Extracted from:

Ruby on Rails Background Jobs with Sidekiq

Run Code Later without Complicating Your App

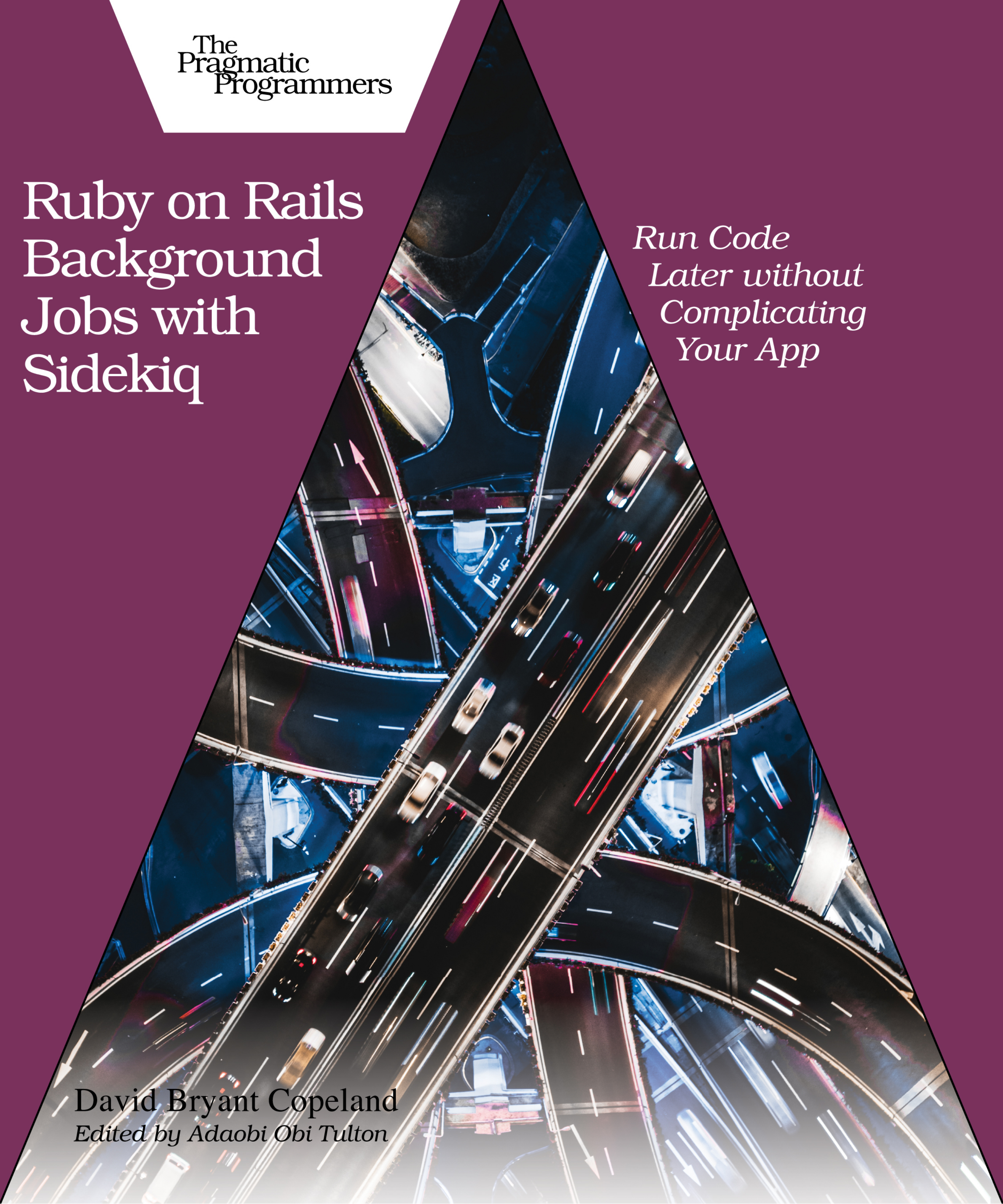
The Pragmatic Bookshelf

Dallas, Texas

The Pragmatic Programmers

Ruby on Rails Background Jobs with Sidekiq

Run Code Later without Complicating Your App



David Bryant Copeland

Edited by Adaobi Obi Tulton

Ruby on Rails Background Jobs with Sidekiq

Run Code Later without Complicating Your App

David Bryant Copeland

The Pragmatic Bookshelf

Dallas, Texas

The team that produced this book includes:

Publisher: Dave Thomas
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Adaobi Obi Tulton

ISBN-13: 979-8-88865-036-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2023

Creating a Baseline Sidekiq Configuration

As you'll learn, operating an app that uses Sidekiq requires vigilance and adjustment. Your workloads are unique to your app, which means your Sidekiq configuration will be equally unique. But it's important to see a baseline configuration that you can build on. Sidekiq has great defaults, but setting up an explicit and flexible configuration will be helpful later when you need to make changes.

First, install the Sidekiq gem by adding this line to your Gemfile. Note that call to gem is preceded by a comment explaining what Sidekiq does. "Sidekiq" is a cool name, but for anyone on your team that hasn't heard of it, a few words in the Gemfile can go a long way.

snapshots/1-2/sidekiq-book/Gemfile

```
  # This is used to run the dev environment
  gem "foreman"
➤
➤ # Sidekiq is used for running background jobs
➤ gem "sidekiq"

  # Prevents our workers from running too long if a request
  # doesn't return in time.
```

You can install this with bundle install:

```
> bundle install
«Lots of output»
```

Set Sidekiq's Configuration Options

There are four options you need to configure in Sidekiq:

- *Redis*: Sidekiq needs to know how to connect to Redis. This can be configured with an environment variable that Sidekiq will read.
- *Concurrency*: Sidekiq needs to know how many threads to run per server process. You are likely to want to change this without having to wait for a re-deploy so allowing an environment variable to override a default is a good strategy here.
- *Timeout*: When Sidekiq asks a job to terminate, the timeout is how long it will wait for forcing a job to stop. This is another value you may want to change without a re-deploy, so you can use an environment variable that overrides a default.

- *Queues*: Queues aren't added often, and they must be accompanied by a code change that uses the queue. Explicitly hard-code the list of queues to Sidekiq's default value so you know where to change it in the future.

To connect to Redis, you'll use a few environment variables. In the example app's development and test environment, the environment variables are managed by the dotenv gem.[2] That gem will examine the files .env.development and .env.test and use them to set environment variables for development and testing, respectively.

If you look at the files in the example app, you can see that there is a Redis URL set to the environment variable SIDEDKIQ_REDIS_URL (this is already set so that you could validate the example app's ability to access Redis before installing Sidekiq):

snapshots/0-0/sidekiq-book/.env.development

```
DATABASE_URL=postgres://postgres:postgres@db:5432/sidekiq-book_development
SIDEKIQ_REDIS_URL=redis://redis:6379/1
FULLFILLMENT_API_URL=http://fake-api-server:4000/fulfillment
PAYMENTS_API_URL=http://fake-api-server:4000/payments
EMAIL_API_URL=http://fake-api-server:4000/email
ERROR_CATCHER_API_URL=http://fake-api-server:4000/error-catcher

# Limit all requests to 5 seconds
RACK_TIMEOUT_SERVICE_TIMEOUT=5
```

The example app uses the name SIDEDKIQ_REDIS_URL to be explicit about what the purpose of the Redis instance is. Sharing a Redis with another function, like caching, is a recipe for disaster as a full cache could prevent you from queuing jobs or vice versa.

But Sidekiq doesn't know you've used this name. Rather than create an initializer to fetch the Redis URL from the environment, Sidekiq allows you to set *another* environment variable named REDIS_PROVIDER that contains the name of the environment variable that contains the Redis URL. In .env.development, set REDIS_PROVIDER to SIDEDKIQ_REDIS_URL:

snapshots/1-1/sidekiq-book/.env.test

```
DATABASE_URL=postgres://postgres:postgres@db:5432/sidekiq-book_test
SIDEKIQ_REDIS_URL=redis://redis:6379/2
➤ REDIS_PROVIDER=SIDEKIQ_REDIS_URL
FULLFILLMENT_API_URL=http://fake-api-server:4000/fulfillment
PAYMENTS_API_URL=http://fake-api-server:4000/payments
EMAIL_API_URL=http://fake-api-server:4000/email
ERROR_CATCHER_API_URL=http://fake-api-server:4000/error-catcher
```

2. https://github.com/bkeepers/dotenv

You'll need to make a similar change to .env.test:

snapshots/1-1/sidekiq-book/.env.development

```
DATABASE_URL=postgres://postgres:postgres@db:5432/sidekiq-book_development
SIDEKIQ_REDIS_URL=redis://redis:6379/1
REDIS_PROVIDER=SIDEKIQ_REDIS_URL
FULLFILLMENT_API_URL=http://fake-api-server:4000/fulfillment
PAYMENTS_API_URL=http://fake-api-server:4000/payments
EMAIL_API_URL=http://fake-api-server:4000/email
ERROR_CATCHER_API_URL=http://fake-api-server:4000/error-catcher

# Limit all requests to 5 seconds
RACK_TIMEOUT_SERVICE_TIMEOUT=5
```

This indirect mechanism may seem odd, but it is useful. It was originally designed to help developers on Heroku, who were not usually able to control the name of the environment variable for the Redis instances Heroku provided. Despite this historical curiosity, the indirection through REDIS_PROVIDER allows you to perform a manual failover on any hosting platform, even one you manage yourself. You could deploy a second Redis instance and set a new variable named, say NEW_REDIS_URL, to its connection string. Once you change REDIS_PROVIDER to have the value NEW_REDIS_URL and restart your app, your app will be using this new Redis without you having made any code changes. You may never need this, but it's handy to have and minimizes the amount of configuration in your app.

For the rest of the configuration options, you'll use config/sidekiq.yml. Sidekiq parses this as if it were a .erb file, so you can put embedded code into it to read from the environment via ENV. You can use this to allow the environment to change the values for concurrency and timeout, falling back to Sidekiq's defaults if the environment variable is not set. There's no need to use this technique for queues or maximum retries, since these values are not likely to require changing without an associated code change. Explicitly setting them to their defaults is a good idea, however, so that it's easier to know where to change them in the future.

snapshots/1-1/sidekiq-book/config/sidekiq.yml

```
:concurrency: <%= ENV.fetch("SIDEKIQ_CONCURRENCY") { 5 } %>
:timeout: <%= ENV.fetch("SIDEKIQ_TIMEOUT_SECONDS") { 25 } %>
:queues:
  - default
```

Configure the Web UI

The last part of the configuration is to set up the Sidekiq Web UI. The Web UI is a Rack App that you can mount onto any path in the app.[3] It's an invaluable resource for understanding and managing Sidekiq in your app (we'll use it in Handling Permanent Failures via Monitoring, on page ? and learn about some of its insights on job behavior in Sidekiq's Web UI Provides Observability at a Glance, on page ?).

First, require sidekiq/web at the top of config/routes.rb like so:

snapshots/1-1/sidekiq-book/config/routes.rb

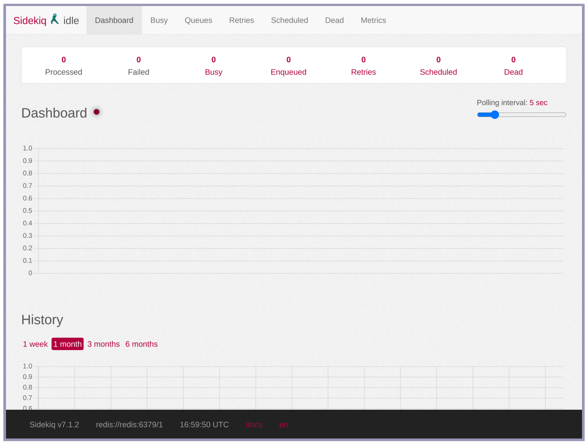
```ruby
require "sidekiq/web" # Brings in the Sidekiq Web UI

Rails.application.routes.draw do
  resources :orders, only: [ :new, :create, :show ]
  resources :simulated_behaviors, only: [ :edit, :update ]
```

Then, mount the app using the mount method. The app's class is Sidekiq::Web and can be mounted to /sidekiq (though you can use whatever path you like):

snapshots/1-2/sidekiq-book/config/routes.rb

```ruby
  resources :simulated_behaviors, only: [ :edit, :update ]
  root "welcome#show"
  # make the Sidekiq Web UI available on /sidekiq
  mount Sidekiq::Web => "/sidekiq"
end
```

Restart your server and navigate to http://localhost:3000/sidekiq and you should see the Web UI looking similar to the following screenshot:



The Sidekiq Web UI showing various statistics about how Sidekiq is running.

3. https://github.com/rack/rack

Since this is a standard Rack App, you can secure access to it by whatever means you like. Just be sure that you *do* secure access to it, since it can expose sensitive information and allow a bad actor to create operational chaos.

Set Up Sidekiq to Run in Development

Before you start coding, you need to be able to run Sidekiq locally when you execute bin/dev. bin/dev uses the file Procfile.dev as a way to know what commands you want to run when starting up your app. Right now, it has commands for running the Rails server as well as bundling front-end assets. Add a new line that runs the Sidekiq server:

snapshots/1-2/sidekiq-book/Procfile.dev

```
  web: PORT=3000 bin/rails s
➤ sidekiq: bundle exec sidekiq
  js: yarn build --watch
  css: yarn build:css --watch
```

Now let's use Sidekiq!