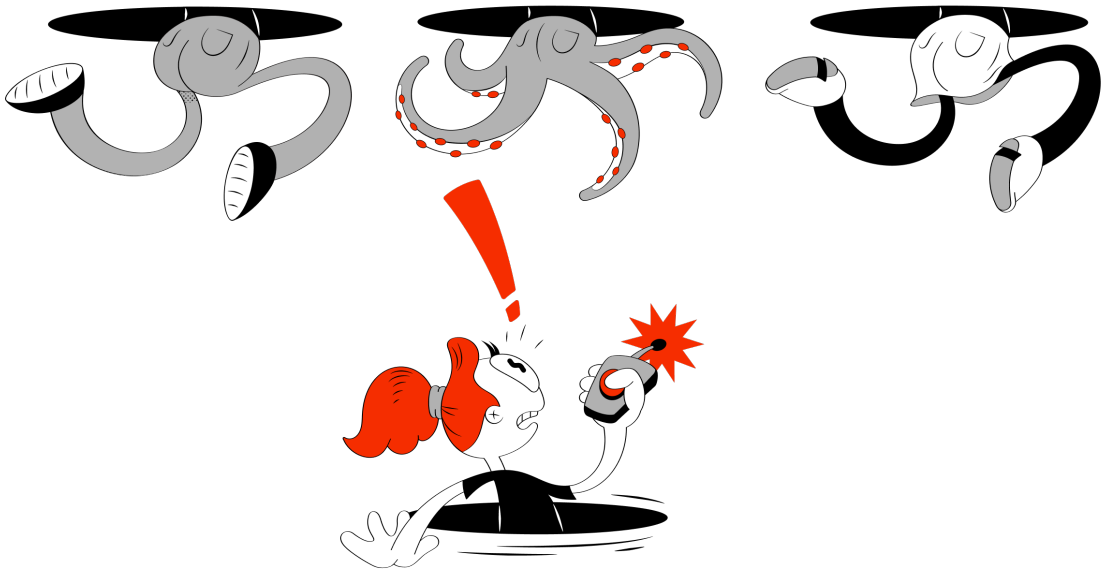
Using traefik for routing paths to web apps

Setup a reverse proxy to simplify routing

Written by Seth Corker for Benevolent Bytes Blog



If you're working with a single web app that handles routing, that's pretty straight-forward, but what happens when working with multiple web apps? It can be a pain to get these running easily and setup routing correctly. Let's take a use case I've been experimenting with lately when looking at building a website comprised of multiple web apps, you might hear this referred to as micro frontends. The first step is to ensure my various web apps are all reachable on the same domain from different paths. Let's take a look what we're trying to achieve and then some tools we'll use to do it.

I want each app to be on a different path

At its core, each app should own its path so.

- The Home app could own / and serve a single page

- The Shop app could own `/shop` and any subpaths with that prefix, e.g. `/shop/2031/info`
- The Showcase could own `/showcase`, you get the idea.

Assuming we try to run this setup locally, you'll probably run each app on its own port. Not ideal. It means each app has to specify a port that it won't conflict with other apps. This works with 3, what about if it grows to 10 different apps? The other problem we run into is that the URL changes every time, you'll go from `localhost:8000` to `localhost:8080` to wherever else. We want to have these apps on the same domain and feel part of a cohesive experience. This is where a reverse proxy will help.

What's a reverse proxy?

A server that sits in front of our web apps and forwards the requests to them. It's precisely what we need to make sure when I visit `localhost/shop` the Shop web app responds to the request. From a user's perspective, this creates a cohesive experience and from a development perspective, each app can be isolated and use whatever technology we like.

So, now we know we want a reverse proxy, what options do we have?

Nginx

Nginx is much more than just a reverse proxy, but for our purposes it would work pretty well. It's flexible, fast and has been my go-to tool for doing this job for a long time. The biggest issue I have with Nginx is that to get what we want, we still need to spin up our servers with unique ports, so they don't conflict. If this isn't a problem you care about, then Nginx is a great solution. It's a problem for me though, so I was on the search for a better way and for local development, Traefik seems to fit the bill.

Traefik

Traefik proxy describes itself like this:

> *Traefik is an open-source Edge Router that makes publishing your services a fun and easy experience. It receives requests on behalf of your system and finds out which components are responsible for handling them.*

Why I've had success with it comes down to ease of use. The documentation is wonderful and easy to follow. My favourite feature is the integration with Docker, it allows you to run all docker images and add some labels for configuration so Traefik can find them. This solves the port issue I described earlier. This solution allows for all your apps to run without explicit exposing ports, you just let Traefik know what port it should forward requests to and when it should send those requests to a particular app. An easy reverse proxy like Traefik is a great solution for micro frontends or even just easier frontend and backend setup for single-page web apps and APIs.

How do we set up Traefik with Docker locally?

Docker compose

I'm going to use a single docker compose file to demonstrate how to set up three web apps, Home, Shop and Showcase. This is to simplify the example and focus on Traefik and configuration. I've used it with multiple directories too where each project runs on a different tech stack, uses different ports etc. but all use the same docker network, so they can be picked up by Traefik.

Our first service

Setting up the reverse proxy is much easier than Nginx. It can all be done within the docker-compose.yml file with a few lines. As long as Traefik is running, it will pick up new services that we add to Docker.

```
services:
  reverse-proxy:
    image: traefik:v2.4
    command: --api.insecure=true --providers.docker
    ports:
      - "80:80"
      - "8080:8080"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    labels:
      - traefik.enable=false
```

The web app config

The web apps themselves will all use the same Deno web server source, but run on different containers with slightly different config. Mainly, the `APP_NAME` environment variable will be passed in to each app, so we can show it in the title to distinguish between them. In a real scenario, each web app would probably do something a little more involved than just spitting out some basic HTML.

The Deno web server container will look like this:

```
services:
  home:
    restart: always
    image: denoland/deno:1.11.2
    command: run --allow-net --allow-read --allow-env /
app/server.ts
    labels:
      - traefik.http.routers.home.rule=Host(`deno-
mfe.localhost`)
      -
traefik.http.services.home.loadbalancer.server.port=8080
    environment:
      - APP_NAME=Home
    volumes:
      - ./app:/app
```

The key configuration that Traefik uses is labels.

- `traefik.http.routers.home.rule=Host(`deno-mfe.localhost`)` defines the domain we want to use, in this case I chose `deno-mfe.localhost` so navigating to it will be easy.
- `traefik.http.services.home.loadbalancer.server.port=8080` is what tells Traefik that it should send requests to port `8080` because that's what Deno is listening to.

There are lots more you can do, but this will do just fine for our purposes. To support different paths that point to each app we can use the same rule but add a `PathPrefix` like this `traefik.http.routers.shop.rule=Host(`deno-`

`mfe.localhost`) && PathPrefix(`/shop`)`. This label tells Traefik to match a request if it has the same host and the `/shop` prefix.

We'll add our remaining apps, the docker config now looks like this:

```
version: "3.8"

services:
  reverse-proxy:
    image: traefik:v2.4
    command: --api.insecure=true --providers.docker
    ports:
      - "80:80"
      - "8080:8080"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    labels:
      - traefik.enable=false

  home:
    restart: always
    image: denoland/deno:1.11.2
    command: run --allow-net --allow-read --allow-env /
app/server.ts
    labels:
      - traefik.http.routers.home.rule=Host(`deno-
mfe.localhost`)
      -
traefik.http.services.home.loadbalancer.server.port=8080
    environment:
      - APP_NAME=Home
    volumes:
      - ./app:/app

  shop:
    restart: always
    image: denoland/deno:1.11.2
```

```
    command: run --allow-net --allow-read --allow-env /
app/server.ts
    labels:
      - traefik.http.routers.shop.rule=Host(`deno-
mfe.localhost`) && PathPrefix(`/shop`)
      -
traefik.http.services.shop.loadbalancer.server.port=8080
    environment:
      - APP_NAME=Shop
    volumes:
      - ./app:/app

  showcase:
    restart: always
    image: denoland/deno:1.11.2
    command: run --allow-net --allow-read --allow-env /
app/server.ts
    labels:
      - traefik.http.routers.showcase.rule=Host(`deno-
mfe.localhost`) && PathPrefix(`/showcase`)
      -
traefik.http.services.showcase.loadbalancer.server.port=
8080
    environment:
      - APP_NAME=Showcase
    volumes:
      - ./app:/app
    expose:
      - 8080
```

The Deno web server

Deno is a runtime for JavaScript much like Node.js, the reason I chose it for this example is that everything fits neatly into 24 lines of code. The web server isn't crucial, you could just as easily have set up a Flask and React app or anything you like.

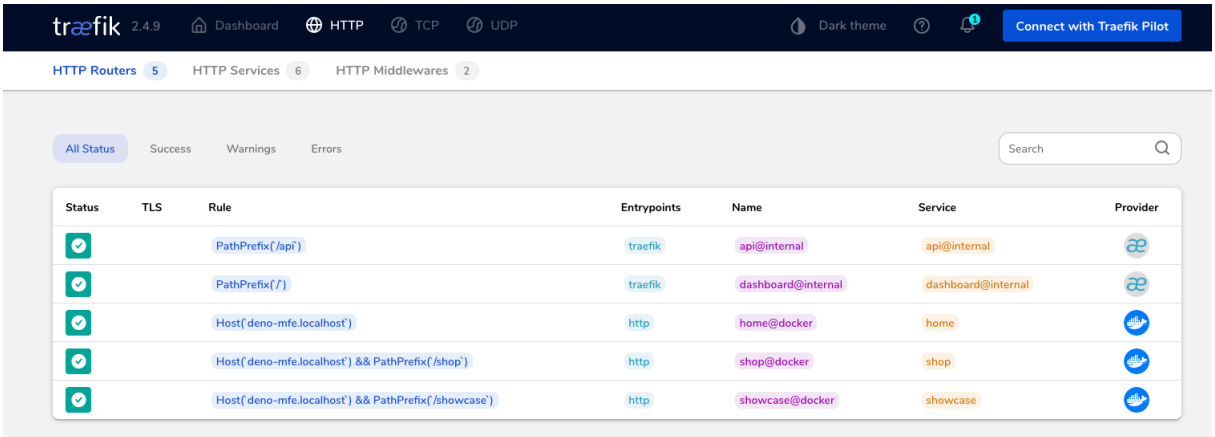
This example sets up a simple HTTP server that responds with some HTML. We're using the `APP_NAME` environment variable to distinguish between apps when we come to testing.

```
import { serve } from "https://deno.land/std@0.99.0/
http/server.ts";

const port = parseInt(`${Deno.env.get("PORT")}`) ||
8080;
const appName = Deno.env.get("APP_NAME") || "Unknown";

const s = serve({ port });

for await (const req of s) {
  const headers = new Headers();
  headers.set("Content-Type", "text/html");
  req.respond({
    body: `
  <html>
    <h1>Hello from ${appName}</h1>
    <ul>
      <li><a href="/">Home</a>
      <li><a href="/shop">Shop</a>
      <li><a href="/showcase">Showcase</a>
    </ul>
  </html>
  `,
    headers,
  });
}
```

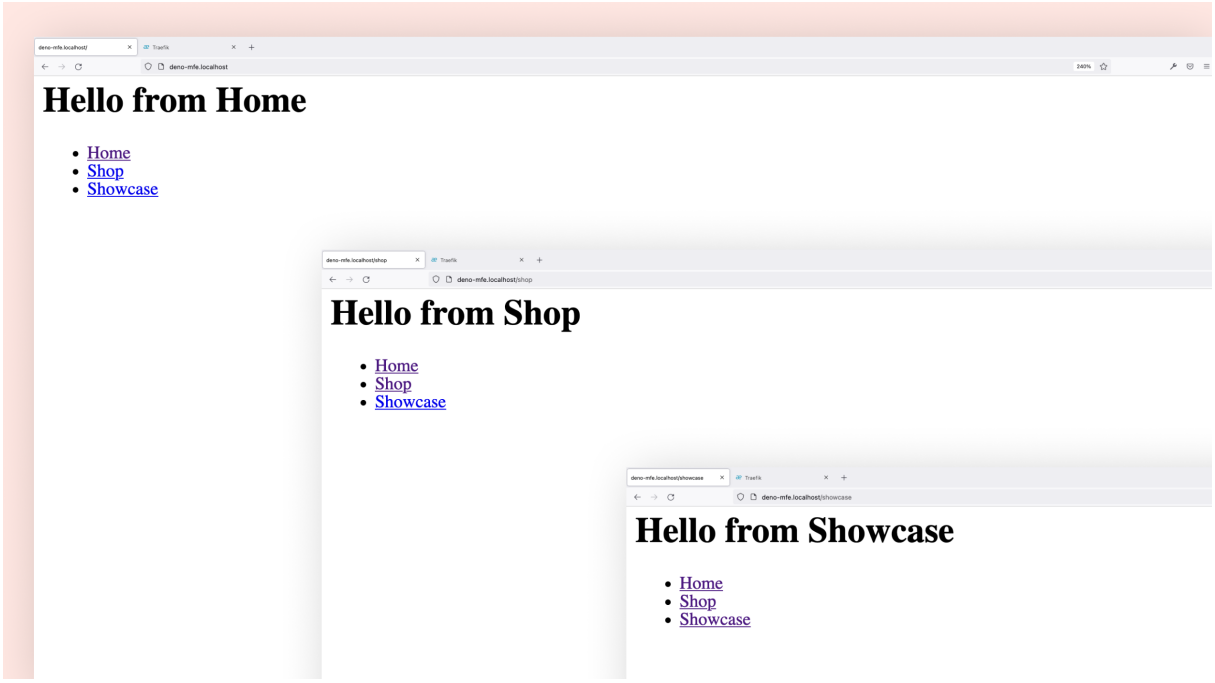
What does it look like?

With our setup complete, we can build and run our stack with `docker compose up --build`.

You'll notice we have no ports exposed from Docker's perspective, Traefik will handle that for us. Now it's up, let's take a look at the dashboard. We can navigate to http://localhost:8080 to see what our proxy is doing. You'll see the configuration that's set up and if you start up any other services (providing they are on the same docker network), Traefik will pick them up and start routing to them.



Finally, we can navigate to the host we specified – `http://deno-mfe.localhost` to see a page served by our Home app. Navigating to `/shop` and `/showcase` route the requests to the Shop app and Showcase app respectively.



There you have it, a more friendly way of routing requests to different web servers without messing about with ports.

What I like about Traefik and Docker

Working locally with multiple services can be challenging to set up, Docker and Docker Compose do an impressive job of equalising environments and making spinning up those services more straightforward but, as you deal with more services; it can be difficult to get all these set up just right and talking to each other. This problem is exacerbated when you're dealing with other teams creating their own services that by default use the same ports. You end up creating overrides for your docker-compose files for development and carefully assigning ports to each service, updating .env files and making sure everything works as expected. Traefik makes that much simpler.

With multiple projects, each service just has to specify the path they own and the port that the service listens on. Then run Traefik with a simple config, and it will discover the services. With an external network, teams can create their own services and during local development use Traefik to route requests in an easier to reason about manner. Environment variables referencing ports don't need to be updated, there's no additional overrides, and everything is much easier. It also works for different url requirements, I've shown an app per path approach but you just as easily give each service a more friendly url on subdomains or split it by

query string, headers and more. It's easy to get started and pretty customisable. If you're dealing with configuring ports and juggling lots of small services, Traefik might just make it easier to reason about those services. Give it a go.