How To Send Web Push Notifications from Django Applications

Posted October 24, 2018 41.7k UBUNTU 18.04 DJANGO APPLICATIONS PROGRAMMING PROJECT

By Richard Umoffia
Become an author

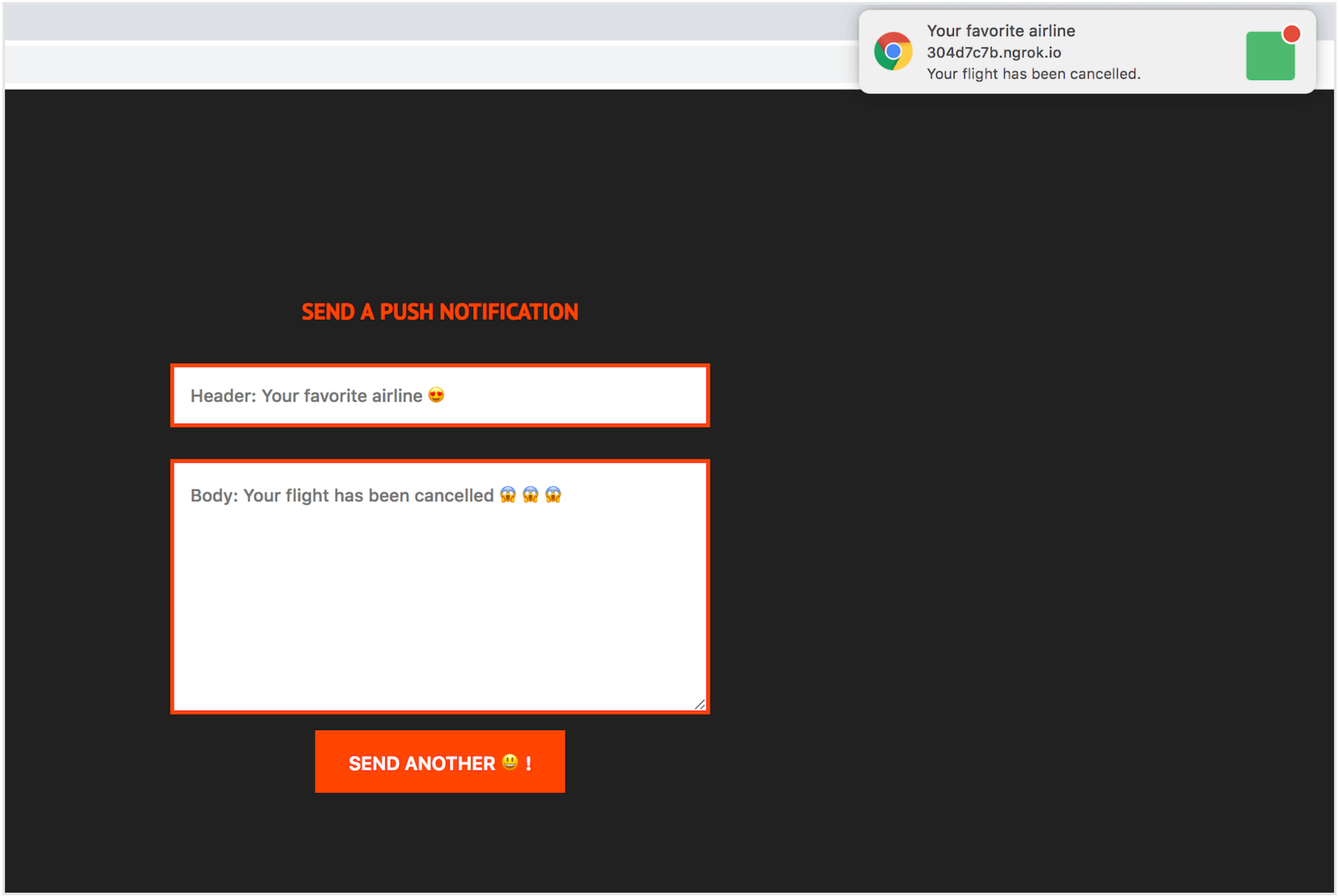
The author selected the Open Internet/Free Speech Fund to receive a donation as part of the Write for DOnations program.

Introduction

The web is constantly evolving, and it can now achieve functionalities that were formerly only available on native mobile devices. The introduction of JavaScript service workers gave the web newfound abilities to do things like background syncing, offline caching, and sending push notifications.

Push notifications allow users to opt-in to receive updates to mobile and web applications. They also enable users to re-engage with existing applications using customized and relevant content.

In this tutorial, you'll set up a Django application on Ubuntu 18.04 that sends push notifications whenever there's an activity that requires the user to visit the application. To create these notifications, you will use the Django-Webpush package and set up and register a service worker to display notifications to the client. The working application with notifications will look like this:



Your favorite airline
304d7c7b.ngrok.io
Your flight has been cancelled.

SEND A PUSH NOTIFICATION

Header: Your favorite airline 😍

Body: Your flight has been cancelled 😱 😱 😱

Prerequisites

Before you begin this guide you'll need the following:

- One Ubuntu 18.04 server with a non-root user and an active firewall. You can follow the guidelines in this initial server setup guide for more information on how to create an Ubuntu 18.04 server.
- `pip` and `venv` installed following these guidelines.
- A Django project called `djangopush` created in your home directory, set up following these guidelines on creating a sample Django project on Ubuntu 18.04. Be sure to add your server's IP address to the `ALLOWED_HOSTS` directive in your `settings.py` file.

Step 1 — Installing Django-Webpush and Getting Vapid Keys

Django-Webpush is a package that enables developers to integrate and send web push notifications in Django applications. We'll use this package to trigger and send push notifications from our application. In this step, you will install Django-Webpush and obtain the *Voluntary Application Server Identification (VAPID)* keys that are necessary for identifying your server and ensuring the uniqueness of each request.

Make sure you are in the `~/djangopush` project directory that you created in the prerequisites:

```
$ cd ~/djangopush
```

Activate your virtual environment:

```
$ source my_env/bin/activate
```

Upgrade your version of `pip` to ensure it's up-to-date:

```
(my_env) $ pip install --upgrade pip
```

Install Django-Webpush:

```
(my_env) $ pip install django-webpush
```

After installing the package, add it to the list of applications in your `settings.py` file. First open `settings.py`:

```
(my_env) $ nano ~/djangopush/djangopush/settings.py
```

Add `webpush` to the list of `INSTALLED_APPS`:

~/djangopush/djangopush/settings.py

```
...

INSTALLED_APPS = [
    ...,
    'webpush',
]
...
```

Save the file and exit your editor.

Run migrations on the application to apply the changes you've made to your database schema:

```
(my_env) $ python manage.py migrate
```

The output will look like this, indicating a successful migration:

```
Output
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, webpush
Running migrations:
  Applying webpush.0001_initial... OK
```

The next step in setting up web push notifications is getting VAPID keys. These keys identify the application server and can be used to reduce the secrecy for push subscription URLs, since they restrict subscriptions to a specific server.

To obtain VAPID keys, navigate to the wep-push-codelab web application. Here, you'll be given automatically generated keys. Copy the private and public keys.

Next, create a new entry in `settings.py` for your VAPID information. First, open the file:

```
(my_env) $ nano ~/djangopush/djangopush/settings.py
```

Next, add a new directive called `WEBPUSH_SETTINGS` with your VAPID public and private keys and your email below `AUTH_PASSWORD_VALIDATORS`:

~/djangopush/djangopush/settings.py

```
...

AUTH_PASSWORD_VALIDATORS = [
    ...
]

WEBPUSH_SETTINGS = {
   "VAPID_PUBLIC_KEY": "your_vapid_public_key",
   "VAPID_PRIVATE_KEY": "your_vapid_private_key",
   "VAPID_ADMIN_EMAIL": "admin@example.com"
}

# Internationalization
# https://docs.djangoproject.com/en/2.0/topics/i18n/

...
```

Don't forget to replace the placeholder values `your_vapid_public_key`, `your_vapid_private_key`, and `admin@example.com` with your own information. Your email address is how you will be notified if the push server experiences any issues.

Next, we'll set up views that will display the application's home page and trigger push notifications to subscribed users.

Step 2 — Setting Up Views

In this step, we'll setup a basic `home` *view* with the `HttpResponse` response object for our home page, along with a `send_push` view. Views are functions that return response objects from web requests. The `send_push` view will use the Django-Webpush library to send push notifications that contain the data entered by a user on the home page.

Navigate to the `~/djangopush/djangopush` folder:

```
(my_env) $ cd ~/djangopush/djangopush
```

Running `ls` inside the folder will show you the project's main files:

```
Output
/__init__.py
/settings.py
/urls.py
/wsgi.py
```

The files in this folder are auto-generated by the `django-admin` utility that you used to create your project in the prerequisites. The `settings.py` file contains project-wide configurations like installed applications and the static root folder. The `urls.py` file contains the URL configurations for the project. This is where you will set up routes to match your created views.

Create a new file inside the `~/djangopush/djangopush` directory called `views.py`, which will contain the views for your project:

```
(my_env) $ nano ~/djangopush/djangopush/views.py
```

The first view we'll make is the `home` view, which will display the home page where users can send push notifications. Add the following code to the file:

~/djangopush/djangopush/views.py

```
from django.http.response import HttpResponse
from django.views.decorators.http import require_GET

@require_GET
def home(request):
    return HttpResponse('<h1>Home Page<h1>')
```

The `home` view is decorated by the `require_GET` decorator, which restricts the view to GET requests only. A view typically returns a response for every request made to it. This view returns a simple HTML tag as a response.

The next view we'll create is `send_push`, which will handle sent push notifications using the `django-webpush` package. It will be restricted to POST requests only and will be exempted from *Cross Site Request Forgery* (CSRF) protection. Doing this will allow you to test the view using Postman or any other RESTful service. In production, however, you should remove this decorator to avoid leaving your views vulnerable to CSRF.

To create the `send_push` view, first add the following imports to enable JSON responses and access the `send_user_notification` function in the `webpush` library:

~/djangopush/djangopush/views.py

```python
from django.http.response import JsonResponse, HttpResponse
from django.views.decorators.http import require_GET, require_POST
from django.shortcuts import get_object_or_404
from django.contrib.auth.models import User
from django.views.decorators.csrf import csrf_exempt
from webpush import send_user_notification
import json
```

Next, add the `require_POST` decorator, which will use the request body sent by the user to create and trigger a push notification:

~/djangopush/djangopush/views.py

```python
@require_GET
def home(request):
    ...


@require_POST
@csrf_exempt
def send_push(request):
    try:
        body = request.body
        data = json.loads(body)

        if 'head' not in data or 'body' not in data or 'id' not in data:
            return JsonResponse(status=400, data={"message": "Invalid data format"})

        user_id = data['id']
        user = get_object_or_404(User, pk=user_id)
        payload = {'head': data['head'], 'body': data['body']}
        send_user_notification(user=user, payload=payload, ttl=1000)

        return JsonResponse(status=200, data={"message": "Web push successful"})
    except TypeError:
        return JsonResponse(status=500, data={"message": "An error occurred"})
```

We are using two decorators for the `send_push` view: the `require_POST` decorator, which restricts the view to POST requests only, and the `csrf_exempt` decorator, which exempts the view from CSRF protection.

This view expects POST data and does the following: it gets the `body` of the request and, using the json package, deserializes the JSON document to a Python object using `json.loads`. `json.loads` takes a structured JSON document and converts it to a Python object.

The view expects the request body object to have three properties:

- `head`: The title of the push notification.
- `body`: The body of the notification.
- `id`: The `id` of the request user.

If any of the required properties are missing, the view will return a `JSONResponse` with a 404 “Not Found” status. If the user with the given primary key exists, the view will return the `user` with the matching primary key using the `get_object_or_404` function from the `django.shortcuts` library. If the user doesn’t exist, the function will return a 404 error.

The view also makes use of the `send_user_notification` function from the `webpush` library. This function takes three parameters:

- `User`: The recipient of the push notification.
- `payload`: The notification information, which includes the notification `head` and `body`.
- `ttl`: The maximum time in seconds that the notification should be stored if the user is offline.

If no errors occur, the view returns a `JSONResponse` with a 200 “Success” status and a data object. If a `KeyError` occurs, the view will return a 500 “Internal Server Error” status. A `KeyError` occurs when the requested key of an object doesn’t exist.

In the next step, we’ll create corresponding URL routes to match the views we’ve created.

Step 3 — Mapping URLs to Views

Django makes it possible to create URLs that connect to views with a Python module called a `URLconf`. This module maps URL path expressions to Python functions (your views). Usually, a URL configuration file is auto-generated when you create a project. In this step, you will update this file to include new routes for the views you created in the previous step, along with the URLs for the `django-webpush` app, which will provide endpoints to subscribe users to push notifications.

For more information about views, please see How To Create Django Views.

Open `urls.py`:

```
(my_env) $ nano ~/djangopush/djangopush/urls.py
```

The file will look like this:

~/djangopush/djangopush/urls.py

```
"""untitled URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/2.1/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

The next step is to map the views you've created to URLs. First, add the `include` import to ensure that all of the routes for the Django-Webpush library will be added to your project:

~/djangopush/djangopush/urls.py

```
"""webpushdjango URL Configuration
...
"""
from django.contrib import admin
from django.urls import path, include
```

Next, import the views you created in the last step and update the `urlpatterns` list to map your views:

~/djangopush/djangopush/urls.py

```
"""webpushdjango URL Configuration
...
"""
from django.contrib import admin
from django.urls import path, include

from .views import home, send_push

urlpatterns = [
                  path('admin/', admin.site.urls),
                  path('', home),
                  path('send_push', send_push),
                  path('webpush/', include('webpush.urls')),
              ]
```

Here, the `urlpatterns` list registers the URLs for the `django-webpush` package and maps your views to the URLs `/send_push` and `/home`.

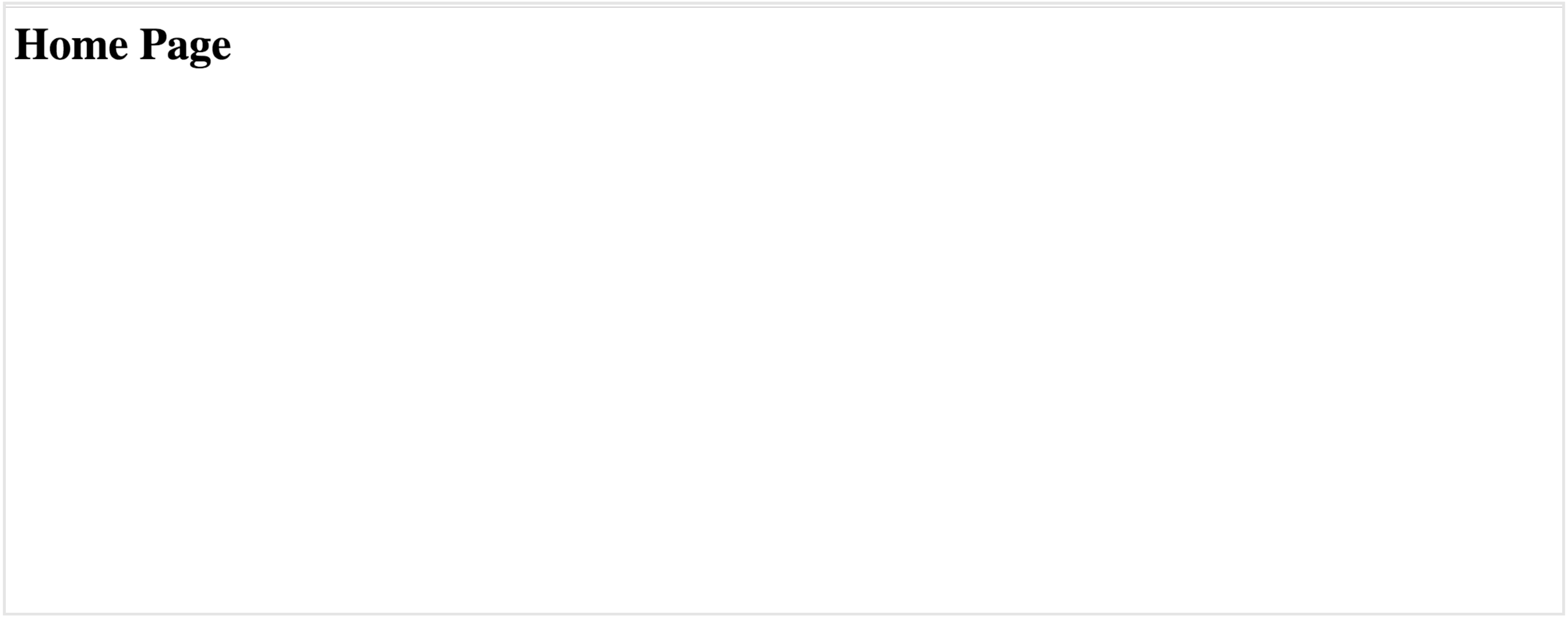
Let's test the `/home` view to be sure that it's working as intended. Make sure you're in the root directory of the project:

```
(my_env) $ cd ~/djangopush
```

Start your server by running the following command:

```
(my_env) $ python manage.py runserver your_server_ip:8000
```

Navigate to `http://your_server_ip:8000`. You should see the following home page:


Home Page

At this point, you can kill the server with `CTRL+C`, and we will move on to creating templates and rendering them in our views using the `render` function.

Step 4 — Creating Templates

Django's template engine allows you to define the user-facing layers of your application with templates, which are similar to HTML files. In this step, you will create and render a template for the `home` view.

Create a folder called `templates` in your project's root directory:

```
(my_env) $ mkdir ~/djangopush/templates
```

If you run `ls` in the root folder of your project at this point, the output will look like this:

```
Output
/djangopush
/templates
db.sqlite3
manage.py
/my_env
```

Create a file called `home.html` in the `templates` folder:

```
(my_env) $ nano ~/djangopush/templates/home.html
```

Add the following code to the file to create a form where users can enter information to create push notifications:

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <meta name="vapid-key" content="{{ vapid_key }}">
    {% if user.id %}
        <meta name="user_id" content="{{ user.id }}">
    {% endif %}
    <title>Web Push</title>
    <link href="https://fonts.googleapis.com/css?family=PT+Sans:400,700" rel="stylesheet">
</head>

<body>
<div>
    <form id="send-push__form">
        <h3 class="header">Send a push notification</h3>
        <p class="error"></p>
        <input type="text" name="head" placeholder="Header: Your favorite airline 😍">
        <textarea name="body" id="" cols="30" rows="10" placeholder="Body: Your flight has been cancelled 😱😱😱"></textare
        <button>Send Me</button>
    </form>
</div>
</body>
</html>
```

The `body` of the file includes a form with two fields: an `input` element will hold the head/title of the notification and a `textarea` element will hold the notification body.

In the `head` section of the file, there are two `meta` tags that will hold the VAPID public key and the user's id. These two variables are required to register a user and send them push notifications. The user's id is required here because you'll be sending AJAX requests to the server and the `id` will be used to identify the user. If the current user is a registered user, then the template will create a `meta` tag with their `id` as the content.

The next step is to tell Django where to find your templates. To do this, you will edit `settings.py` and update the `TEMPLATES` list.

Open the `settings.py` file:

```
(my_env) $ nano ~/djangopush/djangopush/settings.py
```

Add the following to the `DIRS` list to specify the path to the templates directory:

~/djangopush/djangopush/settings.py

```
...
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                ...
            ],
        },
    },
]
...
```

Next, in your `views.py` file, update the `home` view to render the `home.html` template. Open the file:

```
(my_env) $ nano ~/djangpush/djangopush/views.py
```

First, add some additional imports, including the `settings` configuration, which contains all of the project's settings from the `settings.py` file, and the `render` function from `django.shortcuts`:

~/djangopush/djangopush/views.py

```
...
from django.shortcuts import render, get_object_or_404
...
import json
from django.conf import settings

...
```

Next, remove the initial code you added to the `home` view and add the following, which specifies how the template you just created will be rendered:

~/djangopush/djangopush/views.py

```
...

@require_GET
def home(request):
   webpush_settings = getattr(settings, 'WEBPUSH_SETTINGS', {})
   vapid_key = webpush_settings.get('VAPID_PUBLIC_KEY')
```

```
    user = request.user
    return render(request, 'home.html', {user: user, 'vapid_key': vapid_key})
```

The code assigns the following variables:

- `webpush_settings`: This is assigned the value of the `WEBPUSH_SETTINGS` attribute from the `settings` configuration.
- `vapid_key`: This gets the `VAPID_PUBLIC_KEY` value from the `webpush_settings` object to send to the client. This public key is checked against the private key to ensure that the client with the public key is permitted to receive push messages from the server.
- `user`: This variable comes from the incoming request. Whenever a user makes a request to the server, the details for that user are stored in the `user` field.

The `render` function will return an HTML file and a context object containing the current user and the server's vapid public key. It takes three parameters here: the `request`, the `template` to be rendered, and the object that contains the variables that will be used in the template.

With our template created and the `home` view updated, we can move on to configuring Django to serve our static files.

Step 5 — Serving Static Files

Web applications include CSS, JavaScript, and other image files that Django refers to as "static files". Django allows you to collect all of the static files from each application in your project into a single location from which they are served. This solution is called `django.contrib.staticfiles`. In this step, we'll update our settings to tell Django where our static files will be stored.

Open `settings.py`:

```
(my_env) $ nano ~/djangopush/djangopush/settings.py
```

In `settings.py`, first ensure that the `STATIC_URL` has been defined:

~/djangopush/djangopush/settings.py

```
...
STATIC_URL = '/static/'
```

Next, add a list of directories called `STATICFILES_DIRS` where Django will look for static files:

~/djangopush/djangopush/settings.py

```
...
STATIC_URL = '/static/'
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),
]
```

You can now add the `STATIC_URL` to the list of paths defined in your `urls.py` file.

Open the file:

```
(my_env) $ nano ~/djangopush/djangopush/urls.py
```

Add the following code, which will import the `static` url configuration and update the `urlpatterns` list. The helper function here uses the `STATIC_URL` and `STATIC_ROOT` properties we provided in the `settings.py` file to serve the project's static files:

~/djangopush/djangopush/urls.py

```
...
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    ...
]  + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

With our static files settings configured, we can move on to styling the application's home page.

Step 6 — Styling the Home Page

After setting up your application to serve static files, you can create an external stylesheet and link it to the `home.html` file to style the home page. All of your static files will be stored in a `static` directory in the root folder of your project.

Create a `static` folder and a `css` folder within the `static` folder:

```
(my_env) $ mkdir -p ~/djangopush/static/css
```

Open a css file called `styles.css` inside the `css` folder:

```
(my_env) $ nano ~/djangopush/static/css/styles.css
```

Add the following styles for the home page:

~/djangopush/static/css/styles.css

```
body {
    height: 100%;
    background: rgba(0, 0, 0, 0.87);
    font-family: 'PT Sans', sans-serif;
}

div {
    height: 100%;
    display: flex;
    align-items: center;
    justify-content: center;
}

form {
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
    width: 35%;
    margin: 10% auto;
```

```
}

form > h3 {
    font-size: 17px;
    font-weight: bold;
    margin: 15px 0;
    color: orangered;
    text-transform: uppercase;
}

form > .error {
    margin: 0;
    font-size: 15px;
    font-weight: normal;
    color: orange;
    opacity: 0.7;
}

form > input, form > textarea {
    border: 3px solid orangered;
    box-shadow: unset;
    padding: 13px 12px;
    margin: 12px auto;
    width: 80%;
    font-size: 13px;
    font-weight: 500;
}

form > input:focus, form > textarea:focus {
    border: 3px solid orangered;
    box-shadow: 0 2px 3px 0 rgba(0, 0, 0, 0.2);
    outline: unset;
}

form > button {
    justify-self: center;
    padding: 12px 25px;
    border-radius: 0;
    text-transform: uppercase;
    font-weight: 600;
    background: orangered;
    color: white;
    border: none;
    font-size: 14px;
    letter-spacing: -0.1px;
    cursor: pointer;
}

form > button:disabled {
    background: dimgrey;
    cursor: not-allowed;
}
```

With the stylesheet created, you can link it to the `home.html` file using static template tags. Open the `home.html` file:

```
(my_env) $ nano ~/djangopush/templates/home.html
```

Update the `head` section to include a link to the external stylesheet:

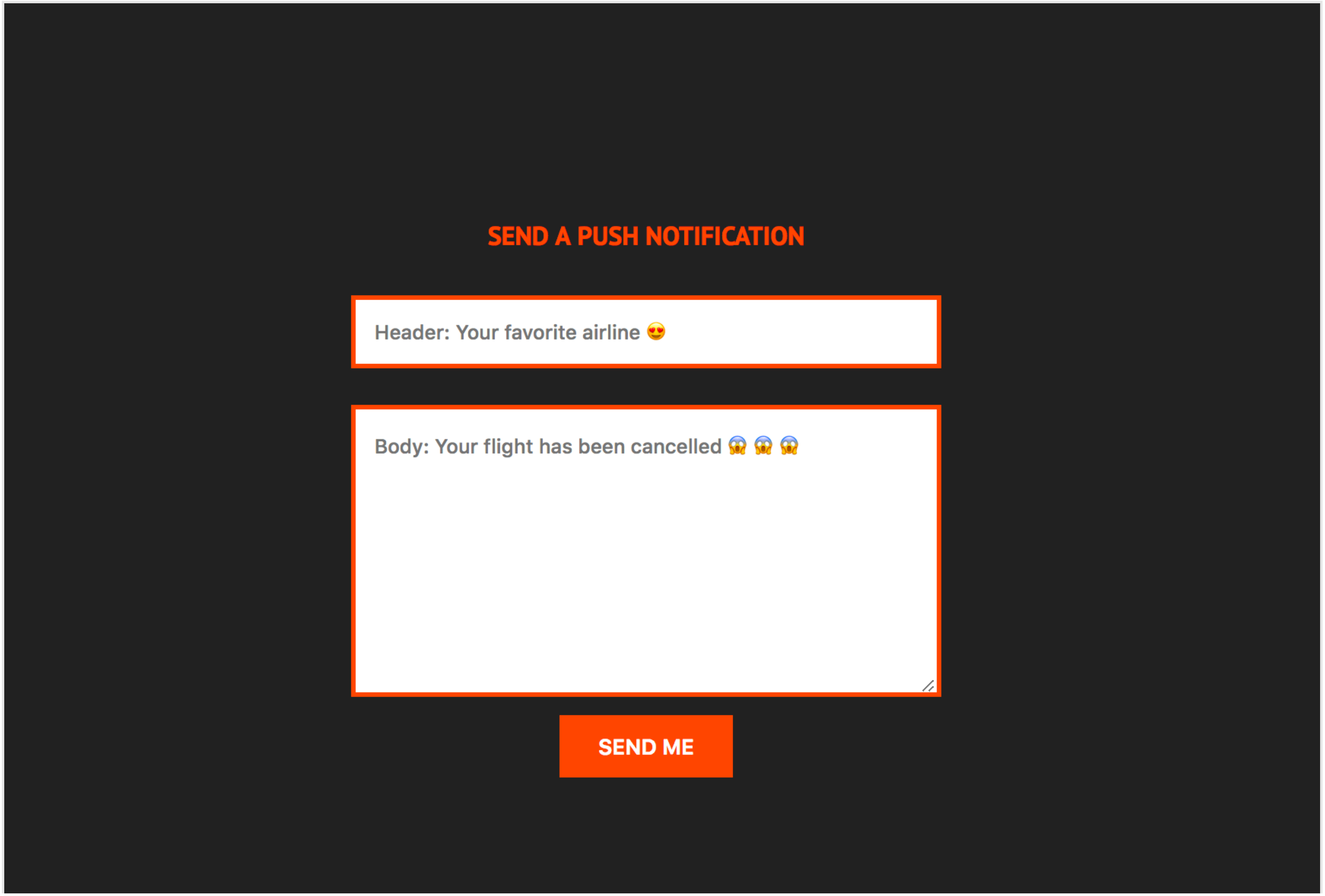
~/djangopush/templates/home.html

```
{% load static %}
<!DOCTYPE html>
<html lang="en">

<head>
    ...
    <link href="{% static '/css/styles.css' %}" rel="stylesheet">
</head>
<body>
    ...
</body>
</html>
```

Make sure that you are in your main project directory and start your server again to inspect your work:

```
(my_env) $ cd ~/djangopush
(my_env) $ python manage.py runserver your_server_ip:8000
```

When you visit `http://your_server_ip:8000`, it should look like this:



Header: Your favorite airline 😍

Body: Your flight has been cancelled 😱 😱 😱

Again, you can kill the server with `CTRL+C`.

Now that you have successfully created the `home.html` page and styled it, you can subscribe users to push notifications whenever they visit the home page.

Step 7 — Registering a Service Worker and Subscribing Users to Push Notifications

Web push notifications can notify users when there are updates to applications they are subscribed to or prompt them to re-engage with applications they have used in the past. They rely on two technologies, the push API and the notifications API. Both technologies rely on the presence of a service worker.

A push is invoked when the server provides information to the service worker and the service worker uses the notifications API to display this information.

We'll subscribe our users to the push and then we'll send the information from the subscription to the server to register them.

In the `static` directory, create a folder called `js`:

```
(my_env) $ mkdir ~/djangopush/static/js
```

Create a file called `registerSw.js`:

```
(my_env) $ nano ~/djangopush/static/js/registerSw.js
```

Add the following code, which checks if service workers are supported on the user's browser before attempting to register a service worker:

~/djangopush/static/js/registerSw.js

```
const registerSw = async () => {
    if ('serviceWorker' in navigator) {
        const reg = await navigator.serviceWorker.register('sw.js');
        initialiseState(reg)

    } else {
        showNotAllowed("You can't send push notifications ☹😢")
    }
};
```

First, the `registerSw` function checks if the browser supports service workers before registering them. After registration, it calls the `initializeState` function with the registration data. If service workers are not supported in the browser, it calls the `showNotAllowed` function.

Next, add the following code below the `registerSw` function to check if a user is eligible to receive push notifications before attempting to subscribe them:

~/djangopush/static/js/registerSw.js

```
...

const initialiseState = (reg) => {
    if (!reg.showNotification) {
        showNotAllowed('Showing notifications isn\'t supported ☹😢');
        return
    }
    if (Notification.permission === 'denied') {
        showNotAllowed('You prevented us from showing notifications ☹🤔');
        return
    }
    if (!'PushManager' in window) {
        showNotAllowed("Push isn't allowed in your browser 🤔");
        return
    }
    subscribe(reg);
}

const showNotAllowed = (message) => {
    const button = document.querySelector('form>button');
    button.innerHTML = `${message}`;
    button.setAttribute('disabled', 'true');
};
```

The `initializeState` function checks the following:

- Whether or not the user has enabled notifications, using the value of `reg.showNotification`.

- Whether or not the user has granted the application permission to display notifications.
- Whether or not the browser supports the `PushManager` API. If any of these checks fail, the `showNotAllowed` function is called and the subscription is aborted.

The `showNotAllowed` function displays a message on the button and disables it if a user is ineligible to receive notifications. It also displays appropriate messages if a user has restricted the application from displaying notifications or if the browser doesn't support push notifications.

Once we ensure that a user is eligible to receive push notifications, the next step is to subscribe them using `pushManager`. Add the following code below the `showNotAllowed` function:

~/djangopush/static/js/registerSw.js

```
...

function urlB64ToUint8Array(base64String) {
    const padding = '='.repeat((4 - base64String.length % 4) % 4);
    const base64 = (base64String + padding)
        .replace(/\-/g, '+')
        .replace(/_/g, '/');

    const rawData = window.atob(base64);
    const outputArray = new Uint8Array(rawData.length);
    const outputData = outputArray.map((output, index) => rawData.charCodeAt(index));

    return outputData;
}
```

```
const subscribe = async (reg) => {
    const subscription = await reg.pushManager.getSubscription();
    if (subscription) {
        sendSubData(subscription);
        return;
    }

    const vapidMeta = document.querySelector('meta[name="vapid-key"]');
    const key = vapidMeta.content;
    const options = {
        userVisibleOnly: true,
        // if key exists, create applicationServerKey property
        ...(key && {applicationServerKey: urlB64ToUint8Array(key)})
    };

    const sub = await reg.pushManager.subscribe(options);
    sendSubData(sub)
};
```

Calling the `pushManager.getSubscription` function returns the data for an active subscription. When an active subscription exists, the `sendSubData` function is called with the subscription info passed in as a parameter.

When no active subscription exists, the VAPID public key, which is Base64 URL-safe encoded, is converted to a Uint8Array using the `urlB64ToUint8Array` function. `pushManager.subscribe` is then called with the VAPID public key and the `userVisible` value as options. You can read more about the available options here.

After successfully subscribing a user, the next step is to send the subscription data to the server. The data will be sent to the `webpush/save_information` endpoint provided by the `django-webpush` package. Add the following code below the `subscribe` function:

~/djangopush/static/js/registerSw.js

```
...

const sendSubData = async (subscription) => {
    const browser = navigator.userAgent.match(/(firefox|msie|chrome|safari|trident)/ig)[0].toLowerCase();
    const data = {
        status_type: 'subscribe',
        subscription: subscription.toJSON(),
        browser: browser,
    };

    const res = await fetch('/webpush/save_information', {
        method: 'POST',
        body: JSON.stringify(data),
        headers: {
            'content-type': 'application/json'
        },
        credentials: "include"
    });

    handleResponse(res);
};

const handleResponse = (res) => {
    console.log(res.status);
};

registerSw();
```

The `save_information` endpoint requires information about the status of the subscription (`subscribe` and `unsubscribe`), the subscription data, and the browser. Finally, we call the `registerSw()` function to begin the process of subscribing the user.

The completed file looks like this:

~/djangopush/static/js/registerSw.js

```
const registerSw = async () => {
    if ('serviceWorker' in navigator) {
        const reg = await navigator.serviceWorker.register('sw.js');
        initialiseState(reg)

    } else {
        showNotAllowed("You can't send push notifications ☹😢")
    }
};

const initialiseState = (reg) => {
    if (!reg.showNotification) {
        showNotAllowed('Showing notifications isn\'t supported ☹😢');
        return
    }
    if (Notification.permission === 'denied') {
        showNotAllowed('You prevented us from showing notifications ☹🙄');
        return
    }
    if (!'PushManager' in window) {
        showNotAllowed("Push isn't allowed in your browser 🤔");

        return
    }
    subscribe(reg);
}

const showNotAllowed = (message) => {
    const button = document.querySelector('form>button');
    button.innerHTML = `${message}`;
    button.setAttribute('disabled', 'true');
};

function urlB64ToUint8Array(base64String) {
    const padding = '='.repeat((4 - base64String.length % 4) % 4);
    const base64 = (base64String + padding)
        .replace(/\-/g, '+')
        .replace(/_/g, '/');

    const rawData = window.atob(base64);
    const outputArray = new Uint8Array(rawData.length);
    const outputData = outputArray.map((output, index) => rawData.charCodeAt(index));

    return outputData;
}

const subscribe = async (reg) => {
    const subscription = await reg.pushManager.getSubscription();
    if (subscription) {
```

```
    if (subscription) {
        sendSubData(subscription);
        return;
    }


    const vapidMeta = document.querySelector('meta[name="vapid-key"]');
    const key = vapidMeta.content;
    const options = {
        userVisibleOnly: true,
        // if key exists, create applicationServerKey property
        ...(key && {applicationServerKey: urlB64ToUint8Array(key)})
    };

    const sub = await reg.pushManager.subscribe(options);
    sendSubData(sub)
};

const sendSubData = async (subscription) => {
    const browser = navigator.userAgent.match(/(firefox|msie|chrome|safari|trident)/ig)[0].toLowerCase();
    const data = {
        status_type: 'subscribe',
        subscription: subscription.toJSON(),
        browser: browser,
    };

    const res = await fetch('/webpush/save_information', {
        method: 'POST',
        body: JSON.stringify(data),
        headers: {
            'content-type': 'application/json'
        }
        },
        credentials: "include"
    });


    handleResponse(res);
};

const handleResponse = (res) => {
    console.log(res.status);
};

registerSw();
```

Next, add a `script` tag for the `registerSw.js` file in `home.html`. Open the file:

```
(my_env) $ nano ~/djangopush/templates/home.html
```

Add the `script` tag before the closing tag of the `body` element:

~/djangopush/templates/home.html

```
{% load static %}
<!DOCTYPE html>
<html lang="en">

<head>
   ...
```

```
</head>
<body>
   ...
   <script src="{% static '/js/registerSw.js' %}"></script>
</body>
</html>
```

Because a service worker doesn't yet exist, if you left your application running or tried to start it again, you would see an error message. Let's fix this by creating a service worker.

Step 8 — Creating a Service Worker

To display a push notification, you'll need an active service worker installed on your application's home page. We'll create a service worker that listens for `push` events and displays the messages when ready.

Because we want the scope of the service worker to be the entire domain, we will need to install it in the application's root. You can read more about the process in this article outlining how to register a service worker. Our approach will be to create a `sw.js` file in the `templates` folder, which we will then register as a view.

Create the file:

```
(my_env) $ nano ~/djangopush/templates/sw.js
```

Add the following code, which tells the service worker to listen for push events:

~/djangopush/templates/sw.js

```
// Register event listener for the 'push' event.
self.addEventListener('push', function (event) {
    // Retrieve the textual payload from event.data (a PushMessageData object).
    // Other formats are supported (ArrayBuffer, Blob, JSON), check out the documentation
    // on https://developer.mozilla.org/en-US/docs/Web/API/PushMessageData.
    const eventInfo = event.data.text();
    const data = JSON.parse(eventInfo);
    const head = data.head || 'New Notification 🕺🕺';
    const body = data.body || 'This is default content. Your notification didn\'t have one 🙄🙄';

    // Keep the service worker alive until the notification is created.
    event.waitUntil(

        self.registration.showNotification(head, {
            body: body,
            icon: 'https://i.imgur.com/MZM3K5w.png'
        })
    );
});
```

The service worker listens for a push event. In the callback function, the `event` data is converted to text. We use default `title` and `body` strings if the event data doesn't have them. The `showNotification` function takes the notification title, the header of the notification to be displayed, and an options object as parameters. The options object contains several properties to configure the visual options of a notification.

For your service worker to work for the entirety of your domain, you will need to install it in the root of the application. We'll use `TemplateView` to allow the service worker access to the whole domain.

Open the `urls.py` file:

```
(my_env) $ nano ~/djangopush/djangopush/urls.py
```

Add a new import statement and path in the `urlpatterns` list to create a class-based view:

~/djangopush/djangopush/urls.py

```
...
from django.views.generic import TemplateView

urlpatterns = [
                  ...,
                  path('sw.js', TemplateView.as_view(template_name='sw.js', content_type='application/x-javascript'))
              ] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

Class-based views like `TemplateView` allow you to create flexible, reusable views. In this case, the `TemplateView.as_view` method creates a path for the service worker by passing the recently created service worker as a template and `application/x-javascript` as the `content_type` of the template.

You have now created a service worker and registered it as a route. Next, you'll set up the form on the home page to send push notifications.

Step 9 — Sending Push Notifications

Using the form on the home page, users should be able to send push notifications while your server is running. You can also send push notifications using any RESTful service like Postman. When the user sends push notifications from the form on the home page, the data will include a `head` and `body`, as well as the `id` of the receiving user. The data should be structured in the following manner:

```
{
    head: "Title of the notification",
    body: "Notification body",
    id: "User's id"
}
```

To listen for the `submit` event of the form and send the data entered by the user to the server, we will create a file called `site.js` in the `~/djangopush/static/js` directory.

Open the file:

```
(my_env) $ nano ~/djangopush/static/js/site.js
```

First, add a `submit` event listener to the form that will enable you to get the values of the form inputs and the user id stored in the `meta` tag of your template:

~/djangopush/static/js/site.js

```
const pushForm = document.getElementById('send-push__form');
const errorMsg = document.querySelector('.error');

pushForm.addEventListener('submit', async function (e) {
    e.preventDefault();
    const input = this[0];
    const textarea = this[1];
    const button = this[2];
    errorMsg.innerText = '';

    const head = input.value;
    const body = textarea.value;
    const meta = document.querySelector('meta[name="user_id"]');
    const id = meta ? meta.content : null;
    ...
    // TODO: make an AJAX request to send notification
});
```

The `pushForm` function gets the `input`, `textarea`, and `button` inside the form. It also gets the information from the `meta` tag, including the name attribute `user_id` and the user's id stored in the `content` attribute of the tag. With this information, it can send a POST request to the `/send_push` endpoint on the server.

To send requests to the server, we'll use the native Fetch API. We're using Fetch here because it is supported by most browsers and doesn't require external libraries to function. Below the code you've added, update the `pushForm` function to include the code for sending AJAX requests:

~/djangopush/static/js/site.js

```
const pushForm = document.getElementById('send-push__form');
const errorMsg = document.querySelector('.error');

pushForm.addEventListener('submit', async function (e) {
     ...
    const id = meta ? meta.content : null;

     if (head && body && id) {
        button.innerText = 'Sending...';
        button.disabled = true;

        const res = await fetch('/send_push', {
            method: 'POST',
            body: JSON.stringify({head, body, id}),
            headers: {
                'content-type': 'application/json'
            }
        });
        if (res.status === 200) {
            button.innerText = 'Send another 😃!';
            button.disabled = false;
            input.value = '';
            textarea.value = '';
        } else {
            errorMsg.innerText = res.message;
            button.innerText = 'Something broke 😢..  Try again?';
            button.disabled = false;
        }
    }
    else {
        let error;
        if (!head || !body){
            error = 'Please ensure you complete the form 🙏🏿'
        }
        else if (!id){
            error = "Are you sure you're logged in? 🙄. Make sure! □□"
        }
        errorMsg.innerText = error;
    }
});
```

If the three required parameters `head`, `body`, and `id` are present, we send the request and disable the submit button temporarily.

The completed file looks like this:

~/djangopush/static/js/site.js

```
const pushForm = document.getElementById('send-push__form');
const errorMsg = document.querySelector('.error');

pushForm.addEventListener('submit', async function (e) {
    e.preventDefault();
    const input = this[0];
    const textarea = this[1];
    const button = this[2];
    errorMsg.innerText = '';

    const head = input.value;
    const body = textarea.value;
    const meta = document.querySelector('meta[name="user_id"]');
    const id = meta ? meta.content : null;

    if (head && body && id) {
        button.innerText = 'Sending...';
        button.disabled = true;

        const res = await fetch('/send_push', {
            method: 'POST',
            body: JSON.stringify({head, body, id}),
            headers: {
                'content-type': 'application/json'
            }
        });
        if (res.status === 200) {
            button.innerText = 'Send another 😃!';
            button.disabled = false;
            input.value = '';
            textarea.value = '';
        } else {
            errorMsg.innerText = res.message;
            button.innerText = 'Something broke ☹..  Try again?';
            button.disabled = false;
        }
    }
    else {
        let error;
        if (!head || !body){
            error = 'Please ensure you complete the form 🙇🏽'
        }
        else if (!id){
            error = "Are you sure you're logged in? 🤔. Make sure! □□"
        }
        errorMsg.innerText = error;
    }
});
```

Finally, add the `site.js` file to `home.html`:

```
(my_env) $ nano ~/djangopush/templates/home.html
```

Add the `script` tag:

~/djangopush/templates/home.html

```
{% load static %}
<!DOCTYPE html>
<html lang="en">

<head>
   ...
</head>
<body>
   ...
   <script src="{% static '/js/site.js' %}"></script>
</body>
</html>
```

At this point, if you left your application running or tried to start it again, you would see an error, since service workers can only function in secure domains or on `localhost`. In the next step we'll use ngrok to create a secure tunnel to our web server.

Step 10 — Creating a Secure Tunnel to Test the Application

Service workers require secure connections to function on any site except `localhost` since they can allow connections to be hijacked and responses to be filtered and fabricated. For this reason, we'll create a secure tunnel for our server with ngrok.

Open a second terminal window and ensure you're in your home directory:

```
$ cd ~
```

If you started with a clean 18.04 server in the prerequisites, then you will need to install `unzip`:

```
$ sudo apt update && sudo apt install unzip
```

Download ngrok:

```
$ wget https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip
$ unzip ngrok-stable-linux-amd64.zip
```

Move `ngrok` to `/usr/local/bin`, so that you will have access to the `ngrok` command from the terminal:

```
$ sudo mv ngrok /usr/local/bin
```

In your first terminal window, make sure that you are in your project directory and start your server:

```
(my_env) $ cd ~/djangopush
(my_env) $ python manage.py runserver your_server_ip:8000
```

You will need to do this before creating a secure tunnel for your application.

In your second terminal window, navigate to your project folder, and activate your virtual environment:

```
$ cd ~/djangopush
$ source my_env/bin/activate
```

Create the secure tunnel to your application:

```
(my_env) $ ngrok http your_server_ip:8000
```

You will see the following output, which includes information about your secure ngrok URL:

```
Output
ngrok by @inconshreveable

Session Status                online
Session Expires               7 hours, 59 minutes
Version                       2.2.8
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://ngrok_secure_url -> 203.0.113.0:8000
Forwarding                    https://ngrok_secure_url -> 203.0.113.0:8000

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

Copy the `ngrok_secure_url` from the console output. You will need to add it to the list of `ALLOWED_HOSTS` in your `settings.py` file.

Open another terminal window, navigate to your project folder, and activate your virtual environment:

```
$ cd ~/djangopush
$ source my_env/bin/activate
```

Open the `settings.py` file:

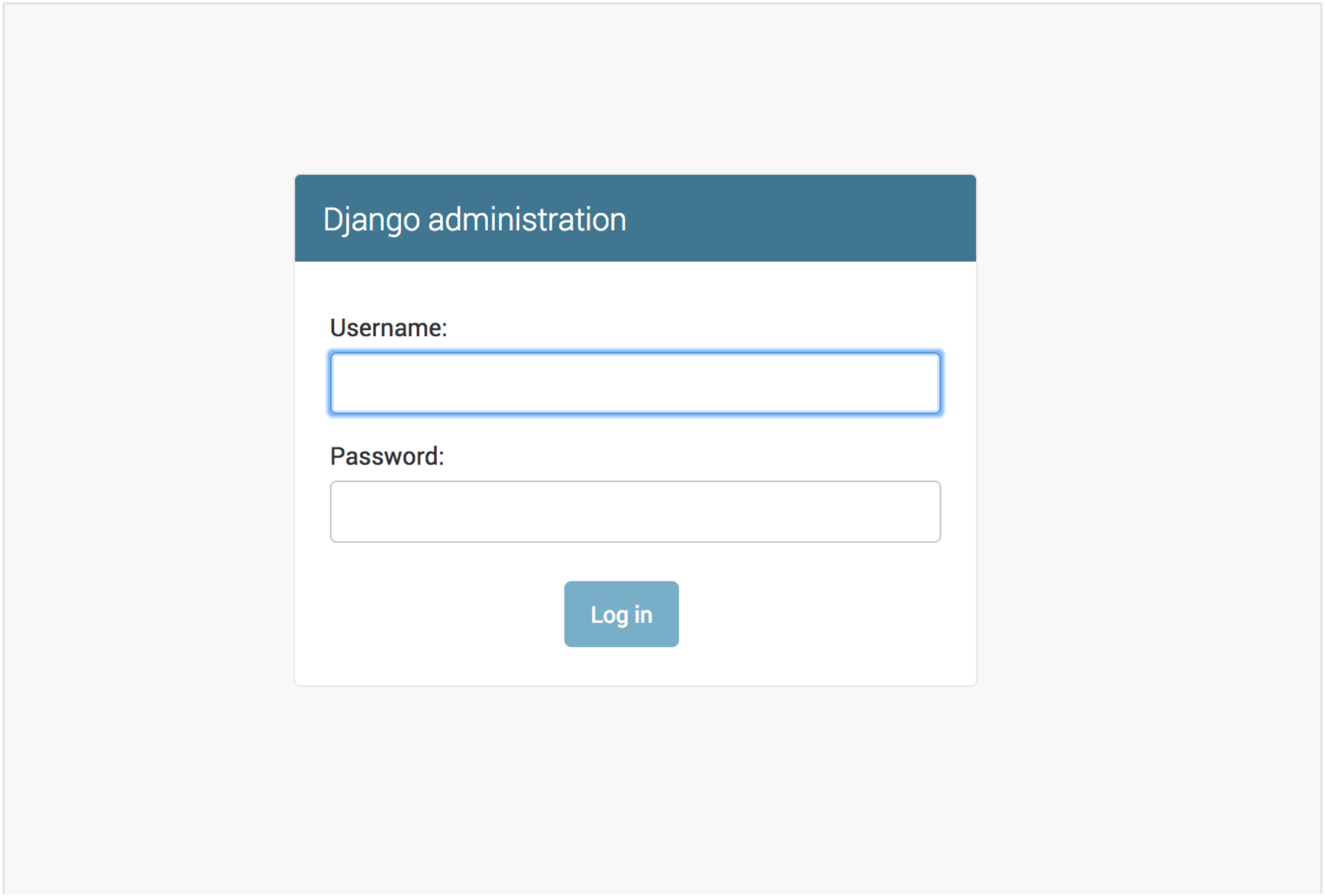
```
(my_env) $ nano ~/djangopush/djangopush/settings.py
```

Update the list of `ALLOWED_HOSTS` with the ngrok secure tunnel:

~/djangopush/djangopush/settings.py

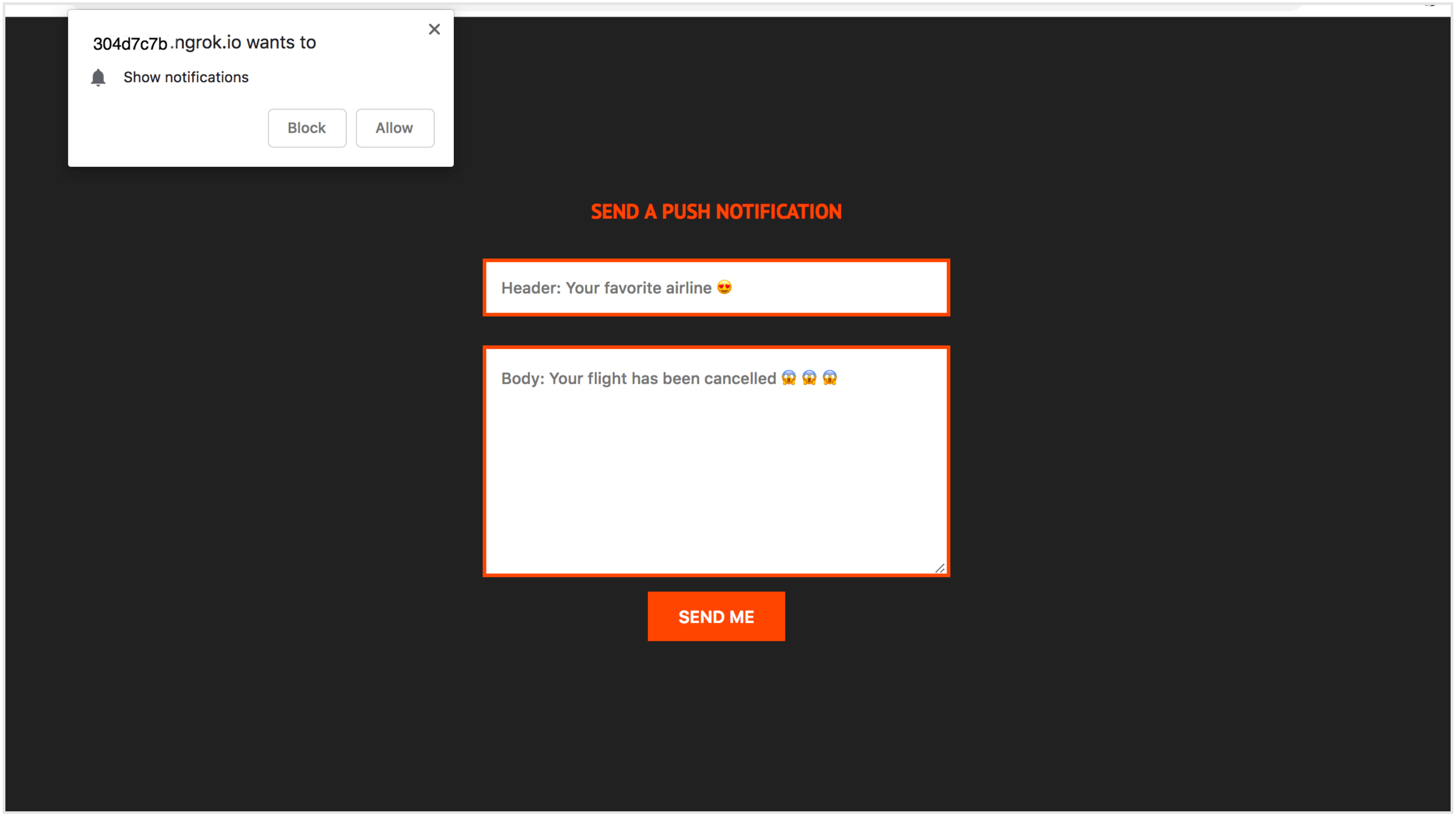
```
...

ALLOWED_HOSTS = ['your_server_ip', 'ngrok_secure_url']
...
```

Navigate to the secure admin page to log in: `https://ngrok_secure_url/admin/`. You will see a screen that looks like this:


Django administration
Username:
Password:
Log in

Enter your Django admin user information on this screen. This should be the same information you entered when you logged into the admin interface in the prerequisite steps. You are now ready to send push notifications.

Visit `https://ngrok_secure_url` in your browser. You will see a prompt asking for permission to display notifications. Click the **Allow** button to let your browser display push notifications:


304d7c7b.ngrok.io wants to

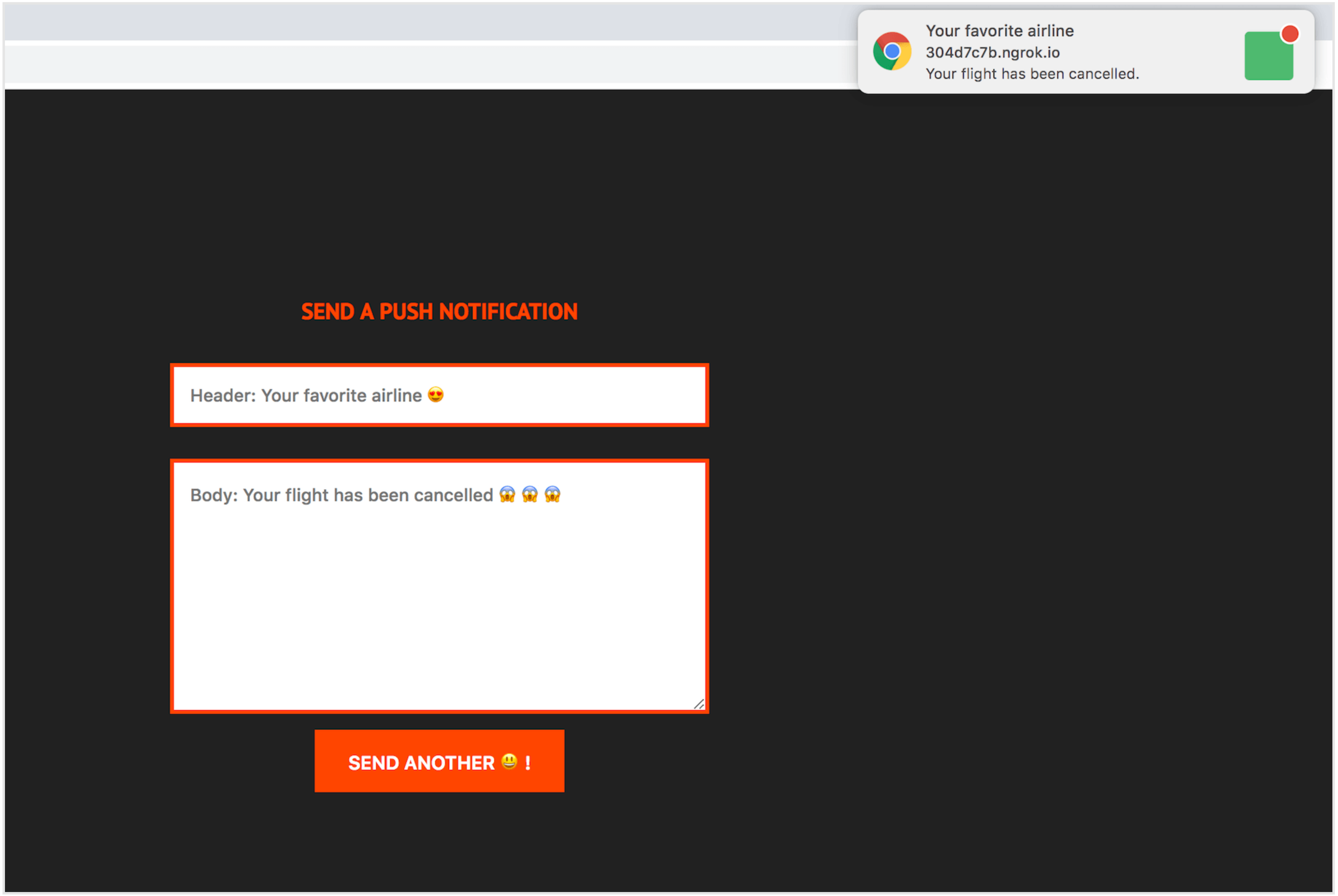
Show notifications

SEND A PUSH NOTIFICATION

Header: Your favorite airline 😍

Body: Your flight has been cancelled 😱 😱 😱

Submitting a filled form will display a notification similar to this:



Your favorite airline
304d7c7b.ngrok.io
Your flight has been cancelled.

SEND A PUSH NOTIFICATION

Header: Your favorite airline 😍

Body: Your flight has been cancelled 😱 😱 😱

Note: Be sure that your server is running before attempting to send notifications.

If you received notifications then your application is working as expected.

You have created a web application that triggers push notifications on the server and, with the help of service workers, receives and displays notifications. You also went through the steps of obtaining the VAPID keys that are required to send push notifications from an application server.

Conclusion

In this tutorial, you've learned how to subscribe users to push notifications, install service workers, and display push notifications using the notifications API.

You can go even further by configuring the notifications to open specific areas of your application when clicked. The source code for this tutorial can be found here.

By Richard Umoffia

Editor: Kathleen Juell

Was this helpful? Yes No

15

Report an issue

Related

CHEATSHEET

How To Manage Sorted Sets in Redis

In Redis, sorted sets are a data type similar to sets in that both are non repeating groups of strings. The...

TUTORIAL

How To Build an Inspirational Quote Application Using AdonisJs and MySQL

In this tutorial, you'll build an application with...

TUTORIAL

How To Migrate Redis Data to a DigitalOcean Managed Database

Redis provides a number of methods one can use t...

TUTORIAL

How To Acquire a Let's Encrypt Certificate Using Ansible on Ubuntu 18.04

Using a configuration management tool such as...

Still looking for an answer?

Ask a question

Search for more help

15 Comments

Leave a comment...

Sign In to Comment

mahalokesh55 *November 28, 2018*

0

Clients can apply to exchange their EPFO claims online with the assistance of this entrance. A part has an alternative to present https://epfindia.pro his case either through his present manager or the past one

macdonjo *February 22, 2019*

0

I got an error:

```
POST http://localhost:8000/webpush/save_information 400 (Bad Request)
```

0 richyafro *February 24, 2019*

The endpoint here should be `http://<your_server_ip>:8000/webpush/save_information` or `http://<ngrok_url>/webpush/save_information`

0 iananich *March 3, 2019*

Is there any way to let unauthenticated users subscribe for personal notifications? Or at least group notifications.

0 richyafro *March 5, 2019*

Yes, you can create a group and send notifications to the group

0 nikhilkumar4 *March 5, 2019*

I am getting this error!

Please Help!

Internal Server Error: /send_push

0 richyafro *March 5, 2019*

Please ensure that you've logged in to Django admin to authenticate your user before making requests nikhilkumar4 *March 6, 2019*

1 i did the authentication but still its showing the same error?

is there any other reason?

nikhilkumar4 *March 7, 2019*

1 Actually its Working , i am able to send the push notification a couple of times , but after few attempts its showing the same error. (Internal Server Error: /send_push)

Avrl *March 21, 2019*

0 got the same error, actually the error is causing due to the missing of trailing slash in (/send*push) change /send*push to /send_push/

Avrl *March 21, 2019*

0 I've got no errors, neitherin my developer console nor in my python console, then also not able to send the notification. i've done exactly what you've told

matbwork *March 28, 2019*

0 By follwing this tutorial I was able to set notifications Thks.

(it works also with localhost on firefox) :-)

0

richyafro *July 22, 2019*

Happy this helped you 😊

0

arp1051 *July 22, 2019*

Hi dear Richard Umoffia

excuse me would you please tell me how to configuring the notifications to open specific areas of your application when clicked?

I add the following code to the end of the sw.js file but it doesn't work:

```
self.addEventListener('notificationclick', function(event) {
  console.log('[Service Worker] Notification click Received.');

  event.waitUntil(
    clients.openWindow('https://example.com')
  );
});
```

would you please help me?

0

richyafro *July 22, 2019*

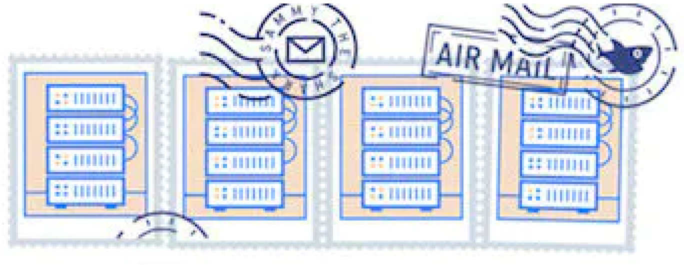
Hi, please ensure you're working on your localhost or a secure domain. Service workers don't work over http. If you're already doing that, I'll suggest you go through this codelab to help you.

BECOME A CONTRIBUTOR

You get paid; we donate to tech nonprofits.

CONNECT WITH OTHER DEVELOPERS

Find a DigitalOcean Meetup near you.



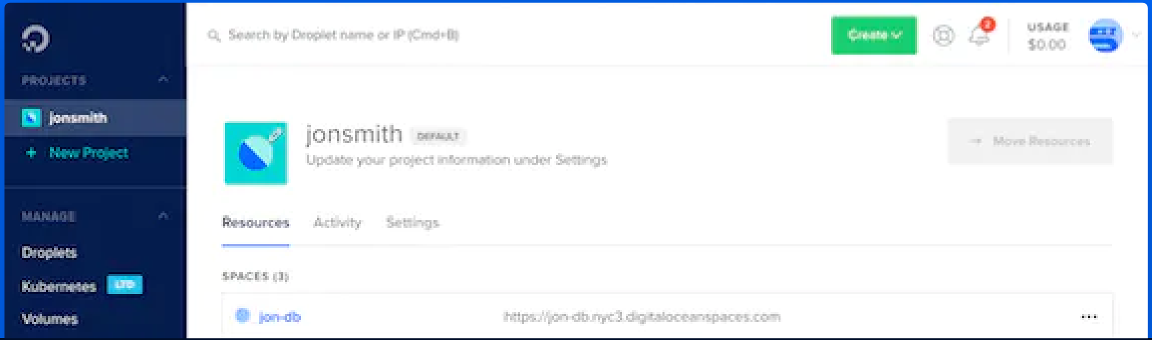
GET OUR BIWEEKLY NEWSLETTER

Sign up for Infrastructure as a Newsletter.

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're running one virtual machine or ten thousand.

Learn More



Company

About
Leadership
Blog
Careers
Partners
Referral Program
Press
Legal & Security

Products

Products Overview
Pricing
Droplets
Kubernetes
Managed Databases
Spaces
Marketplace
Load Balancers
Block Storage
Tools & Integrations
API
Documentation
Release Notes

Community

Tutorials
Q&A
Tools and Integrations
Tags
Product Ideas
Meetups
Write for DOnations
Droplets for Demos
Hatch Startup Program
Shop Swag
Research Program
Currents Research
Open Source

Contact

Support
Sales
Report Abuse
System Status