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.

Copyright © The Pragmatic Programmers, LLC.

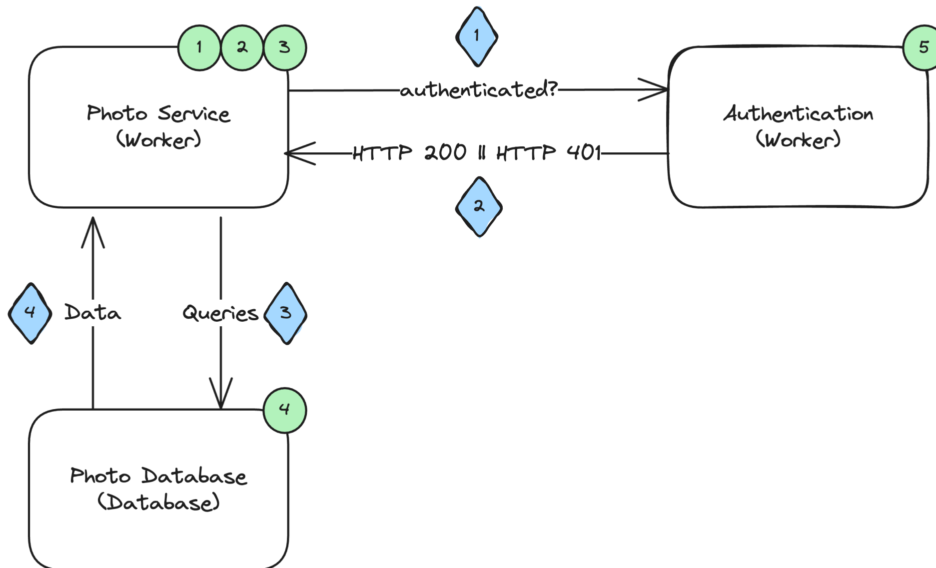
CHAPTER 1

Deploy Your First Serverless Function

Let's jump right in and deploy your first serverless function to Cloudflare. You're going to build an API for storing photos. Over the course of the next five chapters, you will gradually build up the functionality of the API. Here are the steps you'll take:

1. Deploy a simple Worker (you'll do that in this chapter)
2. Update the Worker to include some API endpoints, using in-memory storage for now
3. Explore how you can test a worker programmatically
4. Replace the in-memory storage with a SQL database
5. Introduce some middleware to provide simple authentication, using worker-to-worker calls

By the time you finish Chapter 5, you'll have built all of this. If, like me, you like to see it pictured, here's a simple diagram of what you're building. Circled numbers show the new elements introduced in each chapter. Diamond-numbered steps represent the sequence of interactions when the application processes a request.



OK, let's create a Cloudflare Worker.

Create A New Worker

When working on Cloudflare's platform, you'll mainly use their command-line tool (CLI) called Wrangler alongside C3. They do pretty much everything you need when building applications on Cloudflare — creating projects, deploying code, checking serverless app logs and much more.

Right now, you'll use C3 to make what Cloudflare calls a worker (Azure calls them Functions, AWS calls them Lambdas — different names, same deal: deploying code in a serverless setup). While different cloud providers might have different names for their products, they fundamentally work in the same way.

Cloudflare Workers run on Google Chrome's V8 JavaScript engine, meaning you can use whatever you use in client-side JavaScript. It's important to realize that it's not a Node environment, though it does support a growing number of Node packages.[1]

In May 2023, three new APIs — stream, path, and string decoder — were added to Cloudflare's runtime. More Node APIs will be likely added over time, and I suspect eventually there will be full Node compatibility, but for now, there's a limit to the npm packages you can use.

1. https://developers.cloudflare.com/workers/platform/nodejs-compatibility/

Language Support

Cloudflare supports more than just JavaScript and TypeScript; any language that compiles to JavaScript is fair game. For example, PHP, FSharp, Scala, Python, Kotlin, and more.

If you're into WebAssembly (WASM), you're covered too. While WASM language support is expanding, Cloudflare specifically highlights C and Rust as languages that work well on their platform when compiled to WASM.

First-class support for other programming languages is coming too. In April 2024, Cloudflare announced that Python can now be deployed directly to a Worker without the need to compile it to JavaScript or WebAssembly.

For information on other languages and the complete list of what's supported, Cloudflare has an exhaustive list in their docs.

Before you dive in, ensure you have Node.js and Node Package Manager (npm) installed - you'll need at least version 18 of Node. Wrangler relies on them to do its thing.[2]

Once installed, you can create a new project using C3, which is what Cloudflare calls its CLI tool for use with `npm create`:

```
npm create cloudflare@2.21.1
```

Note: I'll be using the version number of 2.21.1 when creating new projects. When creating your own projects, you can use cloudflare@latest to use the latest version, ensuring you have the latest and greatest features.

At the time of writing, Cloudflare will upgrade you automatically to the latest minor version, even if you specify a specific version. Therefore, by the time you run this command, there will likely be newer versions.

You'll be prompted to answer a few prompts.

- Cloudflare will ask to install the create-cloudflare npm module
- For the directory, you can use whatever you like, I am going to use `photo-service`. This will also, by default, be the name of your worker.
- Select "Hello World" worker
- Use TypeScript
- Use Git

2. https://docs.npmjs.com/downloading-and-installing-node-js-and-npm

- Lastly, it will ask if you want to deploy your application. For now, opt not to deploy.

After a moment, you should see something like this:

```
$ npm create cloudflare@2.21.1
Need to install the following packages:
  create-cloudflare@2.21.1
Ok to proceed? (y) y

using create-cloudflare version 2.21.1

╭ Create an application with Cloudflare Step 1 of 3
│
├ In which directory do you want to create your application?
│ dir ./photo-service
│
├ What type of application do you want to create?
│ type "Hello World" Worker
│
├ Do you want to use TypeScript?
│ yes typescript
│
├ Copying files from "hello-world" template
│
├ Retrieving current workerd compatibility date
│ compatibility date 2023-10-30
│
├ Do you want to use git for version control?
│ yes git
│
╰ Application created
```

The script will then install dependencies using npm. You might think `workerd` is a typo: it's not. It's the name for runtime Cloudflare uses for its Workers.

Although you're using `npm create`, under the hood a lot of the work is being done by Wrangler. You'll interact more closely with Wrangler throughout the book. In short, it's Cloudflare's Developer Platform command-line interface (CLI), allowing you to manage projects and resources.

That's all there is to creating a new worker. You could now create a Cloudflare account and run the deploy command, and you'd have your first worker deployed without any hassle.

For now though, let's see what your options are for configuring that worker.

Configure A Worker

Let's take a look at a few important files.

First, there's that wrangler.toml. I'd never come across Tom's Obvious Minimal Language (TOML) before, but if you've ever worked with YAML, TOML is providing the same thing, a human-friendly format for your application's configuration.

This is where all your Cloudflare-specific configuration for your worker goes. It's presently pretty bare, but you'll be dressing it up in later chapters. There are four configuration options currently set:

- *name*: the name for your worker. This value must be url-friendly, so must not include spaces, but you can use hyphens for separating words.
- *main*: this tells Cloudflare which file is the entry point to your worker, so when it receives a request, it will call the fetch function in the file configured here.
- *compatibility_date*: finally, this is used by Cloudflare to determine what version of their worker's runtime to use. Cloudflare recommends periodically updating this date, but you can just leave it, and Cloudflare will support old runtimes forever (their words!).
- *compatibility_flags*: allow you to tweak the runtime environment, and we'll cover this later in the book.

The compatibility date is there because bugs you rely on might get fixed in newer versions, and Cloudflare's platform could undergo backwards-incompatible changes. Check out Cloudflare's documentation for the complete list of such changes.[3]

wrangler.toml is the configuration file you will be changing the most. But two others are worth mentioning:

- *package.json*: used to define all the npm packages your worker depends on
- *tsconfig.json*: used to tweak the TypeScript configuration

That covers the configuration files, so let's take a look at the code C3 has created to get you started.

Implement Logic In Your Workers

It's time to look at the code that will be executed when your worker receives a request. All code in the book will be TypeScript, but if types aren't your thing feel free to use vanilla JavaScript.

3. https://developers.cloudflare.com/workers/platform/compatibility-dates/

• Click HERE to purchase this book now. discuss

All application code for your workers will be in the src directory, and for now Cloudflare has created a single file for you. It will look something like this:

01-example-worker.ts

```
export interface Env {
  // MY_KV_NAMESPACE: KVNamespace;
}

export default {
  async fetch(
    request: Request,
    env: Env,
    ctx: ExecutionContext
  ): Promise<Response> {
    return new Response('Hello World!');
  },
};
```

Short and sweet, but this demonstrates something very important: the entry point to every Cloudflare worker. Each Cloudflare worker must export a default module, exposing methods that Cloudflare will call when your worker receives a request.

As you can see, a single fetch function was generated, which is called whenever a HTTP request is received by your worker. There's also an interface defined for the environment that is used to define some Cloudflare-specific dependencies. You'll be using these throughout the book, with the first one introduced in Chapter 4, Add Persistence To The API, on page ?.

The fetch function takes three parameters:

1. *request* represents the HTTP request that was received, and exposes methods that allow you to see the body, headers and HTTP method. Its type comes from the Fetch API that's built into JavaScript, with Cloudflare adding some extra functionality to it.[4]
2. *env* provides information available in the environment, such as secrets, and in later chapters will give you access to other Cloudflare services, such as databases, caches and queues. You can see some information and links to these products in the Env interface defined at the top of the file.
3. *ctx* is the execution context, and provides access to a couple of methods that allow you to change the behavior of your worker. For example, it provides a method called waitUntil that allows you to extend the execution time of your worker after returning a response. This is particularly handy

4. https://developer.mozilla.org/en-US/docs/Web/API/Request

for returning a response to the client, while still processing something in your worker.

Lastly, the function has to return a response to the incoming HTTP request. The response is always an instance of the Response class, that once again comes from the Fetch API. For now, the worker is just returning a piece of text that I'm sure you've seen before.

Organizing Serverless Code

Good programming practices still apply to serverless, with the entry point to your serverless function effectively being the controller, and then incoming requests can be routed by the controller to your application's logic.

For instance, creating a simple API might involve having a serverless function for each endpoint. Imagine /foo and /bar having their own serverless functions, implemented by individual Workers. Alternatively, you could go for a more "monolithic" approach, routing all requests to a single function, which then handles them accordingly.

There's no strict right or wrong here — just trade-offs. Multiple functions allow you to keep each function small and focussed, but may lead to code duplication. With multiple functions, it's also a bit trickier to visualize how the functions map to the API structure. On the flip side, a monolithic function reduces duplication and offers a clear API view, but the code might get complex as your API grows. In this chapter, you'll get to see what the monolithic approach looks like.

As a rule of thumb, I'd start with a monolithic function. If things get complicated, you can split it later. Thanks to the ease of deploying serverless functions, making such changes is typically way less costly than breaking up a monolithic container-based app.

Run Scheduled Tasks

Cloudflare lets you define Cron Triggers that execute your worker based on a schedule. To do so, add a scheduled function to your worker and set up the schedule using cron syntax. For instance, in your wrangler.toml, add this:

```
[triggers]
crons = ["*/15 * * * *"]
```

• Click HERE to purchase this book now. discuss

This would result in the worker's scheduled function being executed every 15 minutes. As crons is an array, you can configure a worker to be executed on different schedules if you need. Here's how you can differentiate between different triggers being executed:

01-example-scheduled.ts
```
export default {
  async scheduled(
    event: ScheduledController,
    env: Env,
    ctx: ExecutionContext
  ) {
    switch (event.cron) {
    // You can set up to three schedules maximum.
    case "*/15 * * * *":
      console.log("This will be executed every 15 minutes");
      break;
    case "*/30 * * * *":
      console.log("This will be executed every 30 minutes");
      break;
    }
  }
}
```

When the scheduled function is triggered, you can see which cron trigger is being executed by looking at event.cron. It's possible to define both a fetch and scheduled function and have your worker handle HTTP requests and scheduled tasks.

The current worker has no need to make use of scheduled tasks, but it's useful to know this functionality is available for future.

In order to see the full lifecycle of a Cloudflare worker, let's deploy this skeleton worker to Cloudflare.