Serve HTML from a Worker

In Chapter 6, ***Build a Static Website with Pages***, and Chapter 9, ***Upload and Store Files on R2***, you used Cloudflare Pages to build and deploy your front end. However, it's possible to serve HTML from a Worker to act as your front end using Static Assets Workers.[33] In the same way you used Cloudflare Pages to build a full-stack application using Next.js, you can do the same with Static Assets Workers, too. Don't be fooled by the name: building dynamic websites is absolutely possible.[34]

You aren't going to use Next.js here, to keep things a bit simpler. You'll need the HTML itself, which I'm not going to include directly here as it's too long, but it's available to download alongside all the code in this book. You'll find it named `13-app.html`. This is what it would look like once rendered:

The initial page load contains an input box where the user enters their name, and once they submit, they're presented with this screen. I'll go over the JavaScript needed to interact with the back end for the chat, but we'll tackle that at the end.

Once you've downloaded the source files provided alongside the book, retrieve the HTML from `13-app.html` and store it in a new file under `public/index.html`.

As Workers don't have access to the file system during runtime, you can't just load the HTML from the file and serve it. Instead, you can configure the Worker to be able to serve static assets from a specific folder by editing the `wrangler.toml` file:

```
compatibility_date = "2024-09-25"
assets = { directory = "./public/" }
```

First, you need to update the compatibility date to a more recent one if it's not set to the date above. The compatibility date effectively determines what version of the Workers runtime you use, and as Static Assets Workers are very new at the time of writing, you may need to use a newer version of the runtime than what was set when the project was created.

Second, you define a directory where your static assets will be served from. When a request comes into a Worker with **assets** defined, it first checks to see if the path requested matches a static asset; if it does, that'll be returned. If no matches are found, it'll execute the **fetch** method of your Worker. More information on routing can be found in the docs.[35]

You don't need to for this application, but you can optionally set a binding that allows you to programmatically access assets:

```
assets = { directory = "./public/", binding = "ASSETS"
}
```

You can then dynamically access assets from your Worker if you need to, using **env.ASSETS.fetch(request)**. You could use this to return a different image from the assets directory for the same path, based on HTTP headers or other data points. Maybe you have different variations of your logo you want to serve based on the country of origin, for example.

The Worker can now return static assets, but for the API, we'll need to make use of the **fetch** function of the Worker. You'll make use of **itty-router** once more to handle routing; add it to your project's dependencies by running:

```
$ npm install --save itty-router
```

Let's create the basic code needed to serve routes next. Import the necessary dependencies from `itty-router`, create the Router and add its one route to render HTML in `src/index.ts` below the line that imports the `DurableObject` class:

13-router.ts

```
import {
  error,
  Router,
} from 'itty-router'

const router = Router();

router
  .all('*', () => error(404))
```

Lastly, update the `fetch` method like so:

13-update-fetch.ts

```
export default {
  async fetch(
    request: Request,
    env: Env,
    ctx: ExecutionContext
  ): Promise<Response> {
    return router.fetch(request, env);
  },
};
```

The two exports, `Env` and `MyDurableObject`, are technically superfluous for now—but don't remove them yet, we'll do so later in the chapter.

You can validate the HTML is being rendered by running your Worker with `npm run dev` and accessing the URL given by Wrangler in the output. You can type your name in the box and hit the button, and the page should update to say it's connecting to chat.

There's no back end logic yet to handle chats, and if you look in your terminal where your Worker is running, you'll see a 404 returned for `/api/chat`. When a new chat is created, or someone joins an existing chat, that's the endpoint that is hit, so you'll implement that next.