

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

by David Bryant Copeland

Version: P1.0 (October 2023)

Copyright © 2023 The Pragmatic Programmers, LLC. This book is licensed to the individual who purchased it. We don't copy-protect it because that would limit your ability to use it for your own purposes. Please don't break this trust—you can use this across all of your devices but please do not share this copy with other members of your team, with friends, or via file sharing services. Thanks.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

About the Pragmatic Bookshelf

The Pragmatic Bookshelf is an agile publishing company. We're here because we want to improve the lives of developers. We do this by creating timely, practical titles, written by programmers for programmers.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at <http://pragprog.com>.

Our ebooks do not contain any Digital Restrictions Management, and have always been DRM-free. We pioneered the beta book concept, where you can purchase and read a book while it's still being written, and provide feedback to the author to help make a better book for everyone. Free resources for all purchasers include source code downloads (if applicable), errata and discussion forums, all available on the book's home page at pragprog.com. We're here to make your life easier.

New Book Announcements

Want to keep up on our latest titles and announcements, and occasional special offers? Just create an account on pragprog.com (an email address and a password is all it takes) and select the checkbox to receive newsletters. You can also follow us on twitter as [@pragprog](https://twitter.com/pragprog).

About Ebook Formats

If you buy directly from pragprog.com, you get ebooks in all available formats for one price. You can synch your ebooks amongst all your devices (including iPhone/iPad, Android, laptops, etc.) via Dropbox. You get free updates for the life of the edition. And, of course, you can always come back and re-download your books when needed. Ebooks bought from the Amazon Kindle store are subject to Amazon's policies. Limitations in Amazon's file format may cause ebooks to display differently on different devices. For more information, please see our FAQ at pragprog.com/#about-ebooks. To learn more about this book and access the free resources, go to <https://pragprog.com/book/dcsidekiq>, the book's homepage.

Thanks for your continued support,

The Pragmatic Bookshelf

The team that produced this book includes: Dave Thomas (Publisher), Janet Furlow (COO), Tammy Coron (Managing Editor), Adaobi Obi Tulton (Development Editor)

For customer support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Table of Contents

[Acknowledgments](#)

[Preface](#)

[Introduction](#)

[What You'll Need](#)

[What You'll Learn](#)

1. [From Zero To a Sustainable Sidekiq Config](#)

[Setting Up the Example App and Installing Sidekiq](#)

[Creating a Baseline Sidekiq Configuration](#)

[Moving Code to a Sidekiq Job](#)

[Up Next](#)

2. [Handle Failures by Planning Ahead](#)

[Making Sure Jobs Succeed...Eventually](#)

[Understanding Why Jobs Fail](#)

[Handling Permanent Failures via Monitoring](#)

[Managing Transient Failures Through Smarter Code](#)

[Monitoring for Failed Jobs That Stop Retrying](#)

[Up Next](#)

3. [Safely Retry Jobs by Making Them Idempotent](#)

[Retrying Failed Jobs Creates New Failures](#)

[Understanding Idempotence](#)

[Breaking Down Large Jobs into Safely Retriable Parts](#)

[Addressing Idempotence Issues with Third Parties](#)

[Testing Code for Idempotency](#)

[Assessing Code for Idempotency](#)

[Using Database Transactions Can Help...Sometimes](#)

[Up Next](#)

4. [Sustainable Operations and Development](#)

[Monitoring Sidekiq Performance](#)

[Using Queues and Concurrency to Control Performance](#)

[Organizing Sidekiq Job Code](#)

[Additional Topics to Explore Next](#)

[Wrapping Up](#)

Early Praise for *Ruby on Rails Background Jobs with Sidekiq*

Dave's years of experience with Sidekiq are apparent; he shows you how to use Sidekiq for long-term success. I appreciate his focus on sustainable operations treating application bugs and performance regressions as inevitable. You'll learn how to monitor your jobs to ensure such issues are quickly and easily discovered.

→ Mike Perham

Founder of Sidekiq

Sidekiq is a critical part of big Rails architecture, and using it correctly can make a huge difference in your infrastructure performance. This book shows you how to use Sidekiq safely and sustainably, using techniques that Dave Copeland has used in deploying large-scale Rails apps. It's like hiring a senior engineer in book form.

→ Noel Rappin

Author of *Programming Ruby 3.2* 5th Edition

Acknowledgments

I would like to thank Sidekiq’s author, Mike Perham (as well as all the people who have contributed to Sidekiq over the years), for making an awesome background job system for Rails. I’d also like to thank Patrick Joyce for allowing me—someone with no professional Rails or background job experience at the time—to manage a very busy background job system charging customers money. I learned a lot.

Preface

My first job with Ruby on Rails was at LivingSocial in 2011, about two months before the biggest shopping day in the United States called "Black Friday." I was the brand new owner of our payments system, which was built on background jobs using Resque. I had never heard of Resque and never worked on a system that had background jobs. I had never worked on a system that charged people money.

The only other person that knew anything about this code got so sick at one point that the CTO had to call his wife to make sure he was still alive (he was). During that time, thousands of background jobs were failing. They sat in a queue with error messages no one understood. The CTO, VP of Engineering, and I sat in front of a computer together and eventually figured out how to fix these jobs. They wouldn't be the last, but I became self-sufficient over the next year. It was a painful crash course in resiliency, fault tolerance, and background jobs.

Many years later, I was building a system that was almost entirely background jobs. I chose Sidekiq since it was clearly the de facto standard for background jobs in Rails. Although I had a lot of experience with background jobs at this point, I had none with Sidekiq. It was immediately obvious how much thought was put into the design. Everything I needed was there, and any strategy I wanted to employ to manage production issues was easily supported. It was a joy to work with.

Thinking back, if someone had explained and demonstrated what happens in

the real world, and shown me some examples of how to deal with it, I could've avoided that harrowing trial by fire. If someone had then said, "Now that you see what happens in a production system, use Sidekiq," I could've saved myself quite a few deep dives into other systems' codebases.

This book, and the code that goes with it, are what I would've wanted. Not just theory, but a real demonstration of what can actually happen, and how to manage it.

Introduction

Sidekiq is a reliable, well-supported, high-performing way to run code in the background of any Rails app. According to Planet Argon's 2022 Ruby on Rails Survey,^[1] 58 percent of the respondents use it for their background job needs, with Delayed Job being the next most popular at only 14 percent. This is no accident. Whether you need to offload slow code out of a controller action, or make a flaky API call more fault tolerant, Sidekiq is the way to do that.

Even though you just need a few lines of code to run a Sidekiq job, there is quite a bit of complexity hiding in the shadows. A Rails app with Sidekiq jobs is really a distributed system. It has all the same challenges as a massive microservices-based architecture. Your jobs will fail and you need to know when they do. Your code will need to evolve to allow these failures to self-heal, and this requires shedding some naivete about software design.

Many experienced developers learn how to manage background jobs and Sidekiq through the School of Hard Knocks.^[2] Luckily for you, this short book can let you skip at least a few semesters.

What You'll Need

This isn't an intro to Sidekiq, so you should have some understanding of what it is and how it works. That said, you'll still be able to follow along if you are experienced with other background job systems like Resque or Delayed Job. You should know the basics of Rails, though you don't need to be an expert.

You will also need a way to run Docker, which we'll use to run the examples. The book includes two applications that allow you to simulate common failure modes when using Sidekiq. Docker will allow you to run them with a single command. If you are using Windows, you will need the Windows Subsystem for Linux, Version 2.[\[3\]](#)

What You'll Learn

First, you'll get Sidekiq configured for a reasonable production environment in Chapter 1, [From Zero To a Sustainable Sidekiq Config](#). After that, you'll learn how to manage failed jobs in Chapter 2, [Handle Failures by Planning Ahead](#). In that chapter, you'll quickly learn how to know when a job fails, how not to be bothered when a transient failure eventually succeeds and, of course, how to find out about transient failures that *don't* eventually succeed.

In Chapter 3, [Safely Retry Jobs by Making Them Idempotent](#), you'll learn how to design your code so it can be safely retried many times in a Sidekiq job, all without having more effects than desired. If you've ever had to manually refund a customer who was mistakenly double charged, this is your chapter. Finally, in Chapter 4, [Sustainable Operations and Development](#), you'll learn how to set up useful monitoring for your Sidekiq installation, manage performance via queues and concurrency, and keep all this code organized.

When you're done, you'll be ready to handle the most common challenges everyone faces when running Sidekiq in production. Let's get started!

Footnotes

- [1] <https://rails-hosting.com/2022/#configuration>
- [2] https://en.wikipedia.org/wiki/School_of_Hard_Knocks.
- [3] <https://learn.microsoft.com/en-us/windows/wsl/install>

Chapter 1

From Zero To a Sustainable Sidekiq Config

Starting with this chapter, and for the rest of the book, we'll be doing work inside an example app. This app is included in the source code of the book in `DIR`. It has a few basic features that we'll use to learn various things about Sidekiq and background jobs.

Before we start, you should get it set up. Once that's done, we'll move some code from the app into a Sidekiq job to see how that works, then get Sidekiq's testing support set up to quickly learn about how Sidekiq jobs affect our tests. Even if you're familiar with Sidekiq, there will be a few important lessons here about configuring Sidekiq as well as a high-level testing approach.

Setting Up the Example App and Installing Sidekiq

The example app requires Docker to run the app, Postgres, Redis (the key/value store Sidekiq uses to store jobs to be processed), and a fake-API server used to simulate payments, email, and order fulfillment. There is a README in the code to explain in more detail how the app works and how the Docker-based setup works. Once you install Docker,^[4] `cd` to the development environment in `dev-environment`, and perform a one-time step of building the Docker container where you'll do your work:

```
> dx/build
```

Then, start everything up, like so:

```
> dx/start
```

In a new terminal session, you can run `bash` inside the Docker container that will run the example app, like so:

```
> dx/exec bash
```

Running 'bash' inside container with id big string

```
root@dcaa5f00cca0:~/work#
```

You'll then need to `cd` into the example book's root directory:

```
root@dcaa5f00cca0:~/work# cd sidekiq-book
```

```
root@dcaa5f00cca0:~/work/sidekiq-book#
```

You can run all the commands you need to work on the app from this bash session. You don't have to edit the app's code this way, however. You can edit the code directly on your computer and it will be reflected inside the Docker container. For example, if you changed a test on your computer, you can switch to the container and run `bin/rails test` to see the results.

Let's get the app set up. From here on out, all command lines will be happening inside the Docker container unless indicated otherwise.

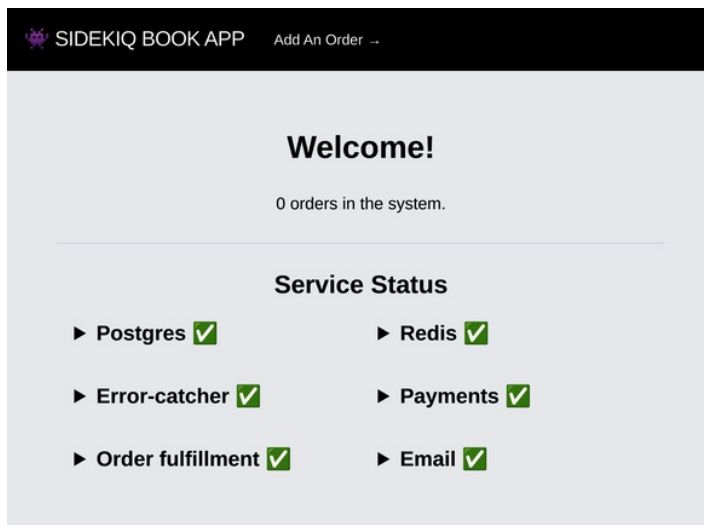
```
> bin/setup
```

lots of output

That should've installed the needed gems and set up your database. Now run the app:

```
> bin/dev
```

The app is now running on port 4000 inside the container, which is mapped to port 4000 on your computer, so you should be able to go to <http://localhost:4000> and see the app running. The home page, shown below, shows some basic status information about Postgres, Redis, and three fake services that our Sidekiq jobs will interact with.



Screenshot of the welcome page of the example app that shows the status of Postgres, Redis, a Payments service, an Email service, and an Order fulfillment service.

Next, we'll install and set up Sidekiq with a baseline configuration you can use for production.

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.^[5] 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 `SIDEDIQ_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 `SIDEDIQ_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 `SIDEDIQ_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
```

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.^[6] It's an invaluable resource for understanding and managing Sidekiq in your app (we'll use it in [Handling Permanent Failures via Monitoring](#) and learn about some of its insights on job behavior in [Sidekiq's Web UI Provides Observability at a Glance](#)).

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

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

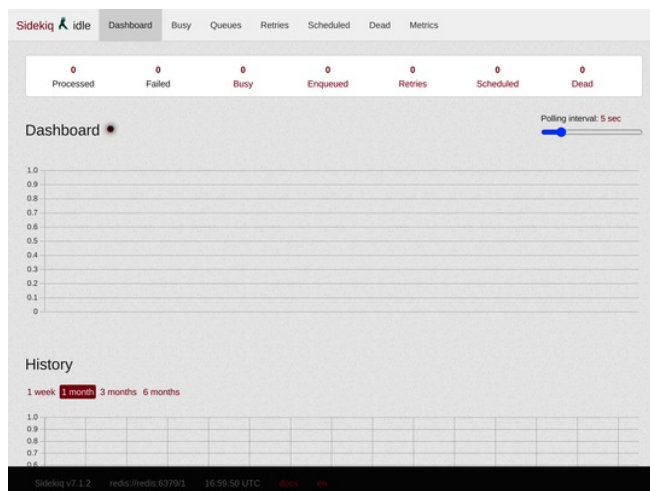
```
» require "sidekiq/web" # Brings in the Sidekiq Web UI
»
Rails.application.routes.draw do
  resources :orders, only: [ :new, :create, :show ]
  resources :simulated_behaviors, only: [ :edit, :update ]
```

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
```

```
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.

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!

Moving Code to a Sidekiq Job

A common use of Sidekiq jobs is to move long-running code out of the web request/response cycle and into a background job to free up web server resources to handle other requests. This often means that the user can get a response much more quickly since they won't have to wait for the slow code from, say, the payments service, to complete. Let's move all three HTTP calls out of the main request/response path and into a Sidekiq job.

Create a New Sidekiq Job

The example app's order creation logic makes calls to payments, email, and order fulfillment services over HTTP. If these were real services provided by third parties, the order creation feature would be very slow, and our app's web servers would spend a lot of time doing nothing (meaning you'd need a lot of web servers to sit around doing nothing to handle the app's traffic).

Here's the main logic of how everything works currently:

```
snapshots/1-2/sidekiq-book/app/services/order_creator.rb
```

```
def create_order(order)
  if order.save
    payments_response = charge(order)
    if payments_response.success?

      email_response    = send_email(order)
      fulfillment_response = request_fulfillment(order)

      order.update!(
        charge_id: payments_response.charge_id,
        charge_completed_at: Time.zone.now,
        charge_successful: true,
        email_id: email_response.email_id,
        fulfillment_request_id: fulfillment_response.request_id)
    else
      order.update!(
```



```

      charge_completed_at: Time.zone.now,
      charge_successful: false,
      charge_decline_reason: payments_response.explanation
    )
  end
end
order
end

```

The methods `charge`, `send_email`, and `request_fulfillment` wrap the calls to those three services (you may examine the entire file if you want to see how that works). We can think about the process as having two parts. The first validates the order, and the user has to wait for that to complete. The second part is to handle payments, email, and order fulfillment and the user does *not* have to wait for those to complete. Let's change the code so that this second part of the logic is triggered from a job called `CompleteOrderJob`.

You don't want a lot of logic in the job classes. They should act like controllers in that they make sense of the arguments given, then defer execution to another class. In our case, `OrderCreator` exists already, so assume a method named `complete_order` will exist and write your job like so:

```

snapshots/1-2/sidekiq-book/app/jobs/complete_order_job.rb

```

```

class CompleteOrderJob
  include Sidekiq::Job

  def perform(order_id)
    order = Order.find(order_id)
    OrderCreator.new.complete_order(order)
  end
end

```

`complete_order` will basically do everything from the original `create_order` method after the model passes validation.

```

snapshots/1-2/sidekiq-book/app/services/order_creator.rb

```

```

end

```

```

» def complete_order(order)
»   payments_response = charge(order)
»   if payments_response.success?
»
»     email_response    = send_email(order)
»     fulfillment_response = request_fulfillment(order)
»
»     order.update!(
»       charge_id: payments_response.charge_id,
»       charge_completed_at: Time.zone.now,
»       charge_successful: true,
»       email_id: email_response.email_id,
»       fulfillment_request_id: fulfillment_response.request_id)
»   else
»     order.update!(
»       charge_completed_at: Time.zone.now,
»       charge_successful: false,
»       charge_decline_reason: payments_response.explanation
»     )
»   end
» end
private

def charge(order)

```

Lastly, remove the code from `create_order` and replace it with code to queue the new job.

snapshots/1-3/sidekiq-book/app/services/order_creator.rb

```

if order.save
# XXX   payments_response = charge(order)
# XXX   if payments_response.success?
# XXX
# XXX     email_response    = send_email(order)
# XXX     fulfillment_response = request_fulfillment(order)
# XXX
# XXX     order.update!(
# XXX       charge_id: payments_response.charge_id,
# XXX       charge_completed_at: Time.zone.now,
# XXX       charge_successful: true,

```

```

# XXX      email_id: email_response.email_id,
# XXX      fulfillment_request_id: fulfillment_response.request_id)
# XXX      else
# XXX      order.update!(
# XXX      charge_completed_at: Time.zone.now,
# XXX      charge_successful: false,
# XXX      charge_decline_reason: payments_response.explanation
# XXX      )
# XXX      end
»      CompleteOrderJob.perform_async(order.id)
end
order

```

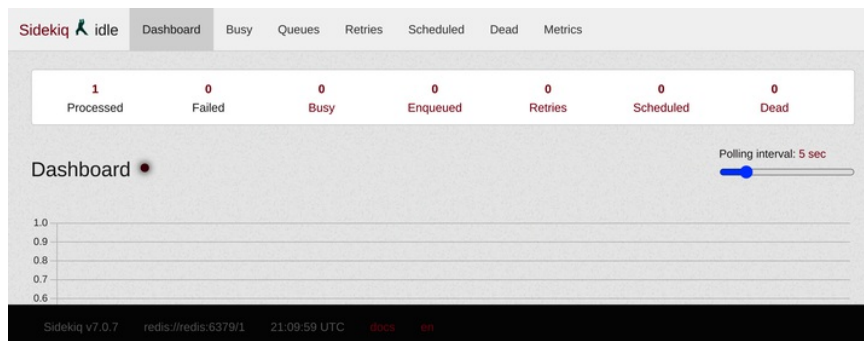
With this in place, go back to the app and restart it. Create an order, then watch the output of `bin/dev`. You should see a line like so (which is wrapped to fit in this book):

```

17:54:49 sidekiq.1 | 2023-03-13T17:54:49.253Z pid=3353 tid=8ep \
class=CompleteOrderJob jid=a6610de48b9 elapsed=0.234 INFO: done

```

This means that Sidekiq processed the job. If you go to the Web UI, you can see that this is reflected there as well:



The Sidekiq Web UI showing that one job was processed.

Although everything is still working, your system test disagrees:

```

> bin/rails test test/system/create_orders_test.rb
Running 2 tests in a single process (parallelization thresho...
Run options: --seed 27262

```

```

# Running:

```

```
2023-03-23T21:58:30.626Z pid=3171 tid=873 INFO: Sidekiq 7.0....
```

```
F
```

```
Failure:
```

```
CreateOrdersTest#test_credit_card_decline [/root/sidekiq-boo...
```

```
expected to find text "Payment Declined: Insufficient funds"...
```

```
rails test test/system/create_orders_test.rb:55
```

```
F
```

```
Failure:
```

```
CreateOrdersTest#test_creating_an_order [/root/sidekiq-book/...
```

```
charge_id was not set.
```

```
Expected nil to not be nil.
```

```
rails test test/system/create_orders_test.rb:4
```

```
Finished in 0.090284s, 22.1524 runs/s, 166.1432 assertions/s...
```

```
2 runs, 15 assertions, 2 failures, 0 errors, 0 skips
```

```
Test Failed
```

Rails's test output isn't very user-friendly, but you should see that both tests have failed. If you examine the tests, you'll see that both tests are failing because the code you moved to a background job has not been executed by the time the test started its assertions. You need to have the jobs run during the tests.

Update Tests to Account for Sidekiq Jobs

Running the Sidekiq server in a test environment is tricky. You can't control exactly when the jobs get executed, so you may end up making assertions before the code has actually run. To help manage this, Sidekiq provides two testing modes: *fake* and *inline*.

- *Fake*: Jobs are queued to an internal data structure you can examine and manipulate. This is the default mode once you've set up Sidekiq's

testing support.

- *Inline*: Jobs are executed when queued, as if they were inlined into the code that queued them.

“Fake” is the best option here, at least as a default (the mode can be overridden in individual tests). This will require our tests to explicitly force jobs to execute, which can couple tests to the jobs being run, but it also means our tests reflect reality. In “Inline” mode, you can’t look at a test and understand how jobs interact with the feature. Inline mode also makes it difficult or impossible to make assertions about the state of the system after a job has been queued but before it has been executed. For a system test, this could be critical as a user may see a different UI while a job is executing.

To enable testing mode, you’ll need to require `sidekiq/testing` in `test/test_helper.rb`:

```
snapshots/1-3/sidekiq-book/test/test_helper.rb
```

```
require_relative "../config/environment"
require "rails/test_help"
require "minitest/autorun"
» require "sidekiq/testing"
require "minitest/mock"
require "factory_bot"
```

Fake mode is the default when Sidekiq’s testing support is enabled, so you don’t need to do any further configuration to enable it. But you do want to make sure there are no jobs from a previous run still in the queue. To do that, call `Sidekiq::Worker.clear_all` inside `setup` in the base test case:

```
snapshots/1-4/sidekiq-book/test/test_helper.rb
```

```
# Add more helper methods to be used by all tests here...
setup do
  Rails.cache.clear
»  Sidekiq::Job.clear_all
end
end
```

The test can now execute the jobs so the assertions can succeed. In Fake mode, Sidekiq adds the method `drain` to all job classes that will execute all queued jobs of that class. Sidekiq also provides `Sidekiq::Worker.drain_all`, which will execute all jobs of all classes. Using that will avoid over-coupling the test to the implementation.

For both tests, you want to add the call to `drain_all` right after you submit the order. Here's the change for the first of the two tests (the use of `refresh` will be explained in a minute):

```
snapshots/1-4/sidekiq-book/test/system/create_orders_test.rb
```

```
fill_in "order[quantity]", with: 2

click_on "Place Order" # place paid order
» Sidekiq::Job.drain_all
» refresh

refute_selector "aside[data-error]"
order = Order.last
```

Make the same change for the second test:

```
snapshots/1-5/sidekiq-book/test/system/create_orders_test.rb
```

```
fill_in "order[quantity]", with: 1

click_on "Place Order" # place declined order
» Sidekiq::Job.drain_all
» refresh

assert_text "Payment Declined: Insufficient funds"
order = Order.last
```

Sure enough, when you rerun the test, everything should be working:

```
> bin/rails test test/system/create_orders_test.rb
Running 2 tests in a single process (parallelization thresho...
Run options: --seed 37387
```

Running:

2023-03-23T22:16:04.909Z pid=4455 tid=ad7 INFO: Sidekiq 7.0....

..

Finished in 0.108150s, 18.4929 runs/s, 258.9002 assertions/s...

2 runs, 28 assertions, 0 failures, 0 errors, 0 skips

The reason we had to call `refresh` in our tests has to do with how the UI works. Right after the `CompleteOrderJob` is queued, the controller returns and shows the order's show view. You may have noticed that as the third-party services are called, various attributes like `email_id` or `charge_decline_reason` are set on the order. The view knows that these values may be blank.

In order to properly assert the view's behavior *after* the Sidekiq job has executed, you have to refresh the page. You could make the page auto-refresh, but for now, you can manually refresh it just to see everything working.

This does demonstrate a reason why Fake mode is preferable: before refreshing the page, you could make assertions about how the order's show view should appear while the job is running. Once you've completed the assertions, call `drain_all`, then `refresh` to proceed.

Before we leave this section, a test of the `OrderCreator` is now failing. We can use the same technique we just did to get these tests to pass. First, add `Sidekiq::Worker.drain_all` and a `reload` to the test for a decline:

```
snapshots/1-5/sidekiq-book/test/services/order_creator_test.rb
```

```
)
resulting_order = @order_creator.create_order(order)
assert_equal order, resulting_order, "should return the same order"
» Sidekiq::Job.drain_all
» resulting_order.reload
refute resulting_order.charge_successful
assert_equal "Insufficient funds", resulting_order.charge_decline_reason
assert_nil resulting_order.charge_id
```


Now, make a similar change in the test for success:

```
snapshots/1-6/sidekiq-book/test/services/order_creator_test.rb
```

```
resulting_order = @order_creator.create_order(order)
assert_equal order, resulting_order, "should return the same order"
» Sidekiq::Job.drain_all
» resulting_order.reload
assert resulting_order.charge_successful
assert_nil resulting_order.charge_decline_reason
refute_nil resulting_order.charge_id
```

Once that's done, you can run all the tests via [bin/ci](#).

Up Next

You've got the example app set up, created your first job, and updated your tests to account for it. Next, let's learn about job failure. You just put a bunch of important code into a background job, so what happens if that job fails?

Footnotes

[4] <https://docs.docker.com/get-docker/>

[5] <https://github.com/bkeepers/dotenv>

[6] <https://github.com/rack/rack>

Chapter 2

Handle Failures by Planning Ahead

When a user is on your website or app and something goes wrong—even if it's not something you expected to fail—the user will often retry what they were doing. If the underlying problem was transient, say their internet connection momentarily dropped, the retry will succeed and all is well (outside of a minor annoyance).

Your Sidekiq jobs don't have a patient user hanging around willing to retry when things go wrong. Even if they did, there's no guarantee that retrying the jobs will fix any underlying causes for failure (of which there are many). Nevertheless, your code needs to run to completion and you have to have some strategy to deal with failures. That's what you'll learn about here.

You'll learn how to monitor the failures of our jobs, how to allow transient failures to automatically retry without over-notifying you, and how to make sure that every job eventually succeeds. But first, it's good to know why this matters at all.

Making Sure Jobs Succeed...Eventually

We write code to solve a problem or achieve some result. When that code is split up such that some of it runs while a user is waiting and some runs in a background job, it's hard to clearly understand what code ran by looking at the server logs. For example, if your users are successfully creating orders, but none of the `CompleteOrderJob` jobs are completing, you have a problem.

The biggest challenge in managing a system with Sidekiq jobs is making sure that the jobs all eventually complete. This requires knowing what jobs have failed, why they failed, and fixing anything needed to make them succeed when retried. For even a moderately sized system, this can be an overwhelming task, at least without some sort of automation.

There are three ways to deal with this: 1) ignore them and hope for the best, 2) manually examine each job failure to figure out what to do, or 3) devise and implement a strategy to manage only those jobs that require a developer to intervene. I've been on teams that have used each one of these and, if you can't guess already, you'll be learning about the third way to deal with this.

To develop a strategy to manage failed Sidekiq jobs, you'll need to first understand why jobs fail.

Understanding Why Jobs Fail

Failure is a bit vague, but in the context of Sidekiq jobs, it means any condition during which the execution of the job stops before completion. It might stop due to an exception, a bug, or a gopher chewing through the power cable at your data center.

Technically speaking, Sidekiq considers a job to have failed if it raises an exception that's not caught inside `perform`. In that case, Sidekiq will automatically retry the job over the course of 20 days using an increasingly exponential delay between each retry. If, after that time, the job is still failing, Sidekiq considers it *dead* and will stop retrying it (this can be configured per job but isn't recommended).

There are two categories of failures, and you'll learn a strategy for handling both. *Permanent* failures will never succeed no matter how many times you retry them. Bugs in the code are a common example of such a failure.

Transient failures are failures that, if retried later, could succeed. Network connectivity is a common example. When the network is restored, the job can be retried and should complete successfully.

Handling Permanent Failures via Monitoring

If a job has failed and will never succeed, it stands to reason that the only way to fix the underlying problem is first to become aware of it. An extremely common way to do this is to configure an *error catching service* (or *error catcher*), like Honeybadger^[7] or Bugsnag.^[8]

An error catcher is a service that integrates with your app to receive notifications of any unhandled exception, including those from your Sidekiq jobs. The error catcher then notifies you about the exception via email or an alerting system like Pager Duty. The idea is that you don't have to proactively check for failed jobs; instead, you allow the error catcher to notify you if a job fails. The example app you set up includes a mock error catcher so you can see how it works without having to sign up for a real service and manage that integration.

To connect Sidekiq to an error catcher, you'll need to configure an *error handler*. Sidekiq's configuration provides the attribute `error_handlers`, an array of `Proc` objects that are called when an unhandled error is caught. Our mock error catcher has a client library class included in the example app `ErrorCatcherServiceWrapper`, which has a method named `notify` that will send the exception to the service.

To set this up, you'll need to create an initializer for Sidekiq in `config/initializers/sidekiq.rb`. It will look like the following code, with the `Proc` configured for the error catcher:

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

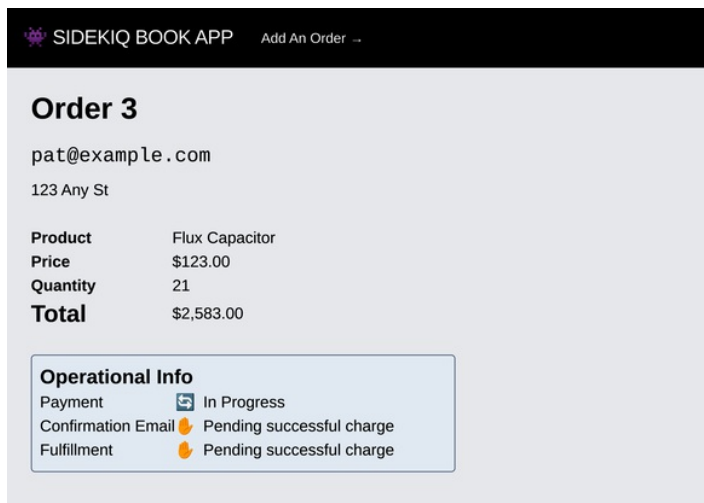
```
Sidekiq.configure_server do |config|
  config.error_handlers << ->(exception,context_hash) {
    ErrorCatcherServiceWrapper.new.notify(exception)
  }
end
```

Let's see this in action by introducing a permanent failure. Add a line of code to raise an exception at the start of `OrderCreator's complete_order` method, which is what `CompleteOrderJob` calls:

```
snapshots/2-1/sidekiq-book/app/services/order_creator.rb
```

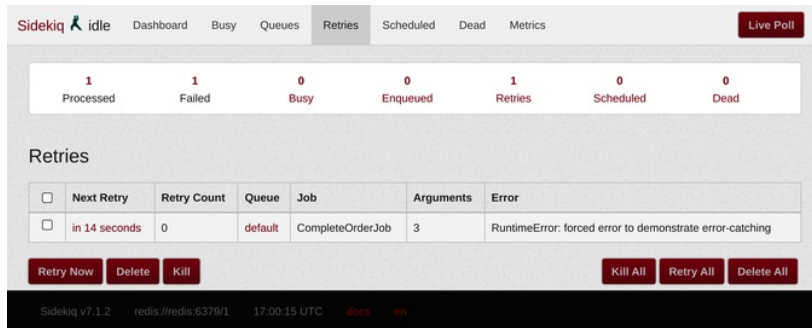
```
def complete_order(order)
  » raise 'forced error to demonstrate error-catching'
  payments_response = charge(order)
  if payments_response.success?
```

Now, create a new order using the example app. After doing so, you should be redirected to the new order's show page, but even after reloading that page, the order should still indicate that payment, email, and order fulfillment are in progress, as shown in the image [here](#).




A screenshot of the order show page that indicates that the payment, notification email, and order fulfillment request are all pending.

Now, Go to the Sidekiq Web UI at <http://localhost:3000/sidekiq> and click on "Retries." You should see that the `CompleteOrderJob` is there, along with the error message you added to `complete_order`. Sidekiq will also show you when it's planning on retrying the job, as shown in this screenshot:



A screenshot of the Sidekiq Web UI's Retries page. It shows a single job's class and error message, along with an indicator of when the job will be retried.

The mock error catcher has a Web UI you can examine to view the notifications it has received. You can see it at <http://localhost:3001>, and it should look like the shot below. If you see a notification here, a real error catcher would've notified you.

 **MOCK ERROR CATCHER**

Notifications

Time	Exception	Message
2023-07-03 17:00:10 +0000	RuntimeError	forced error to demonstrate error-catching

A screenshot of the mock error catcher that shows a table with entries for all the retry attempts. Each entry shows a timestamp, the exception class name, and the exception's message.

Now, undo the syntax error:

```
snapshots/2-2/sidekiq-book/app/services/order_creator.rb
```

```
def complete_order(order)
  » # removed forced error
  payments_response = charge(order)
  if payments_response.success?
```

Go back to the Retries section of the Sidekiq Web UI. If the job is still there waiting to be retried, click it, then click “Retry.” You should be returned to the Retries section and not see anything there—the job succeeded! If you refresh the order show page, you should see that all the operational stuff has completed properly.

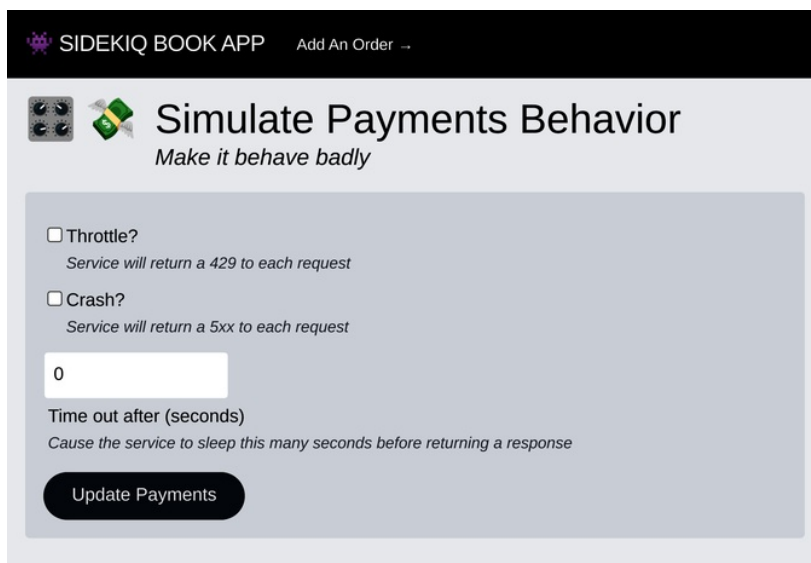
What’s important to take away here is that if you had not been notified about the job failure, you wouldn’t have known to fix the underlying problem and the job would never have succeeded. This mechanism of being notified of failure works extremely well...until you start getting transient failures, which are far more common.

Managing Transient Failures Through Smarter Code

Transient failures will trigger the error catcher notification we just set up. By their nature, transient failures eventually succeed, which means you will be notified of a failure that requires no intervention. The example app allows you to simulate a transient failure.

See Transient Failure and Recovery in Action

Navigate to the home page at <http://localhost:3000>, then click the triangle next to “Payments.” You should see some information about the service, as well as a link labeled “Manage...” Click that link. You should see the following form:



The screenshot shows a web interface for the 'SIDEKIQ BOOK APP'. At the top, there is a navigation bar with the app name and a link 'Add An Order ...'. Below this is a section titled 'Simulate Payments Behavior' with the subtitle 'Make it behave badly'. The form contains three checkboxes: 'Throttle?' (unchecked) with the description 'Service will return a 429 to each request', 'Crash?' (unchecked) with the description 'Service will return a 5xx to each request', and a text input field with the value '0' and the label 'Time out after (seconds)' with the description 'Cause the service to sleep this many seconds before returning a response'. At the bottom of the form is a button labeled 'Update Payments'.

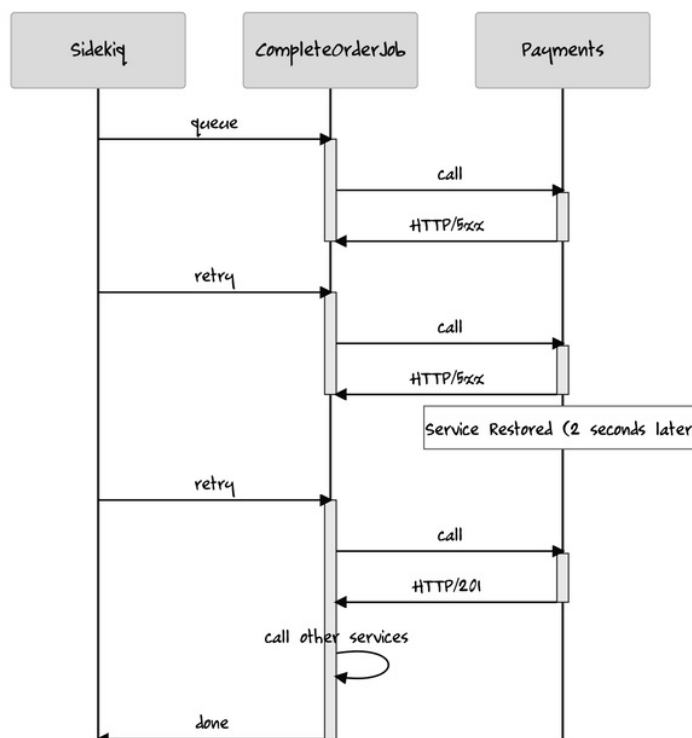
A screenshot showing a form to allow changing the behavior of the fake payments service. There are controls for throttling, crashing, and slowing down responses as well as a button labeled “Update Payments.”

Check the “Crash” checkbox, then click “Update Payments.” This will tell the payments service to return HTTP 5xx errors, and simulate a brief outage.

Create an order as you did before. The order won't complete, and you'll see a failure in Sidekiq's Web UI. This time, the exception will be `BaseServiceWrapper::HTTPError`, a class internal to the example app. It's thrown when any API call gets a 5xx HTTP status code.

Now, go back to the page where you managed the payments service, uncheck "Crash," then click "Update Payments." This will return the payments service to normal behavior, thus ending the brief simulated outage. Let the job retry on its own. In the Sidekiq Web UI, the Retries section will show you when the job will be retried, and hopefully it's no more than a few minutes. Wait that long, then reload the page. The job should be gone (since it retried and succeeded). If you refresh the order show page, it should show as completed.

This demonstrates that automatically retrying a transient error will allow the job to succeed without your intervention. The simulation may have felt manual, but imagine this entire scenario playing out over a few seconds. The following diagram shows the interaction:



A sequence diagram showing the `CompleteOrder` job calling the

payments service, getting a 5xx, being retried, getting a 5xx again, then being retried a third time, at which point it succeeds. The job then calls the other two services successfully.

Although the retry mechanism allows the job to succeed eventually, the error catcher still notifies you. Navigate to <http://localhost:3001/error-catcher/ui>, and you'll see the transient `BaseServiceWrapper::HTTPError` errors. This means you would be notified about these errors even though no action was required.

This is the core problem to solve when managing Sidekiq job failures. You don't need the error-catching service to notify you about a transient failure, but you *do* need to know about a permanent one.

Prevent Transient Errors from Notifying You

Most error catchers allow you to ignore particular errors. That means you can configure the error catcher to ignore `BaseServiceWrapper::HTTPError` and you won't get a notification whenever a job failed but was retried. This configuration would apply to more than just jobs, however. If you got a `BaseServiceWrapper::HTTPError` from, say, a controller, you wouldn't be notified about that either. That's not what you want. You only want to ignore certain errors from Sidekiq jobs. You want to be notified if an error happens outside a Sidekiq job.

A straightforward way to set this up is to create a special exception class that the error catcher will be configured to ignore, thrown only by a job that is experiencing a transient error. This means your job can rescue `BaseServiceWrapper::HTTP` and re-throw it as this new special exception. Let's call the exception something extremely obvious: `IgnorableExceptionSinceSidekiqWillRetry`. I won't pretend that's a beautiful class name, but it's certainly self-explanatory.

The code to translate the exception should go in the job code, like so:

```
snapshots/2-2/sidekiq-book/app/jobs/complete_order_job.rb
```

```
def perform(order_id)
  order = Order.find(order_id)
  OrderCreator.new.complete_order(order)
» rescue BaseServiceWrapper::HTTPError => ex
»   raise IgnorableExceptionSinceSidekiqWillRetry.new(ex)
end
end
```

You'll see the code for `IgnorableExceptionSinceSidekiqWillRetry` in a moment. Next, configure the error catcher to ignore this new exception. How you do this highly depends on the error catcher you have set up. Usually, you add the exception's class to some list in a configuration block. For the mock error catcher, add this line of code in `config/initializers/error_catcher.rb`:

```
snapshots/2-2/sidekiq-book/config/initializers/error_catcher.rb
```

```
Rails.application.config.to_prepare do
»   ErrorCatcherServiceWrapper.ignored_errors << IgnorableExceptionSinceSidekiqWillRetry
end
```

The last step is to implement the new exception. Since it's only relevant to Sidekiq jobs, put it in `app/jobs`:

```
snapshots/2-2/sidekiq-book/app/jobs/ignorable_exception_since_sidekiq_will_retry.rb
```

```
class IgnorableExceptionSinceSidekiqWillRetry < StandardError
  def initialize(exception)
    super(exception.message)
    @cause = exception
  end

  def cause = @cause
end
```

You'll need to restart your server to try this out; however, it will make this behavior easier to see if we reset the error catcher and clear out Redis as well. When you stop the server, run `bin/rails dev:reset`. This Rake task will reset the

fake API and Redis clean states. Then, start the server with [bin/dev](#) as usual.

Repeat the process of configuring the payments service to crash, creating an order, restoring the payments service, and allowing the [CompleteOrderJob](#) to complete. Once that happens, you should *not* see anything in the error catcher UI, thus demonstrating that you would not be notified of a failing job that will eventually succeed after retrying.

Notice that we've continued to separate concerns regarding the code. [OrderCreator](#) still only has code about creating and completing orders. All of the code to manage the logistics around job failure and notification is part of the [CompleteOrderJob](#) class—right where it belongs (we'll talk about simplifying this code across multiple jobs in [Organizing Sidekiq Job Code](#)).

Unfortunately, much like the cane toads of Australia,^{[\[9\]](#)} the solution to one problem can create a new problem. In this case, what if the payments service API call never succeeds?

Monitoring for Failed Jobs That Stop Retrying

Suppose our payments service deprecated the API call being used, and it now always returns an HTTP 503. That might not be how *you'd* handle API deprecation, but you can't control the payment processors of the world. In this case, what you've been treating as a transient error is now a permanent one. Luckily, you know how to handle permanent errors: monitor them. Unluckily, you just spent the last section disabling notifications about this particular error.

Sidekiq provides a solution called the *dead set*. The dead set is a set of jobs that have been retried too many times and have been set aside, never to be retried again. The default setting for `max_retries` results in a job being moved to the dead set after about 20 days. When that happens, you can use the Web UI to retry the job, as long as you know to look there.

To ensure you are notified when a job moves to the dead set, Sidekiq allows the configuration of a `death handler`. This works much like error handlers, in that you provide a `Proc` that will be called. Configure this using the `death_handlers` attribute on the Sidekiq server's `config`:

snapshots/2-2/sidekiq-book/config/initializers/sidekiq.rb

```
Sidekiq.configure_server do |config|
  » config.death_handlers << ->(job,exception) {
  »   ErrorCatcherServiceWrapper.new.notify(
  »     "#{job['class']} won't be retried: #{exception.message}"
  »   )
  » }
  config.error_handlers << ->(exception,context_hash) {
    ErrorCatcherServiceWrapper.new.notify(exception)
  }
}
```

To see this in action without waiting twenty days, you can temporarily change the Sidekiq configuration to only retry jobs once before moving them

to the dead set. Do this by adding `max_retries` to `config/sidekiq.yml`:

```
:concurrency: <%= ENV.fetch("SIDEKIQ_CONCURRENCY") { 5 } %>
:timeout: <%= ENV.fetch("SIDEKIQ_TIMEOUT_SECONDS") { 25 } %>
» :max_retries: 1
:queues:
  - default
```

Restart your server, then repeat everything. This time, you should see a notification in the error catcher UI with the message you put in the death handler. If you go to the Sidekiq Web UI, click on “Dead,” and you should see the job. This demonstrates that a transient failure will eventually notify you if the job never succeeds.

Don’t forget to remove `max_retries` from your `config/sidekiq.yml`.

The strategy you just implemented—notifying yourself about permanent errors, avoiding notification for transient ones, and then ensuring dead jobs notify you no matter what—forms a solid foundation for managing Sidekiq jobs. This strategy ensures that all jobs eventually succeed without overloading you with non-actionable notifications. Over time, you’ll tweak this setup as you learn which sorts of failures are transient and which are not.

It’s worth pointing out that you really can’t avoid having to handle failures in some way. No paid service can alleviate you of this need, since addressing failure requires understanding the specifics of your app and your jobs. Anyone who tells you different is selling something.

Up Next

Failure is never this simple. What if *part* of the job succeeds, but part of it fails? When the job gets automatically retried, the *entire* job will be re-executed, and this may not be what you want. To allow our error handling strategy to work, jobs must be absolutely safe to retry. They must be idempotent.

Footnotes

[7] <https://www.honeybadger.io>

[8] <https://www.bugsnag.com>

[9] https://en.wikipedia.org/wiki/Cane_toads_in_Australia

Chapter 3

Safely Retry Jobs by Making Them Idempotent

Putting flaky code in a Sidekiq job is a great way to have it automatically fix itself by retrying. But is all code safe to just retry? What if we charged the user some money, but failed to send them their confirmation email? We'd retry the job and charge them again! In this chapter, we'll learn about idempotence, a concept we have to apply to make jobs safe to retry. It's complicated, but we'll also learn a few practical techniques for making jobs idempotent.

Retrying Failed Jobs Creates New Failures

Let's see how retrying a job can cause a different failure. If you recall, the example app's services can cause problems that trigger retries in the example app. In the last chapter, we configured the payments service to crash on every call. Let's do the same thing, but for the email service.

As a quick reminder, here's how to do that:

1. Go to the example app's home page at <http://localhost:4000>.
2. Click on the triangle next to the email service to expand its info. Click the link labeled "Manage..."
3. On the new screen, check the box for "Crash," then click the button "Update Email."

At this point, every email service request will return a 5xx, which will cause our `CompleteOrderJob` to fail. Go ahead and create an order. As before, you should see the failed `CompleteOrderJob` in the retry section of the Sidekiq Web UI.

Wait for the job to retry a few times. The fake payments service has a UI where you can view all payments it has charged. Visit it at <http://localhost:3001/payments/ui> and you'll see more than one charge for this order, as in the screenshot that follows.



PAYMENTS

Charges

Customer Id	Amount Cents	Status	Charge Id	Metadata
cust_0e184ba713f59526	258300	success	ch_63d74d53	{"order_id"=>6}
cust_0e184ba713f59526	258300	success	ch_38babffb	{"order_id"=>6}

A screenshot of the fake payments UI showing two or more charges for the same order.

When the email service is restored, you'll find that the order has been charged for each retry. You want the order charged only once, even if there was a transient failure with sending the email. In fact, you want the entire job's effects to be had exactly once, even if the job is retried multiple times. This is called *idempotence*.

Understanding Idempotence

Idempotence can be a difficult concept to grasp, and it can be equally difficult to determine that a routine or job is idempotent. The key is to ask yourself what will happen if the method is called more than once with the same arguments.

Here is a simplified example, where a job is adding more quantity to a product:

```
class AddMoreProductsJob
  include Sidekiq::Job
  def perform(product_id, num_new_products)
    product = Product.find(product_id)
    product.update!(
      quantity_remaining: product.quantity_remaining + num_new_products
    )
  end
end
```

If the job gets retried for some reason, even if every line of code has executed, it will have added twice the number of products as intended. That means it's not idempotent. You could make it idempotent by removing the calculation and passing in the final value:

```
class AddMoreProductsJob
  include Sidekiq::Job
  def perform(product_id, new_quantity_remaining)
    product = Product.find(product_id)
    product.update!(
      quantity_remaining: new_quantity_remaining
    )
  end
end
```

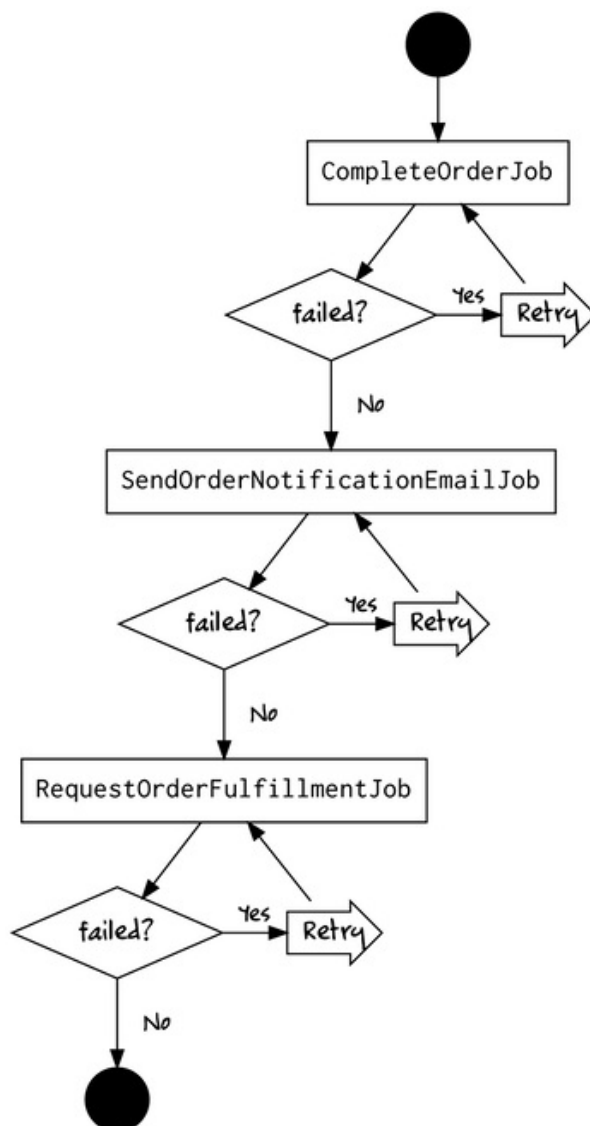
The job can be safely retried and is idempotent. However, it's less useful because the calculation for the correct value of `quantity_remaining` has to be done

somewhere else. That “somewhere else” is likely not idempotent. While no system can be completely idempotent, critical parts of it can be, and Sidekiq jobs are one such part.

Back to [CompleteOrderJob](#), it’s clearly not idempotent. One reason is the one you just experienced, where the entire job was retried, even though part of it was successful before the first failure. We can address this issue by breaking up the job into smaller jobs that only retry the specific part that failed.

Breaking Down Large Jobs into Safely Retriable Parts

`CompleteOrderJob` has three steps: charge the user, send an email, and request fulfillment. If each step were its own job, the failure mode you simulated above would not result in multiple charges for the order. Let's set that up by changing `CompleteOrderJob` to queue a new job called `SendOrderNotificationEmailJob` that will then queue another new job named `RequestOrderFulfillmentJob`. As you can see from the image [here](#), this should solve the problem.



A flowchart showing the `CompleteOrderJob` queuing the job `SendOrderNotificationEmailJob`, which then queues the job `RequestOrderFulfillmentJob`. If `CompleteOrderJob` results in a decline, it skips the other two jobs.

To make this change, you'll need two new jobs that will call two new methods on `OrderCreator`. The first job is `SendOrderNotificationEmailJob`, which goes in `app/jobs/send_order_notification_email_job.rb` and calls `send_notification_email`.

```
snapshots/3-1/sidekiq-book/app/jobs/send_order_notification_email_job.rb
```

```
class SendOrderNotificationEmailJob
  include Sidekiq::Job
  def perform(order_id)
    order = Order.find(order_id)
    OrderCreator.new.send_notification_email(order)
  rescue BaseServiceWrapper::HTTPError => ex
    raise IgnorableExceptionSinceSidekiqWillRetry.new(ex)
  end
end
```

Next, create `RequestOrderFulfillmentJob` in `app/jobs/request_order_fulfillment_job.rb`, which will call `request_order_fulfillment`.

```
snapshots/3-1/sidekiq-book/app/jobs/request_order_fulfillment_job.rb
```

```
class RequestOrderFulfillmentJob
  include Sidekiq::Job
  def perform(order_id)
    order = Order.find(order_id)
    OrderCreator.new.request_order_fulfillment(order)
  rescue BaseServiceWrapper::HTTPError => ex
    raise IgnorableExceptionSinceSidekiqWillRetry.new(ex)
  end
end
```

Next, add `send_notification_email` and `request_order_fulfillment` to `OrderCreator`:

```
snapshots/3-1/sidekiq-book/app/services/order_creator.rb
```

```

    )
  end
end
» def send_notification_email(order)
»   email_response = send_email(order)
»   order.update!(email_id: email_response.email_id)
»   RequestOrderFulfillmentJob.perform_async(order.id)
» end
»
» def request_order_fulfillment(order)
»   fulfillment_response = request_fulfillment(order)
»   order.update!(
»     fulfillment_request_id: fulfillment_response.request_id
»   )
» end
»
private

def charge(order)

```

Make Small Changes



Note that the code still waits for the email to be sent before requesting fulfillment, even though these operations are unrelated. You can optimize this later, but it's often easier to make as few changes to the system as possible at one time. Leaving the order of operations will result in a system that works just as it did before, which maintains everyone's mental model of the system.

With the code extracted, you should now remove the code from `complete_order` and replace it with a call to queue `SendOrderNotificationEmailJob`:

```

    payments_response = charge(order)
    # XXX   if payments_response.success?
    # XXX
    # XXX   email_response    = send_email(order)
    # XXX   fulfillment_response = request_fulfillment(order)
    # XXX
    # XXX   order.update!(
    # XXX     charge_id: payments_response.charge_id,
    # XXX     charge_completed_at: Time.zone.now,
    # XXX     charge_successful: true,
    # XXX     email_id: email_response.email_id,
    # XXX     fulfillment_request_id: fulfillment_response.request_id)
  »   if payments_response.success?
  »     order.update!(
  »       charge_id: payments_response.charge_id,
  »       charge_completed_at: Time.zone.now,
  »       charge_successful: true,
  »     )
  »     SendOrderNotificationEmailJob.perform_async(order.id)
  else
    order.update!(
      charge_completed_at: Time.zone.now,

```

Repeat the order creation process you did at the start of the chapter. Before doing that, run `bin/rails dev:reset` and restart your server so you have a clean slate.

If you allow the email to fail multiple times as before, you'll see that the order was only charged once:



PAYMENTS

Charges

Customer Id	Amount Cents	Status	Charge Id	Metadata
cust_0e184ba713f59526	258300	success	ch_3df8a678	{"order_id"=>7}

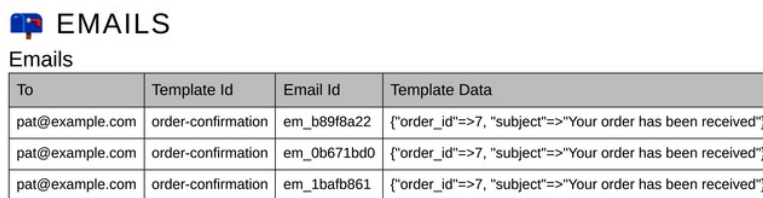
A screenshot of the fake payments UI showing only one charge for the order.

This is a great improvement, but there's still another potential issue. What if, each time the call to the email service failed, the email service actually sent the email, but returned an error to the app? This is definitely possible, and there isn't a good way to detect that this happened. It also means that our new [SendOrderNotificationEmailJob](#) is not idempotent!

To fix this issue, let's look at another technique, which is to lean on a third party to fill in missing details when deciding what to do with a failed job.

Addressing Idempotence Issues with Third Parties

The fake email service behaves in the way described above: when you ask it to crash, it's configured to send an email first and *then* crash. If you navigate to its UI at <http://localhost:3001/email/ui>, you'll see that there are multiple emails to the same person for the same order:



The screenshot shows a table titled 'EMAILS' with a sub-header 'Emails'. The table has four columns: 'To', 'Template Id', 'Email Id', and 'Template Data'. It contains three rows of data, all representing emails sent to 'pat@example.com' for an 'order-confirmation' template. The 'Email Id' values are 'em_b89f8a22', 'em_0b671bd0', and 'em_1bafb861'. The 'Template Data' for all rows is '{"order_id"=>7, "subject"=>"Your order has been received"}'.

To	Template Id	Email Id	Template Data
pat@example.com	order-confirmation	em_b89f8a22	{"order_id"=>7, "subject"=>"Your order has been received"}
pat@example.com	order-confirmation	em_0b671bd0	{"order_id"=>7, "subject"=>"Your order has been received"}
pat@example.com	order-confirmation	em_1bafb861	{"order_id"=>7, "subject"=>"Your order has been received"}

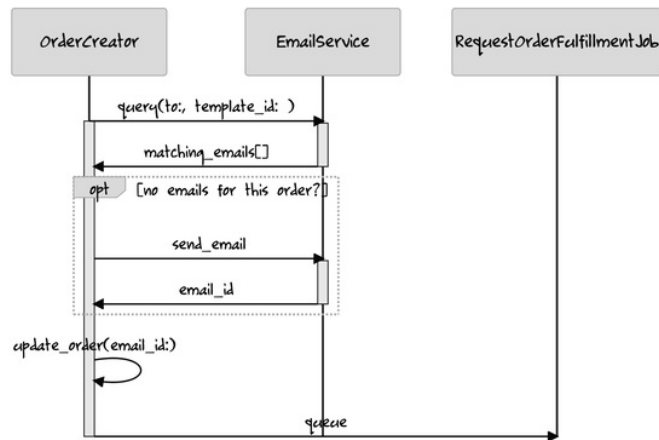
A screenshot of the fake email UI showing multiple emails sent to the same person for the same order.

This is harder to recover from: the part of our system that is told about email delivery is the part that's failing. You can't know if the email was sent—only the email service knows that. There are two common ways to manage this type of failure. One is to ask the service if the operation completed.

Use Third Party Services APIs to Figure Out What Happened

Most third-party services allow you to query the service to find out what operations have been previously performed. The fake email service provides a way to do this, and you can use that to figure out if an email you are about to send has already been sent.

In the case of the example app, only one email confirmation should be sent per order. Thus, the combination of email address, `template_id` and order ID should be unique. If you query the email service and it returns a sent email with those attributes, you can be sure the email you want to send has been sent already, and skip it. If no result is returned, trigger the service to send it. See the diagram that follows.



A diagram showing a check against the email service for matching emails, then an optional area where an email is sent if no matches were found. After that, the email ID is saved and the [RequestOrderFulfillmentJob](#) is queued.

The fake email service provides a way to search by email address and email template ID. You'll have to look through the returned array of emails for the order ID in the email metadata. Here's what that looks like in the [send_notification_email](#) method of [OrderCreator](#):

snapshots/3-3/sidekiq-book/app/services/order_creator.rb

```

end
end
def send_notification_email(order)
  » potential_matching_emails = email.search_emails(
  »   order.email,
  »   CONFIRMATION_EMAIL_TEMPLATE_ID
  » )
  » email_response = potential_matching_emails.detect { |email|
  »   email.template_data["order_id"] == order.id
  » }
  » if email_response.nil?
  »   email_response = send_email(order)
  » end
  » order.update!(email_id: email_response.email_id)
  » RequestOrderFulfillmentJob.perform_async(order.id)
end

```

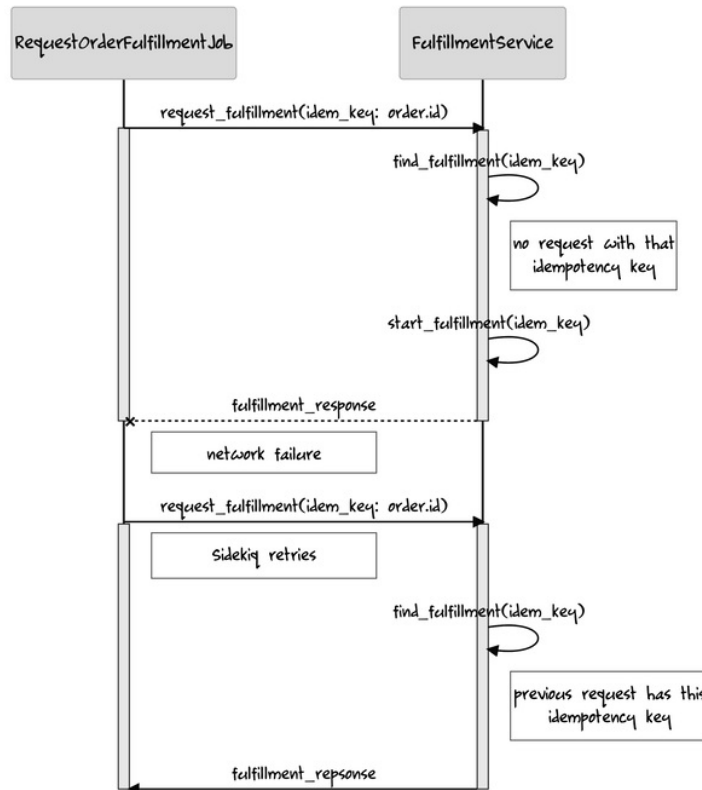
Code like this can feel a bit backward. The first thing it does is to check to see if what it's about to do has been done before. Like being a character in a sci-fi time travel story, you have to think fourth-dimensionally: what might have happened the first time so that the *second* time this code runs, it does the right thing?

Fortunately, more and more third parties provide an easier way to address this issue by allowing you to specify that an operation should be treated as idempotent on the third party's side using a special key.

Use an Idempotency Key if the Service Provides One

The problem you just solved with the email service also exists with the fulfillment service. Although the fake fulfillment service won't fulfill orders when you configure it to crash, you can imagine that in certain situations in the real world (such as a network outage), a retried job could send an order out more than once.

Unlike the email service, the fulfillment service allows you to specify an *idempotency key*. The service will guarantee that it will perform only one request with that key. If it gets a second request with the same key, it won't duplicate the work, but will return as if it had. The fulfillment service handles all the fourth-dimensional thinking and management for you, as outlined in the following diagram:



A sequence diagram showing a failed call to the fulfillment service. That call includes an idempotency key that the service remembers. When the job is retried, the same idempotency key is used, and the service locates the previous response and returns it, instead of re-executing the API call.

The trick when using idempotency keys is to choose one that truly represents a unique operation. If we always use the key “1,” then we’d never send out new orders to new customers. If we use a key based on the current time, a retried request would get a new key, and we’d send out more than one order.

In the case of the example app, no order should be fulfilled more than once. That makes the order’s primary key a good candidate for an idempotency key. Thinking through what the key should be is not always easy. Imagine if orders *could* be fulfilled more than once: the order ID would not be sufficient. You’d likely need some new concept like a `fulfillment_requests` table that tracks each fulfillment, and then use *that* table’s ID as the idempotency key.

One consideration for the choosing the value of the key is where you might encounter it out of context. For example, you may see it in a web UI or a log file as you try to debug why the system did something wrong. If your key is a database identifier (a number or a UUID), it'll be unremarkable and hard to distinguish.

To deal with this issue, add some context directly to the key itself. For example, if an order has the ID 1234, an idempotency key like `"idempotency_key-order-1234"` would be very obvious in log files, web UIs, and more. If you can't do this, or have some other constraints on what the key should be, you should store the value in the database along with the record. This will provide an unambiguous accounting of what key was used.

Let's add an idempotency key to `request_fulfillment`. The fulfillment service will look for it in the metadata passed in the API call:

```
snapshots/3-4/sidekiq-book/app/services/order_creator.rb
```

```
def request_fulfillment(order)
  fulfillment_metadata = {}
  fulfillment_metadata[:order_id] = order.id
  » fulfillment_metadata[:idempotency_key] = "idempotency_key-order-#{order.id}"
  fulfillment.request_fulfillment(
    order.user.id,
    order.address,
```

You can see the advantage when the service supports idempotency keys: this was a one-line change. You also didn't have to consult old Star Trek episodes to help train your brain to think fourth-dimensionally, either!

You addressed the email service and the fulfillment service, but the payments service has the same potential failure modes. As an exercise for you, try to make that call idempotent. Have a look at `PaymentsServiceWrapper` to see what options you have to fix it.

Preventing Retries if You Cannot Make the Call Idempotent

There's almost always a way to isolate critical code into an idempotent routine, and usually the two techniques we just discussed will let you do it. If there isn't, and retrying the job is unsafe, you'll need to prevent the job from retrying. You can do this on a per-job basis by calling `sidekiq_options` in the job, like so:

```
class JobThatCannotBeIdempotent
  include Sidekiq::Job
  sidekiq_options retry: 0

  def perform(...)
    # ...
  end
end
```

The first time that job fails, it'll go to the dead set. Your configured death handler will alert you, and you can take whatever manual action you need to decide what to do.

We'll talk more about thinking through idempotency in a bit, but before we do that, we should make sure we understand how to, or if we even can, test all this.

Testing Code for Idempotency

There's no general way to test a piece of code to determine if it's idempotent. It's usually not possible to have all of your third-party service providers simulate error conditions, so testing idempotent code often involves testing that it behaves the way you intended, given the conditions you wrote it for.

The changes to `send_notification_email` were substantial. You added a new API call and complex logic to examine that call to figure out what to do. The example app doesn't use mocking but since it does have the fake mail service running, you can use it to simulate the situation you're checking for.

The core of the test will call `send_notification_email`, and will expect a previously-used `email_id` to be set on the order. The test should also expect that no email has been sent from the email service.

To test both of those assertions, you'll need to trigger an email prior to calling the test by using `EmailServiceWrapper` directly. You'll also need to have it query for sent emails so you can count the number of emails both before and after the test.

It's a bit complicated, but here's what it will look like:

```
snapshots/3-4/sidekiq-book/test/services/order_creator_test.rb
```

```
    refute_nil resulting_order.fulfillment_request_id
  end

  » test "send_notification_email uses existing email" do
  »   email_service_wrapper = EmailServiceWrapper.new
  »
  »   order = Order.create!(
  »     email: "pat@example.com",
  »     address: "123 Main St",
  »     quantity: 1,
  »     product: create(:product),
```

```

»   user: create(:user),
»   )
»
»   # Pretend the email was sent in a previous execution
»   previously_sent_email = email_service_wrapper.send_email(
»     order.email,
»     OrderCreator::CONFIRMATION_EMAIL_TEMPLATE_ID,
»     {
»       order_id: order.id
»     }
»   )
»   email_id = previously_sent_email.email_id
»   refute_nil email_id, "expected an email to have been send"
»
»   # Grab a count of the emails matching before running the test
»   matching_emails = email_service_wrapper.search_emails(
»     order.email,
»     OrderCreator::CONFIRMATION_EMAIL_TEMPLATE_ID
»   )
»   num_matching_emails = matching_emails.count
»
»   # Test starts here
»   @order_creator.send_notification_email(order)
»   order.reload
»
»   assert_equal email_id, order.email_id,
»     "A different email was sent than the previously-sent one"
»
»   # Re-fetch the emails so we can count them
»   matching_emails = email_service_wrapper.search_emails(
»     order.email,
»     OrderCreator::CONFIRMATION_EMAIL_TEMPLATE_ID
»   )
»   assert_equal num_matching_emails, matching_emails.count,
»     "An email was sent that should have been"
»   end
end

```

You can validate that it's testing what you think by removing the email service query from `send_notification_email` and re-running the test. It will fail.

The change for the fulfillment service is a different story. It was a one-line

change with no logic. If it was easy enough to cover this with a test, it may be worth it, but as it stands, adding coverage is more trouble than it's worth. We would need to set up a potentially complex mocking situation just to assert that we added a value to a hash.

Not all tests are worth writing, and this is a good example. If mocking were already set up, or the third-party service provided an easy way to check that we had used an idempotency key, it would be worth it, but as it stands, we should skip this one.

Before leaving this chapter, let's go over a more generalized way to evaluate code for idempotency issues. It won't save you from having to think through business rules and to pretend to be a time traveler, but it can provide a structure to examine an entire piece of code.

Assessing Code for Idempotency

It's extremely hard to assert that a piece of code is idempotent. What you want to do is examine the code triggered from a Sidekiq job and ask yourself what happens when that code fails and retries. A process that helps do this systematically is as follows:

1. For each line of code, N , mentally execute the code from line 1 to N , and then from line 1 to the end of the routine. Don't worry about *why* the code stops at line N —just assume a gopher chewed through the server's power cable and everything just stopped.
2. For each N , determine the state of the system after completing the first step.
3. If the state of the system is unacceptable, you've identified a likely issue when the Sidekiq job retries after failure.
4. Repeat for all lines of code. Don't forget to perform this exercise when executing the entire routine twice—failures can happen after your code has run that can trigger a retry!
5. For each unacceptable outcome, rework the code to avoid it using the techniques discussed in this chapter.

This isn't foolproof, but it provides a relatively repeatable and teachable method to at least think through how the code will behave in the face of failure.

Using Database Transactions Can Help... Sometimes

If you are experienced with SQL-based databases, you may be wondering where database transactions fit in. If you aren't, a database transaction is a useful tool related to idempotence. A database transaction allows you to make many changes to a database and then apply them all at once. If anything should go wrong attempting to apply the changes, *none* of the changes are made. It's extremely handy, though it can't help as often as you might like and can have unintended performance consequences if you aren't careful.

As an example of how it works, suppose we want to reduce the hypothetical `quantity_remaining` of the product being ordered. You might do this by writing this code:

```
product.update!(quantity_remaining: product.quantity_remaining - 1)
order.update!(charge_id: payments_response.charge_id,
               charge_completed_at: Time.zone.now,
               charge_successful: true)
```

If something went wrong during the call to `order.update!`, the `quantity_remaining` would've been decremented, but the order not marked as charged. Upon retrying this code, the product quantity would be one less than the actual quantity after the order was fulfilled.

If, instead, you wrapped this code inside a call to `ActiveRecord::Base.transaction`, that failure could be eliminated:

```
ActiveRecord::Base.transaction do
  product.update!(quantity_remaining: product.quantity_remaining - 1)
  order.update!(charge_id: payments_response.charge_id,
                 charge_completed_at: Time.zone.now,
                 charge_successful: true)
end
```

If the same failure happens, the change to the `products` table won't be made. This makes this code idempotent in a certain context—it can be safely retried until the last line executes, when the transaction is said to have *committed*. Once the transaction commits, if the entire thing is retried, `quantity_remaining` would be too low and the value for `charge_completed_at` would be incorrect.

Despite this subtlety, it's still a good idea to wrap multiple calls to the database inside a transaction. As you can see, it does eliminate certain problems from happening. There are some caveats, however.

When you open a transaction, the database creates locks on the tables or rows being updated (depending the specifics of what is being changed). This can cause other jobs and requests to block or even timeout, triggering your error catcher. If this happens a lot, you can have a cascading failure across the entire system.

A common cause of this problem that many developers think is clever (myself previously included) is to put code to queue a job inside the transaction block. The thinking is that if there is a problem communicating with Redis or otherwise queuing the job, it will trigger a rollback of any database changes, making the code safe to retry.

```
ActiveRecord::Base.transaction do
  product.update!(quantity_remaining: product.quantity_remaining - 1)
  order.update!(charge_id: payments_response.charge_id,
                charge_completed_at: Time.zone.now,
                charge_successful: true)
  » SendOrderNotificationEmailJob.perform_async(order.id)
end
```

This will hold all the locks created by `ActiveRecord::Base.transaction` open while Sidekiq communicates with Redis. This might not seem like a long time, but it can sometimes be very long, especially if you are in a cloud-hosted environment. The reason I no longer do this—and strongly recommend you don't, either—is that I have seen this happen based on code I wrote or

encouraged others to write.

The system started to slow down drastically and we saw a huge spike in errors that we could not understand. The database had a high number of locks and timeouts. Nothing we saw pointed to this pattern of queuing jobs inside a database transaction. After I and several engineers spent many hours poring through logs and code, we found the culprit. Moving the calls to queue jobs to outside the transaction made the problem go away immediately.

Don't put non-database code inside a transaction if you can help it.

Up Next

If you stop here, you've got the tools you need to create well-performing, fault-tolerant code in production. But there's more to Sidekiq than handling failures and keeping yourself from being inundated with job failure alerts. In the next section, we'll hit several additional topics, such as queue management, concurrency, performance monitoring, code organization, and more.

Chapter 4

Sustainable Operations and Development

Now that you know how to manage failed jobs and write jobs that are safe to retry automatically, it's time to dive into some more long-term operational concerns. We'll start with considerations around performance monitoring and how Sidekiq can help you understand your system.

We'll then talk about two techniques—concurrency and priority queues—that are part of Sidekiq that can help improve performance. Finally, we'll go over some more advanced code organization concerns you'll need as you use Sidekiq more and more.

Monitoring Sidekiq Performance

Web application performance is much deeper than we have space for in this book. Managing it depends on the tools you are using. Nevertheless, you need a foundation on what to measure and how to think about it. In the context of Sidekiq, there are four levels where you want to be able to observe the behavior of your app, starting from highest level to lowest level:

1. *Business Function*—This is an operation that our app exists to perform (in our case, creating orders). How many are created over time? How long does it take? How long did the user wait?
2. *Job Performance*—This is how specific jobs are performing. How many jobs were queued of each type? How long did they take to complete? How many failed and were retried?
3. *Sidekiq Performance*—This is about the Sidekiq subsystem as a whole. How big are the queues? How quickly are they draining?
4. *Redis*—This is the database that underpins Sidekiq. How much space is left? How many connections are being used? What is the latency between our app and Redis?

You want to *monitor* all of these layers, meaning you want to be able to consult some system to answer questions like, “How many connections were there to Redis between 5:45 and 6:13 p.m. on Monday the 4th?” or “How many orders were completed after at least one credit card decline last month?” *Alerting* on these measures is a different story, and we’ll talk about that in a bit.

In almost every case, you will need a third-party tool like Honeycomb^[10] or New Relic^[11] for performance monitoring. While these tools cannot automatically measure your business functions, they almost always provide

out-of-the-box measurements for everything else. Despite this convenience, it's still good to understand how you might access performance information about Sidekiq without these tools.

Sidekiq's Web UI Provides Observability at a Glance

You saw the Sidekiq Web UI earlier when learning about job failure and retries. The Web UI provides a lot more useful insights. Starting with Sidekiq 7, a “Metrics” tab can show you information about recent job executions. Sidekiq will retain these metrics for only eight hours, but it can provide a quick overview of how your jobs are doing at the moment. This is a great starting place for diagnosing a problem or ensuring the system is working properly.

At the volumes you've had in the example app, this tab won't be very interesting, but if you have about 20 minutes, you can populate it with data like I did in the screenshots that follow. The most straightforward way to do this is to create Rake task and use Factory Bot (included in the example app for creating test data) to create fake orders and products. [\[12\]](#)

Create a Rake task in `lib/tasks/dev/load_sidekiq.rake` that looks like so:

```
require "factory_bot"

namespace :dev do
  desc "Load a bunch of jobs into Sidekiq"
  task :load_sidekiq, :environment => do
    products = 10.times.map { FactoryBot.create(:product) }
    1_000.times do |i|
      order = FactoryBot.create(:order, product: products.sample)
      CompleteOrderJob.perform_async(order.id)
      sleep(rand * 2)
    end
  end
end
```

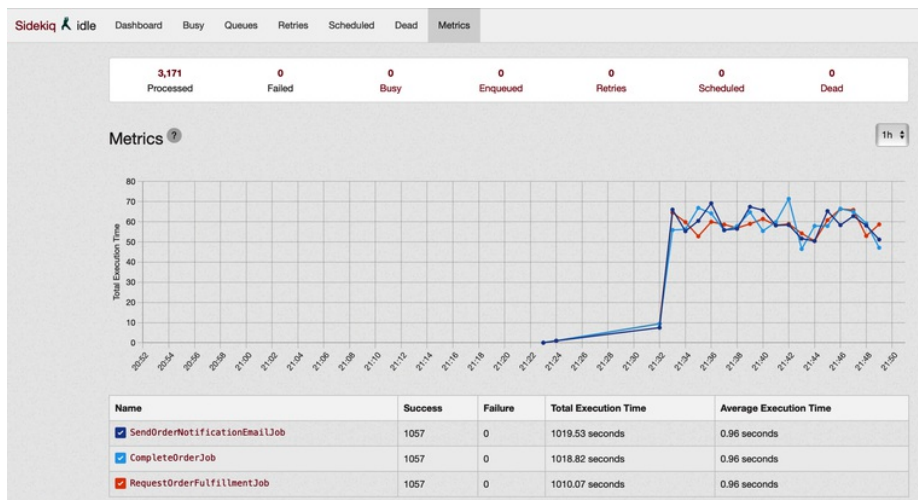
To create more realistic job behavior, you should force your jobs to sleep

randomly. For example, here is how to arrange for `CompleteOrderJob` to sleep some fractional number of seconds between 0 and 2:

```
class CompleteOrderJob
  include Sidekiq::Job

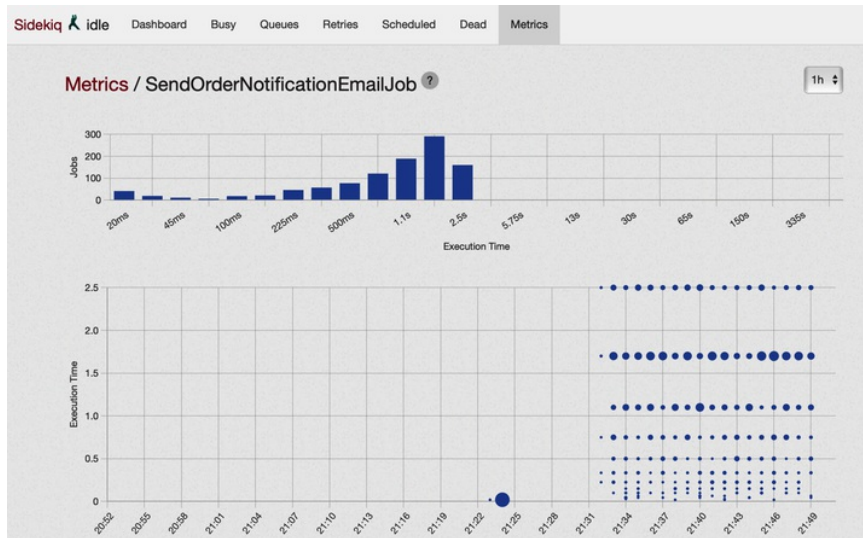
  def perform(order_id)
    » sleep (rand * 2.0)
    order = Order.find(order_id)
    OrderCreator.new.complete_order(order)
  rescue BaseServiceWrapper::HTTPError => ex
    raise IgnorableExceptionSinceSidekiqWillRetry.new(ex)
  end
end
```

Once you've done this to the other jobs, run `bin/rails dev:load_sidekiq`. After about 20 minutes, you should have a more interesting metrics tab like the one below:



A screenshot of the Sidekiq Web UI that shows a line graph with multiple lines. Each line represents one of the jobs and it shows the average execution for each job over time, with a somewhat random distribution. A table shows each job, the number of times that job ran, the total execution time, and the average execution time.

If you click one of the job names, you can see more detail about that job's execution, as in the next screenshot:



A screenshot of the Sidekiq Web UI that shows a bar graph showing a distribution of jobs by execution time, with most jobs taking 1.5 seconds, but a wide array of timings. There is also a two-dimensional dot graph. The X-axis is a timestamp and the Y-axis is an execution time. The size of the dot represents the number of jobs that executed in that time. The distribution is somewhat uniform because of the use of `rand`.

In a real-world app, this tab can be invaluable to give you an overview of how your Sidekiq installation is behaving. What it can't do is provide any sort of automated alerting. While a commercial monitoring tool *can* do that, it's worth understanding what you can do on your own, with just the API provided by Sidekiq.

Sidekiq's Internal API Provides Scriptable Observability

Sidekiq has an internal—but documented and supported—API you can use to provide basic monitoring and alerting without having a third-party monitoring system in place. Even if you have one, basic self-contained alerting can give you some piece of mind that you will know if anything is going wrong.

The approach to using Sidekiq's API for alerting and monitoring is to create

code that checks what you want, then notifies you if something is wrong. Let's create a Rake task that checks if Sidekiq is running by using `Sidekiq::ProcessSet`. This class returns a set of objects representing each Sidekiq process. If the set is empty...you have problems.

The Rake task will be `production:sidekiq:processes_check`, located in `lib/tasks/production/sidekiq/processes_check.rake`. It will check the `size` of the Sidekiq process set and, if it's 0, trigger the error catcher like we did in [Handling Permanent Failures via Monitoring](#).

```
snapshots/4-1/sidekiq-book/lib/tasks/production/sidekiq/processes_check.rake
```

```
namespace :production do
  namespace :sidekiq do
    desc "Alert if Sidekiq isn't running"
    task :processes_check, :environment do
      process_set = Sidekiq::ProcessSet.new
      if process_set.size == 0
        ErrorCatcherServiceWrapper.new.notify("Sidekiq is not running!")
      end
    end
  end
end
```

You can call this with `bin/rails production:sidekiq:processes_check`. Depending on how your app is hosted, you will need to set up something that calls it regularly. This command can be called frequently and—given what it's checking for, that's a good thing—you want to know as soon as possible if Sidekiq has stopped running.

If you don't have any sort of scheduling feature in your production environment, you can create a `bash` script to call this in a loop as a last resort:

```
#!/bin/sh

while true
do
  bin/rails production:sidekiq:processes_check
```



```
sleep 5  
done
```

(Hopefully, you have something more sophisticated and reliable.)

The Sidekiq API provides many other features to understand your Sidekiq process.^[13] You can grab counts of jobs, failures, and more. Instead of triggering the error catcher, you could also email yourself, or even log statistics to another system.

Note that if you *do* have an existing monitoring system, the Sidekiq Wiki documents several means of integration.^[14] Sidekiq Pro^[15]—which is modestly priced add-on to the open source Sidekiq—provides a generic integration with statsd,^[16] which is a somewhat standard way to get metrics into other systems.

One thing to be aware of is that the Sidekiq API uses Redis and not every Redis command is free.

Learn How to Understand the Cost of Calling Sidekiq API Methods

Sidekiq' API ultimately makes one or more calls to Redis, and these calls won't always return instantly. If you are going to build up a collection of scripts to monitor Sidekiq using its API, you'll need to know how to gauge the cost of those calls. Fortunately, Sidekiq's source code is easy to follow and Redis' documentation is excellent in this regard.

The vast majority of the Sidekiq API implementation is in `lib/sidekiq/api.rb` in Sidekiq's source code.^[17] You'll note that each public method makes one or more Redis calls. Usually, a variable named `conn` or `pipeline` is an instance of a Redis wrapper and methods called on those variables are Redis commands. For example, the call to `Sidekiq::ProcessSet#size` that we saw above ultimately calls the Redis command `SCARD`.

Once you know the Redis command, you can consult Redis' documentation to understand the performance implications of that command being called. For example, the documentation for [SCARD](#),^[18] documents the *complexity* of the command as $O(1)$. This is *Big O Notation* and tells you how expensive it is to make that call.^[19] $O(1)$ means that [SCARD](#) takes the same amount of time to complete, regardless of the contents of Redis (more or less). That means it has low complexity, performs well, and you should be fine calling it very frequently.

As a comparison, Sidekiq uses [LRange](#) when iterating over queues.^[20] The complexity of this command is $O(S + N)$. Both S and N are documented, though in Big O Notation, N is usually a placeholder for “the size of the data being operated on.” In this case, the cost of calling [LRange](#) will increase linearly with the number of queues you have. It still may be very fast, but it's an order of magnitude slower than [SCARD](#), all things being equal.

Of course, these are just estimates, but they still give you some insights into what will happen if you call particular Sidekiq API methods frequently. After tracing the API calls to Redis commands a few times, you'll find you can quickly look up any other call and clearly understand what you are doing before you do it. This will prevent you from inadvertently tanking your app's performance by trying to monitor it too aggressively.

This all might seem really low level, but you do need to know how Redis works to understand the performance of your Sidekiq installation.

Understand How Redis Works

Redis is an in-memory data store. If Redis were to restart, or the machine it's running on restarts or goes away (as may happen in a cloud computing environment), all the data is lost. Since that data represents your unexecuted jobs, that means they would be lost. If that happens, to quote the inimitable Egon Spengler, “It would be bad.”

Fortunately, there are many ways to configure Redis in production to vastly reduce the chance of this happening. While you may not need a full-time Redis expert to *operate* Redis, you should avail yourself of one to set it up, especially if you are doing it yourself or using something like Amazon Web Service's (AWS) ElastiCache, since there aren't any presets tailored to Sidekiq.

In a general sense, you want Redis to have some sort of failover so that if the Redis you are using is lost, the system will automatically transition your app to an up-to-date backup. Many Redis vendors provide this, and it can be set up in a self- or cloud-hosted environment. Even still, there is a chance a job could be lost.

While you can't easily monitor for lost jobs directly, you really don't care about them unless that leads to a negative business outcome. This is why monitoring business functions is so critical. In the example app, you would want to monitor all orders created but not charged, or whose email notification hasn't gone out, or where fulfillment has not been requested. If that happens, you can correlate all your measures to figure out why, including a Redis failover happening during the time the order was created.

This is a deep topic, but what to take away here is that Redis is not an ACID-compliant database, [\[21\]](#) and the only way to be sure your business functions complete is to *monitor* them.

Alert on Actionable Problems

If you've followed the advice above, you will have a *lot* of measurements flowing into your monitoring system. Almost none of these can be used to identify problems on their own. For example, paging the entire engineering team when the Redis server's CPU usage goes above 90 percent isn't very useful, because it's not clear what actual problem is happening.

This includes the various statistics about Sidekiq or your specific jobs. You

need to be able to view the performance over time, but you should *alert* on business-level outcomes that are not normal, for example, the rate of order creation and completion. Observe those outcomes over time to establish a baseline, then set up alerting.

That being said, you should make sure to alert when Redis is running out of free space. Since memory is finite, when Redis' memory fills up, your entire system could fall over as you could lose the ability to both queue and process jobs.

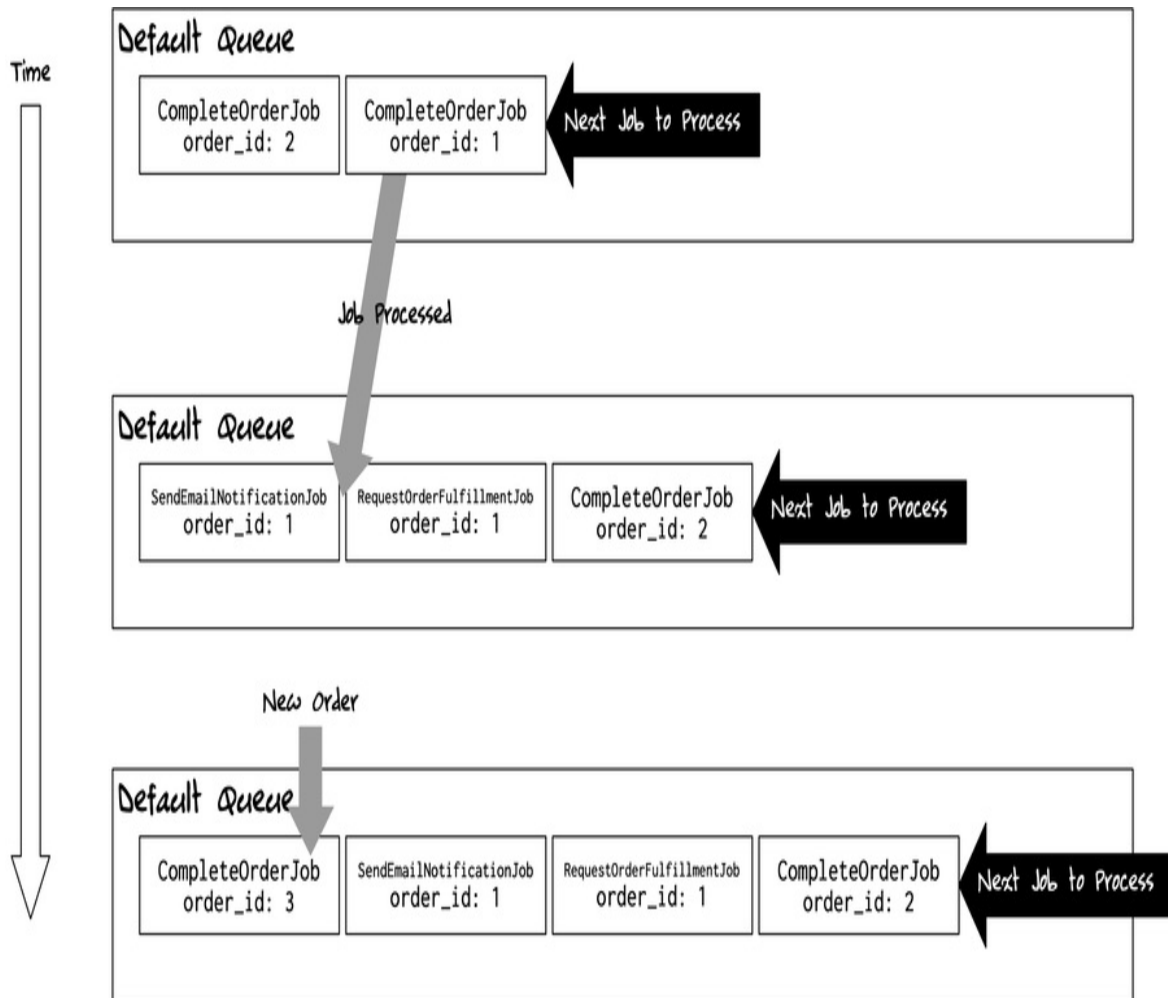
Observe the normal amount of space Redis uses and alert yourself if it goes above it for too long. Knowing in advance you may have trouble can make it a lot easier to get through a period of increased activity.

It's one thing to set up monitoring and alerting, but it's much more difficult to address performance problems you find. Let's review two options provided by Sidekiq: multiple queues and concurrency.

Using Queues and Concurrency to Control Performance

Suppose you are measuring the business function of order creation, specifically how long a user must wait to see confirmation that the system has their order. This timing is based on how long the [CompleteOrderJob](#) takes to complete. Imagine that Sidekiq is processing only one job at a time: each new [CompleteOrderJob](#) must wait for all existing jobs to finish before it is processed.

The following diagram shows how this might play out over a short period of time. You can see that when the first [CompleteOrderJob](#) completes, two new jobs are queued, and while the second [CompleteOrderJob](#) can be processed next, a third one that comes in must wait for the [SendOrderNotificationEmailJob](#) and [RequestOrderFulfillmentJob](#) jobs queued by the first [CompleteOrderJob](#).



A diagram showing a queue with jobs across three time periods. The first shows two [CompleteOrderJob](#) jobs queued, with order #1 ready for processing. The second time period shows the state of the queue after processing the first job. We see that behind the [CompleteOrderJob](#) for order #2 is a [SendOrderNotificationEmailJob](#) and a [RequestOrderFulfillmentJob](#). The third time period shows that the third order job was created while the [CompleteOrderJob](#) for order #2 is behind the two jobs created in the previous period.

There are two constraints on how long it takes for the user to get confirmation that the system has their order: the number of [CompleteOrderJob](#) jobs that can be processed at once, and the number of other jobs queued ahead of any new [CompleteOrderJob](#). In both cases, the user waits for something

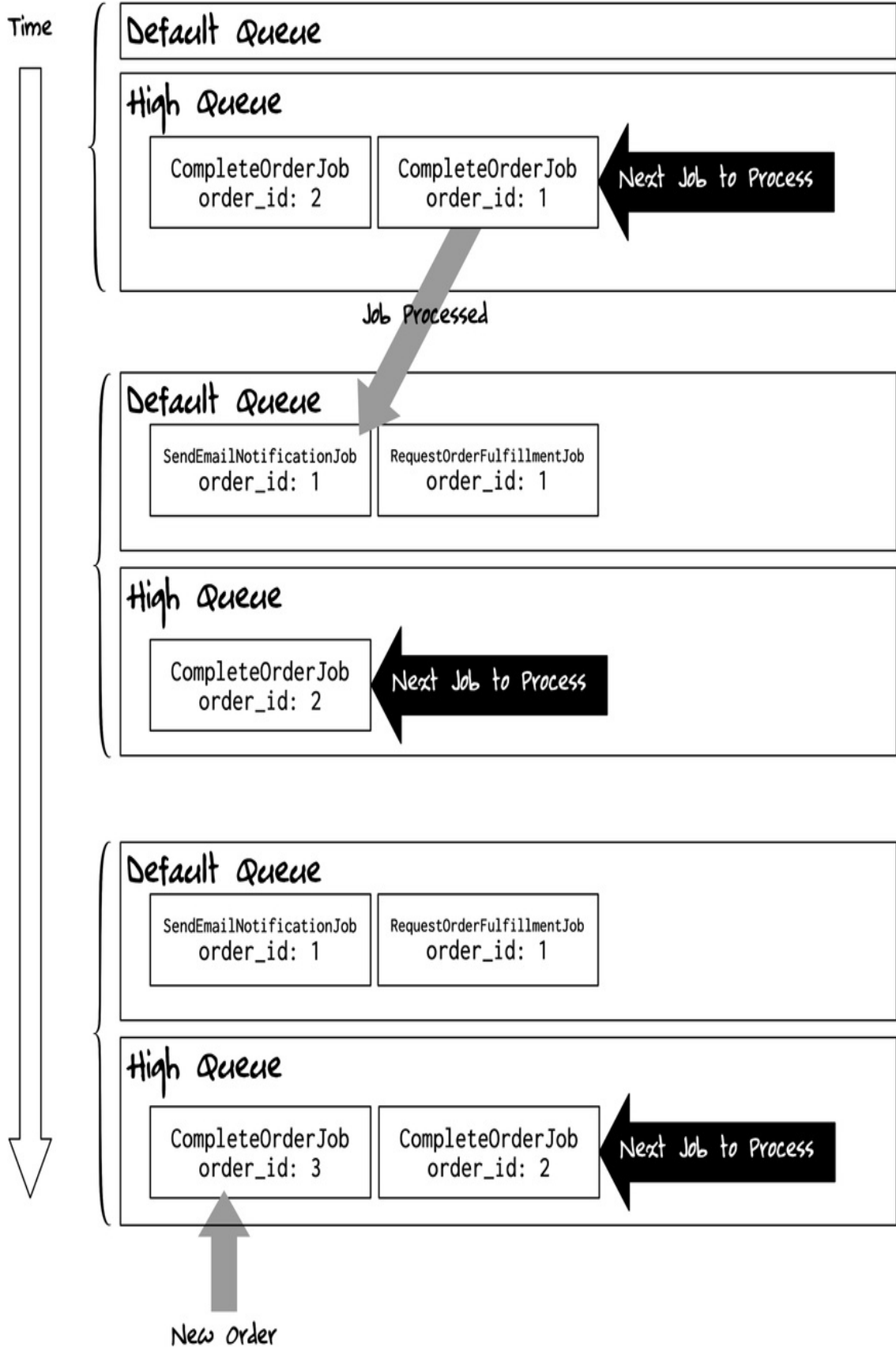
irrelevant to their order's creation.

Sidekiq provides two relatively easy-to-try mechanisms for addressing this: concurrency and priority queues.

Use Queues to Prioritize Jobs

Unlike [CompleteOrderJob](#), the results of the other two jobs aren't time-critical. There's no reason to require a user to wait in order to send notification emails or request fulfillment. If the [CompleteOrderJob](#) jobs could be given priority, that would help this issue.

Sidekiq supports this. You can tell Sidekiq to process all jobs on one queue before any job in another queue is processed. You can then queue [CompleteOrderJobs](#) to the higher priority queue, thus prioritizing those jobs. If you did that, the following diagram shows how the system would behave over time. Note that the third [CompleteOrderJob](#) no longer has to wait for all the jobs, and now just needs to wait for the second [CompleteOrderJob](#) before it can be processed.



A diagram showing queues over three time periods. In the first, the default queue is empty and the high queue contains two `CompleteOrderJob` jobs. The second time period shows that the `CompleteOrderJob` for order #1 has been processed and the `SendOrderNotificationEmailJob` and `RequestOrderFulfillmentJobs` have been placed on the default queue. The third time period shows a third `CompleteOrderJob` queued in the high queue, right behind the job for order #2.

You'll recall that the value `queues:` from `config/sidekiq.yml` is an array. The ordering is what Sidekiq uses to determine priority, so to create a queue with higher priority than the default queue, you can add the queue's name as the first element in this list, like so:

```
snapshots/4-2/sidekiq-book/config/sidekiq.yml
```

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

To use this queue, modify `CompleteOrderJob` to use `sidekiq_options` to set the queue to use:

```
snapshots/4-2/sidekiq-book/app/jobs/complete_order_job.rb
```

```
class CompleteOrderJob
  include Sidekiq::Job
»
» sidekiq_options queue: "high"

  def perform(order_id)
    order = Order.find(order_id)
    OrderCreator.new.complete_order(order)
  rescue BaseServiceWrapper::HTTPError => ex
    raise IgnorableExceptionSinceSidekiqWillRetry.new(ex)
  end
end
```

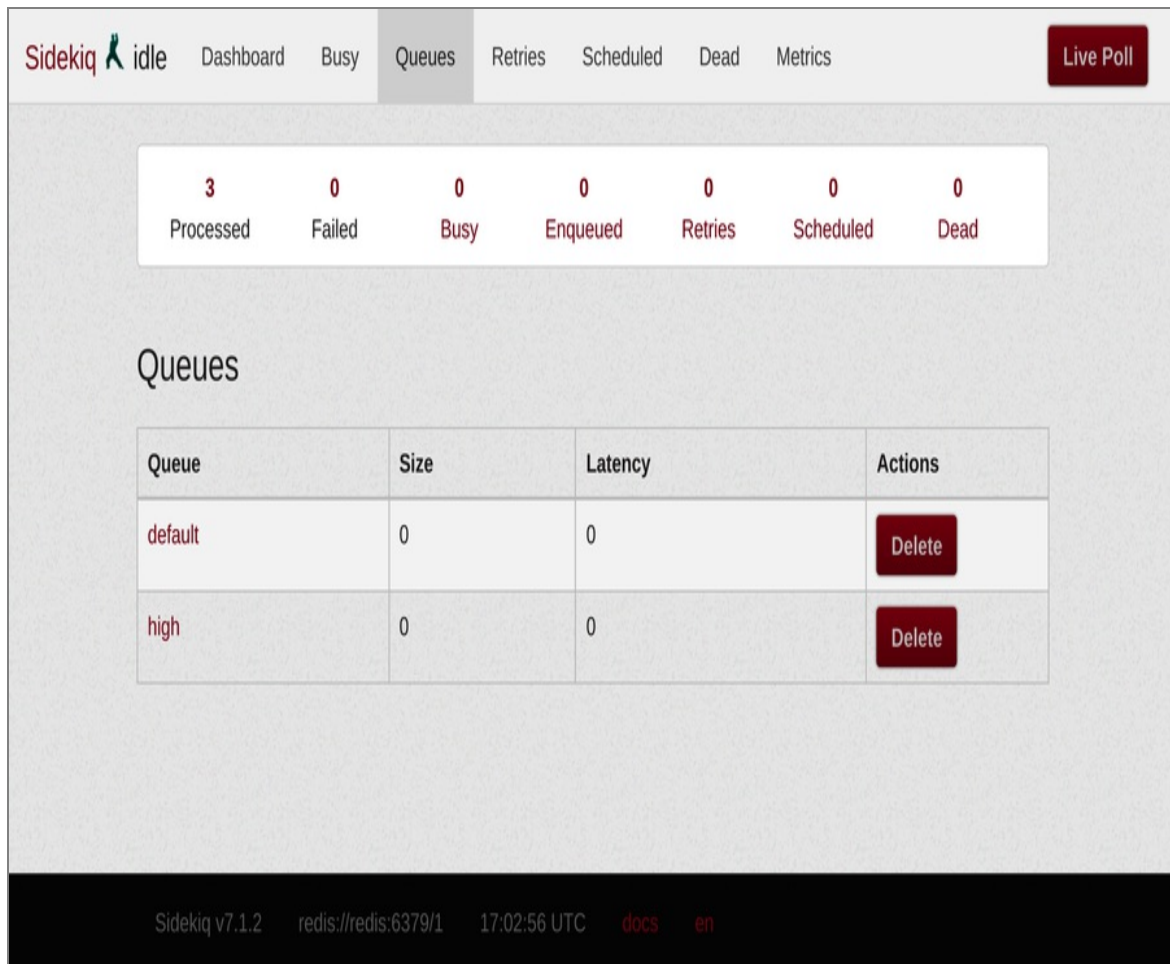
end

With this in place, restart your server, create an order, and visit the Sidekiq Web UI. Click on “Queues” and you should see the high queue (as in the screenshot that follows).

Fake Services Can Be Slowed



If you want to see that the job is processed in the high queue, follow the procedure used in the last two chapters to modify the behavior of the payment service, but instead of configuring it to crash, configure it to process slowly. You can then refresh the Sidekiq Web UI to see the job being processed in the new high priority queue.



A screenshot of the Sidekiq Web UI's queues view that shows both the default queue and the high queue.

Adding queues gives you finer control over the priority of jobs, but what about processing more jobs at once?

Use Concurrency to Process More Jobs at Once

In the example above, two orders were created at roughly the same time, but because we were only processing one job at a time, the second order had to wait. If you recall from [Set Sidekiq's Configuration Options](#), we configured a value for `concurrency`: with a default of 5. This value controls the number of threads each Sidekiq process will use.

Threads are bits of code running concurrently, but inside the same Ruby

process. Threads share memory, processes do not. The way we run the example app locally runs one Sidekiq process that spawns five threads. This means that our default configuration can process five jobs at once (all of which share the same memory—this will be relevant in a moment).

Suppose `CompleteOrderJob` takes two seconds to complete. If we got six orders at once, five of the users would wait two seconds, and one would have to wait four (two for at least one other job to finish and two for their job). If we got eleven orders at once, the eleventh order would have to wait six seconds.

In theory, you can adjust the number of processes and/or the number of threads per process to meet any demand. In practice, you have to do this carefully since each thread and process consumes system resources like CPU or memory. Tuning these two aspects of concurrency can be difficult and time-consuming. It highly depends on what your app does and how it is hosted. But you can make that job a bit easier by being mindful of how concurrency has the potential to create serious problems, even at small scale.

An Example Concurrency Problem

Consider the following code, which stores a value in a class variable.

```
class Fulfiller
  def start_fulfillment(order)
    self.num_fulfilled = self.num_fulfilled + 1
  end

  def self.num_fulfilled=(new_value)
    @num_fulfilled = new_value
  end

  def self.num_fulfilled
    @num_fulfilled || 0
  end
end
```

Because threads share memory, if you have five Sidekiq jobs each running in

a thread and all five call `Fulfiller.new.start_fulfillment(order)`, all five will attempt to modify the class variable `@num_fulfilled`. Depending on how the underlying virtual machine instructions are ordered, the value of `@num_fulfilled` will be unclear and possibly corrupted.

Concurrency issues can also happen when external resources, like your database, are shared. If you recall `AddMoreProductsJob` from [Understanding Idempotence](#), it updates a row in the database. If two of that job were queued for the same product but with different quantities, it's impossible to predict what quantity would eventually be stored in the database.

Avoiding Problems with Concurrent Execution

Like idempotency, it's difficult to analyze code to determine that it's safe to be executed concurrently with other code. It requires you to play out the same code happening at the same time in your head and decide if there is a case where the ordering of operations creates a problem. And, since any two blocks of code could potentially be executed concurrently, it's almost impossible to itemize every possible behavior.

Here are a few tips to help avoid the most common issues:

1. Avoid the use of `matr_accessor` or `catt_accessor`. These create a shared global state, which you almost never need. They are worse than that, however, because they allow that state to be manipulated by instance variables. This allows you to write concurrency-unsafe code that appears to be safe. There is no need to use these methods.
2. Avoid global variables (variables that begin with a dollar sign). If you need some data globally available, assign it to a constant and freeze its value so that you'll at least get a warning if you try to modify it (multiple threads reading the same data is generally safe).
3. Avoid methods that modify class variables, like the `Fulfiller` above. Any

code that stores data in a class variable and provides public methods to change that data is potentially thread unsafe.

4. If you must have shared global data that threads must modify, use the Concurrent Ruby gem, [\[22\]](#) which allows you to wrap access to shared data in thread-safe methods.
5. If you have multiple jobs manipulating the same underlying data in your database, consider serializing those jobs so they run one at a time (we'll see how to do that later in this chapter), or using the locking features of your database to create a more predictable environment.

Preventing concurrency issues that extend beyond your app's codebase can be extremely difficult. Just know that having a Rails app with Sidekiq jobs means that you have a distributed system. The monitoring and alerting discussed in the previous section is the key to managing it. Any concurrency issue *should* show up as a problem with your business function measurements, and the other measures can help you triangulate what may be the cause.

Adjust Concurrency Carefully

Once you've avoided common concurrency issues in your code, you may want to adjust the number of threads or the number of processes, or both. You can do this three ways, depending on your needs.

The first is what was alluded to in [Set Sidekiq's Configuration Options](#), which is to modify the environment variable `SIDEKIQ_CONCURRENCY` in your production environment. That controls the number of threads per process. Make only small changes and observe their effects. More threads consume more memory, but each thread will also consume a database connection, and those are typically limited. Whatever you change this to, you need to configure the size of the Active Record connection pool to match the number of threads. [\[23\]](#)

The second option is to add more server processes. These will consume more CPU and memory than adding threads, but it is often easier to modify the number of processes than it is to modify the number of threads. How you do this depends on how you run your app in production. The Sidekiq server process is run via `bundle exec sidekiq`. Often, running more server processes means allocating more virtual servers to run that process, but it depends on how you have it all set up. Some platforms allow you to scale up and down by adding or removing processes.

The last option is to combine all the techniques we've learned in this section, which is to run multiple Sidekiq servers with individualized configurations. For example, you could run `bundle exec sidekiq`, which will behave as discussed: one process with five threads, processing the high queue first, then the default queue. You could additionally run a command like `bundle exec sidekiq -q one_at_a_time -c 1`, which would run a new process that only processes the queue "one_at_a_time" with only one thread, effectively processing jobs in that queue in a serialized way. `bundle exec sidekiq -h` will show you additional options. The command line options override whatever is in `config/sidekiq.yml`.

The proper way to leverage these options highly depends on your app and workload. Observe how your app behaves. Measure it and make changes only to address a specific problem. If your changes don't fix it, roll them back and try something else.

One last thing to talk about is how to manage the code in the jobs themselves. We've already seen some duplication across jobs, and we still have `ApplicationJob` hanging around, configured to use Active Job, which we haven't discussed.

Organizing Sidekiq Job Code

Other than keeping the code in your jobs to a minimum (as discussed in [Create a New Sidekiq Job](#)), the main issue with organizing Sidekiq job code is managing duplication. There are three ways to do so: using middleware, using generators, and using a common base class. Using a base class should be familiar to you, since that is how most code is shared in Rails codebase. One problem with our Sidekiq jobs is that `ApplicationJob` is using Active Job.

No Need for Active Job

Active Job is a generalized interface for queuing jobs,^[24] that can use any job library underneath, be it Sidekiq, Resque, or something else. This is useful when an external library or Rails' internals need to queue a job. This is less useful when writing a Rails application.

Active Job doesn't alleviate the need to understand Sidekiq in detail—you still need to know how Sidekiq manages parameters, Redis, and errors. If you use Active Job, you have more stuff to learn, without any real benefit to doing so.

Active Job also modifies the parameters queued with a job, namely if you pass in an Active Record. It will handle putting the ID into the job payload, and then using `find` to look it up before the job executes. But it only works with Active Records, and it only saves a single line of very obvious, very hard-to-mess-up code. It's not worth it.

To codify the decision *not* to use Active Job, let's change `ApplicationJob` to be a base class for Sidekiq jobs. A comment documents what you've done and you can include the `Sidekiq::Job` module, so that jobs don't have to include that themselves.


```
# All jobs use Sidekiq, not Active Job
class ApplicationJob
  include Sidekiq::Worker
end
```

With this in place, you can change all existing jobs to extend `ApplicationJob` and remove `include Sidekiq::Job` from them while you are at it.

```
snapshots/4-3/sidekiq-book/app/jobs/complete_order_job.rb
```

```
# XXX class CompleteOrderJob
# XXX include Sidekiq::Job
» class CompleteOrderJob < ApplicationJob

  sidekiq_options queue: "high"

  def perform(order_id)
    order = Order.find(order_id)
    OrderCreator.new.complete_order(order)
  rescue BaseServiceWrapper::HTTPError => ex
    raise IgnorableExceptionSinceSidekiqWillRetry.new(ex)
  end
end
```

```
snapshots/4-3/sidekiq-book/app/jobs/send_order_notification_email_job.rb
```

```
# XXX class SendOrderNotificationEmailJob
# XXX include Sidekiq::Job
» class SendOrderNotificationEmailJob < ApplicationJob
  def perform(order_id)
    order = Order.find(order_id)
    OrderCreator.new.send_notification_email(order)
  rescue BaseServiceWrapper::HTTPError => ex
    raise IgnorableExceptionSinceSidekiqWillRetry.new(ex)
  end
end
```

```
snapshots/4-3/sidekiq-book/app/jobs/request_order_fulfillment_job.rb
```

```
# XXX class RequestOrderFulfillmentJob
# XXX include Sidekiq::Job
» class RequestOrderFulfillmentJob < ApplicationJob
```

```

def perform(order_id)
  order = Order.find(order_id)
  OrderCreator.new.request_order_fulfillment(order)
rescue BaseServiceWrapper::HTTPError => ex
  raise IgnorableExceptionSinceSidekiqWillRetry.new(ex)
end
end

```

But what if someone uses `bin/rails g job` to make a new job?

Changing the Rails Job Generator to Use Sidekiq

Not every developer uses the Rails generator commands, but if you or anyone on your team does, you should make sure it generates a new job that uses Sidekiq and not Active Job. By default, Rails uses an internal template to create a job.^[25] If your Rails app has the file `lib/templates/rails/job/job.rb.tt` in it, *that* template will be used. Despite the odd filename extension, this is an ERB file.

Since `--queue` allows specifying the queue when generating the job, your job should use that value with `sidekiq_options`. It's also good practice to have generated code raise an error. This way, if you forget to implement the job, it will fail loudly in testing or production. Note that `module_namespacing` handles a situation where you put your job in a sub-module.

snapshots/4-2/sidekiq-book/lib/templates/rails/job/job.rb.tt

```

<% module_namespacing do -%>
class <%= class_name %>Job < ApplicationJob
  <% if options[:queue].to_s != "default" -%>
    sidekiq_options queue: "<%= options[:queue] %>"

  <% end -%>

  def perform(args,should,go,here)
    raise "This job has not been implemented"
  end
end
end
<% end -%>

```

You should also override the test case that is generated. Even though you

aren't likely to want to unit test your jobs, the default unit test includes Active Job helper methods which won't work. The template to create to do this is [lib/templates/test_unit/job/unit_test.rb.tt](#)

```
snapshots/4-2/sidekiq-book/lib/templates/test_unit/job/unit_test.rb.tt
```

```
require "test_helper"

<% module_namespacing do -%>
class <%= class_name %>JobTest < ActiveSupport::TestCase
  test "if a test is needed" do
    assert false, "Delete this file if you don't need a unit test"
  end
end
<% end -%>
```

Try it out via `bin/rails g job special_jobs/my_job --queue=high`, then look at the files that were generated:

```
snapshots/4-2/sidekiq-book/app/jobs/special_jobs/my_job.rb
```

```
class SpecialJobs::MyJob < ApplicationJob
  sidekiq_options queue: "high"

  def perform(args,should,go,here)
    raise "This job has not been implemented"
  end
end
```

```
snapshots/4-2/sidekiq-book/test/jobs/special_jobs/my_job_test.rb
```

```
require "test_helper"

class SpecialJobs::MyJobTest < ActiveSupport::TestCase
  test "if a test is needed" do
    assert false, "Delete this file if you don't need a unit test"
  end
end
```

We didn't include the use of `rescue` to manage transient errors. That sort of duplication is better managed by middleware.

Use Middleware to Eliminate Duplication

`ApplicationJob` is a handy place to put job-specific helper methods you may discover you need. You could add one to extract the code we added for transient errors in [Prevent Transient Errors from Notifying You](#). A better way is to use a server middleware.

Middleware in Sidekiq is like Rack Middleware^[26]—it allows you to inject code into Sidekiq’s internal process. Sidekiq has *server middleware*, which allows you to run code around when a job is executed, and *client middleware*, which allows running code around when a job is queued.

By using a server middleware, you can catch every exception thrown by a job and examine it. If the exception is known to be transient, you can wrap it in a `IgnoreableExceptionSinceSidekiqWillRetry`. Otherwise, you can let it bubble up as normal.

Let’s build and test this middleware. The middleware itself can maintain a list of common transient exceptions (like `BaseServiceWrapper::HTTPError`), but it will also allow job classes to specify their own list of transient exceptions via a class method named `transient_exceptions`, like so:

```
class RefundJob < ApplicationJob
  » def self.transient_exceptions
  »   [ Stripe::RateLimitError ]
  » end

  def perform(order_id)
    order = Order.find(order_id)
    Refunder.new.refund(order)
  end
end
```

The middleware should go in `lib/sidekiq_middleware/server/silence_transient_errors.rb`. It will call `yield`, then examine the exception raised. If the exception’s class is in the middleware’s list or the list provided by the job class, it’ll wrap it in

`IgnorableExceptionSinceSidekiqWillRetry`. Otherwise, the exception will be reraised (which will presumably trigger your error catcher).

```
snapshots/4-4/sidekiq-book/lib/sidekiq_middleware/server/silence_transient_errors.rb
```

```
module SidekiqMiddleware
  module Server
    class SilenceTransientErrors
      include Sidekiq::ServerMiddleware
      def call(job_instance, _job_payload, _queue)
        begin
          yield
        rescue => ex
          if transient?(job_instance, ex)
            raise IgnorableExceptionSinceSidekiqWillRetry.new(ex)
          else
            raise
          end
        end
      end
    end

    private

    RETRIABLE_EXCEPTIONS = [
      "BaseServiceWrapper::HTTPError",
    ]

    def transient?(job_instance, ex)
      if ex.class.to_s.in?(RETRIABLE_EXCEPTIONS)
        return true
      end
      job_instance.class.respond_to?(:transient_exceptions) &&
        ex.class.to_s.in?(job_instance.class.transient_exceptions.map(&:to_s))
    end
  end
end
```

Note that the code coerces all classes to strings to avoid any issues with load order. You can configure this middleware in [config/initializers/sidekiq.rb](#):

```
snapshots/4-4/sidekiq-book/config/initializers/sidekiq.rb
```

```
» require "sidekiq_middleware/server/silence_transient_errors"
»
» Sidekiq.configure_server do |config|
»   config.server_middleware do |chain|
»     chain.add SidekiqMiddleware::Server::SilenceTransientErrors
»   end
»   config.death_handlers << ->(job,exception) {
»     ErrorCatcherServiceWrapper.new.notify(
»       "#{job['class']} won't be retried: #{exception.message}"
»     )
»   }
» end
```

Since this middleware is somewhat complex, let's write a test for it.

Test Complicated Middleware

To test the middleware, we'll queue jobs as normal, and execute them via `drain_all`, as we did in the system test. That will ensure that the middleware works *and* is being properly used by Sidekiq. Note that Sidekiq does not configure middleware in a test, so the test will need to do that.

We'll want to have three total tests: one that checks that a normal exception bubbles up when the job runs, another that ensures `BaseServiceWrapper::HTTPError` is correctly translated to a `IgnorableExceptionSinceSidekiqWillRetry`, and a third that uses a job class with a custom transient error.

The `setup` block uses code similar to what's in `app/controllers/simulated_behaviors_controller.rb` to provide access to the fake payments service behavior simulation. The second test uses this to cause the fake payment service to crash, thus causing `CompleteOrderJob` to raise `BaseServiceWrapper::HTTPError`.

```
snapshots/4-4/sidekiq-book/test/lib/sidekiq_middleware/server/silence_transient_errors_test.rb
```

```
require "test_helper"
require "sidekiq_middleware/server/silence_transient_errors"

class SidekiqMiddleware::Server::SilenceTransientErrorsTest < ActiveSupport::TestCase
```

```

setup do
  @payments_service_status = ServiceStatus.find("payments")
  @payments_service_status.update(sleep: 0, throttle: false, crash: false)
  Sidekiq::Testing.server_middleware do |chain|
    chain.add SidekiqMiddleware::Server::SilenceTransientErrors
  end
end

test "normal job raising a non-transient exception" do
  job_instance = CompleteOrderJob.new
  non_existent_order_id = -99
  CompleteOrderJob.perform_async(non_existent_order_id)
  assert_raises ActiveRecord::RecordNotFound do
    Sidekiq::Job.drain_all
  end
end

test "normal job raising a transient exception" do
  @payments_service_status.update(sleep: 0, throttle: false, crash: true)
  order = FactoryBot.create(:order)
  CompleteOrderJob.perform_async(order.id)
  assert_raises IgnorableExceptionSinceSidekiqWillRetry do
    Sidekiq::Job.drain_all
  end
end

class CustomTransientJob < ApplicationJob
  def self.transient_exceptions
    [
      ArgumentError
    ]
  end

  def perform(throw_argument_error)
    if throw_argument_error
      raise ArgumentError
    end
  end
end

test "job raising a custom transient exception" do
  order = FactoryBot.create(:order)
  CustomTransientJob.perform_async(true)
  assert_raises IgnorableExceptionSinceSidekiqWillRetry do
    Sidekiq::Job.drain_all
  end
end

```

end

Because the test uses `CompleteOrderJob`, you can remove the explicit exception handling from it:

```
snapshots/4-4/sidekiq-book/app/jobs/complete_order_job.rb
```

```
class CompleteOrderJob < ApplicationJob

  sidekiq_options queue: "high"

  def perform(order_id)
    order = Order.find(order_id)
    OrderCreator.new.complete_order(order)
    # XXX rescue BaseServiceWrapper::HTTPError => ex
    # XXX raise IgnorableExceptionSinceSidekiqWillRetry.new(ex)
    » # custom exception handling removed
  end
end
```

The test should pass:

```
> bin/rails test \  
  test/lib/sidekiq_middleware/server/silence_transient_errors_test.rb
```

```
Run options: --seed 21614
```

```
# Running:
```

```
2023-04-23T16:27:46.371Z pid=4966 tid=aue INFO: Sidekiq 7.0...
```

```
...
```

```
Finished in 0.187307s, 16.0164 runs/s, 16.0164 assertions/s.
```

```
3 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

You can remove the exception translation from the other two jobs now if you like.

Additional Topics to Explore Next

We can't cover every single aspect of Sidekiq or background jobs in this book. While we've gone through common and critical aspects of running Sidekiq, there are a few other areas you may wish to explore or know about.

Request-Scoped Data Is Available to Your Jobs

The Rails `CurrentAttributes` class allows you to store request-scoped data in a common location available to all classes.^[27] For example, you could put the currently logged-in user's ID into `CurrentAttributes`. Since `CurrentAttributes` uses memory, it may seem that when you queue a job, that memory won't be available when the job is executed.

It's true that the memory used by `CurrentAttributes` won't be available, but Sidekiq can preserve this for you. Suppose you have created the class `Current` in `app/models/current.rb`, as documented by Rails' API. You can arrange to have Sidekiq store everything in the job payload, and restore it before the job executes. Put code like this in `config/initializers/sidekiq.rb`:

```
require "sidekiq/middleware/current_attributes"  
Sidekiq::CurrentAttributes.persist(Current)
```

Queue Many Jobs at Once with `perform_bulk`

You may find yourself needing to queue many jobs at once. For example, suppose instead of queuing `RequestOrderFulfillmentJob` right after sending email, you wanted to request fulfillment for all orders placed that day at midnight, all in one group. While you could iterate over each unfulfilled order via `Order.where(fulfillment_request_id: nil).each`, this would make many Redis calls and could perform poorly as your database gets larger.

Instead, you can use `perform_bulk`. It expects an array of job arguments (essentially, an array of arrays), and a `batch_size` to allow Sidekiq to batch the

Redis calls (which Sidekiq advises you limit to 1,000):

```
order_ids = Order.where(fulfillment_request_id: nil).pluck(:id)
array_of_args = order_ids.zip # Turns each element into an array
RequestOrderFulfillmentJob.perform_bulk(order_ids, batch_size: 500)
```

You could put this code into a nightly job or scheduled task.

Scheduling Jobs Is Available with the Enterprise Version

To schedule jobs to run at a given time—such as a job containing the code above—you can use a third-party gem like Sidekiq Scheduler,^[28] or you can use the Periodic Jobs feature of Sidekiq Enterprise,^[29] which, as the name implies, requires a paid Sidekiq Enterprise plan. Sometimes the best way to solve a problem is to pay someone else to solve it.

The advantage of Periodic Jobs is that it's part of Sidekiq and is supported directly. It's reliable and will always work. The downside is that it requires an Enterprise plan which you may not be able to afford. While Sidekiq Scheduler is free and does work pretty well, it tends to lag behind Sidekiq in terms of features and support. This could prevent you from upgrading Sidekiq in the future.

Sidekiq Enterprise and Pro Plans Solve Real Problems

While I'm neither affiliated with nor compensated by Sidekiq's author, it is worth knowing that the paid plans (Pro and Enterprise) for Sidekiq aren't just there to support the author's work. They fund real features that solve real problems. Periodic Jobs is one of them, but another is reliability when fetching jobs to process.

We discussed earlier that it's possible for jobs to be lost due to the way Redis works. It's also possible to lose a job if Sidekiq has fetched a job to process, but then dies before completing the job *and then* is unable to push the job back to Redis to try again later. While this may be unlikely, it's possible. Sidekiq's Pro plan provides much stronger guarantees that this won't happen.

Depending on what your app does, a reliability guarantee could be worth far more than the cost of a paid plan. The Pro and Enterprise plans also come with support, which could be invaluable. Very few open-source projects have a way to pay the author for support. Rails, for example, does not!

Wrapping Up

This concludes our whirlwind tour of Sidekiq! You should now have a solid foundation to write and manage background jobs, and your Rails app should still be relatively predictable and sustainable. As mentioned above, your Rails app running Sidekiq is a distributed system, so you will benefit greatly from reading more about managing distributed systems in production. The Wikipedia article for the Fallacies of Distributed Computing is a great place to get familiar with the problems and terminology.^[30] The best teacher, however, is experience building, operating, and monitoring real-world software.

Footnotes

[10] <https://www.honeycomb.io>

[11] <https://newrelic.com>

[12] https://github.com/thoughtbot/factory_bot

[13] <https://github.com/sidekiq/sidekiq/wiki/API>

[14] <https://github.com/sidekiq/sidekiq/wiki/Monitoring>

[15] <https://sidekiq.org/products/pro.html>

[16] <https://github.com/statsd/statsd>

[17] <https://github.com/sidekiq/sidekiq/blob/main/lib/sidekiq/api.rb>

[18] <https://redis.io/commands/scard/>

[19] https://en.wikipedia.org/wiki/Big_O_notation

[20] <https://redis.io/commands/lrange/>

[21] <https://en.wikipedia.org/wiki/ACID>

[22] <https://github.com/ruby-concurrency/concurrent-ruby>

[23] <https://api.rubyonrails.org/classes/ActiveRecord/ConnectionAdapters/ConnectionPool.html>

- [24] https://guides.rubyonrails.org/active_job_basics.html
- [25] <https://github.com/rails/rails/blob/main/activejob/lib/rails/generators/job/templates/job.rb.tt>
- [26] https://guides.rubyonrails.org/rails_on_rack.html#action-dispatcher-middleware-stack
- [27] <https://api.rubyonrails.org/classes/ActiveSupport/CurrentAttributes.html>
- [28] <https://github.com/sidekiq-scheduler/sidekiq-scheduler>
- [29] <https://github.com/sidekiq/sidekiq/wiki/Ent-Periodic-Jobs>
- [30] https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

Copyright © 2023, The Pragmatic Bookshelf.

Thank you!

We hope you enjoyed this book and that you're already thinking about what y that decision easier, we're offering you this gift.

Head on over to <https://pragprog.com> right now, and use the coupon code BU your next ebook. Offer is void where prohibited or restricted. This offer does 1 *Pragmatic Programmer* ebook.

And if you'd like to share your own expertise with the world, why not propos of our best authors started off as our readers, just like you. With up to a 50% r and a name you trust, there's nothing to lose. Visit <https://pragprog.com/becor> to get started.

We thank you for your continued support, and we hope to hear from you again

The Pragmatic Bookshelf



SAVE 30%!
Use coupon code
BUYANOTHER2023