The Pragmatic Programmers

Serverless Apps on Cloudflare

Build Solutions, Not Infrastructure

Ashley Peacock

edited by Michael Swaine

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

CHAPTER 5

Worker To Worker Communication

Your API is almost complete, but there is one thing we haven't covered yet: worker-to-worker communication.

In typical software architectures, when you're not dealing with a monolith, you'll likely need your services to talk to each other through API calls. With serverless architectures, it's not smart to squeeze everything into one Worker, so truly monolithic apps are pretty rare.

If you've got multiple APIs contributing to your app, I'd suggest using multiple workers. Traditionally, these calls between services happen over HTTP, and whether they're private or public, they have to deal with network-related issues like latency and failures.

But here's the cool part with Cloudflare: worker-to-worker communication is a breeze. Just like how you added a database as a dependency to the Photo API using a binding, you can do the same with other workers. They're called service bindings.

When Cloudflare fires up a worker with a service binding, it makes it instantly available to that worker. No latency, no delays, and no worries about networking hiccups like you'd have with HTTP calls. In software engineering, this is often known as a zero-cost abstraction. In short, this means it gives you benefits without any drawbacks.

Service bindings not only make your apps more reliable and less prone to failures, but they also promote composability and fine-grained workers without any extra performance cost when communicating between them. It really is a nice feature.

When one worker sends a request over to another, the cost is straightforward - it's just like any other request to a worker. You get billed for the CPU time

that worker uses, plus the sub-request contributes to your overall request count.

Now, to get the hang of making API calls between different workers, you're going to make one last change to your Photo API. You'll add a second worker to bring some authentication into the mix. When you're dealing with multiple APIs scattered across different workers, this move makes it super easy to apply consistent authentication across the board.

You're effectively creating a new worker that behaves like middleware.

To understand how worker-to-worker calls function, let's see them in action.

Create The Authentication Worker

You'll use the same steps as in Chapter 1, Deploy Your First Serverless Function, on page ? to create the authentication worker:

```
npm create cloudflare@2.21.1
```

I suggest running this one level above your photo-service worker, so the folder structure would look like this:

```
photo-service/
├─ .wrangler/
├─ migrations/
├─ node_modules/
├─ src/
├─ test/
authentication-service/
├─ node_modules/
├─ src/
```

When running the command, enter `authentication-service` for the directory, select "Hello World" for the worker type, yes to TypeScript and yes to Git.

With the skeleton of the authentication worker created, you can now add the specific changes needed for the authentication worker.

Monorepo vs. Many Repos

There are two common approaches for arranging all the pieces that make up a single application. The first is what you can see above, where you have two separate folders and each would have its own repository under version control. You would make changes to each project individually, and each one would be deployed separately.

Monorepo vs. Many Repos

The alternative approach is called a monorepo, which involves putting all of the services together in a single repository. When you merge changes to the monorepo, any services updated will all be deployed. You may still need to deploy services in a certain order, which can be handled by your deployment pipeline, but the trigger will be a single merge to the monorepo.

A monorepo is great for sharing code between projects, so an alternative approach to what you will see in this chapter would be to have the authentication logic be a shared library between your many APIs. Monorepos can also be slower to deploy, but you can mitigate a lot of this by only deploying the individual services that were changed. A common downside of monorepos is complexity: if you don't keep it organized, it can become unwieldy and complicated, with dependencies often hard to work out and manage.

Which one is right for your project is going to come down to the individual project, so I would recommend trying out both approaches, and seeing which one works for you.

Make A Worker Private

In the case of the photo service, that API would be public-facing. Therefore, when you deploy to Cloudflare, you get a URL generated that allows you to hit it with requests.

Sometimes, you might not want your service to be accessible to the outside world. Take the authentication service, for example — no need for it to be out there publicly since it's always called by a service binding.

Making a worker private is achieved by disabling the auto-generated URL, which is a simple configuration change. Open wrangler.toml in your authentication worker, and add the following line to the bottom:

```
workers_dev = false
```

That's all there is to it, if you now deployed this worker, you'd see no URL was generated based on your account's subdomain, effectively making it private and unreachable via the public internet.

When deploying to production, you'll most likely want a custom domain associated with the photo service. Assigning custom domains to your projects in Cloudflare will be covered in Chapter 15, Deploy To Production, on page

?. To keep the authentication worker private, you simply wouldn't assign a custom domain.

With the worker now private, let's add the authentication logic to your worker.

Add Authentication Logic

The last change you need to make to the authentication service is to actually add some authentication logic. Now, just a heads-up, you're keeping it dead simple for this demo, but in the real world, I'd advise a more robust approach like OAuth with JWT tokens for authentication.

You'll be using a shared secret key to authenticate requests. Whoever's calling the API just needs to put that secret key into the header of the HTTP request. The authentication service will then cross-check that header value with the secret key.

Let's add the code to the worker, inside of src/index.ts:

05-create-auth-worker.ts

```
export interface Env {
        API_AUTH_KEY: String;
}

export default {
        async fetch(
    request: Request,
    env: Env,
    ctx: ExecutionContext
  ): Promise<Response> {
    const api_key = request.headers.get('x-api-auth-key');

    if (api_key === env.API_AUTH_KEY) {
      return new Response('Authenticated', { status: 200 });
    }

    return new Response('Unauthorized', { status: 401 });
  }
};
```

As you can see, the logic is straightforward. The worker simply retrieves the x-api-auth-key header, and compares it with the secret that's stored in env.API_AUTH_KEY. If it's a match, the worker returns a 200, if it's not, it returns a 401.

You'll perhaps notice API_AUTH_KEY defined in the Env interface at the top. But how does that value get set?