31 23:00:18 UTC