Sustainable Web Development

WITH RUBY ON RAILS

by **David Bryant Copeland**

Practical Tips for Building Web Applications that Last

Sustainable Web Development with Ruby on Rails

Practical Tips for Building Web Applications that Last

David Bryant Copeland

Contents

Acknowledgements

If there were no such thing as Rails, this book would be, well, pretty strange. So I must acknowledge and deeply thank DHH and the Rails core team for building and maintaining such a wonderful framework for all of us to use.

I have to thank my wife, Amy, who gave me the space and encouragement to work on this. During a global pandemic. When both of us were briefly out of work. And we realized our aging parents require more care than we thought. And when we got two kittens named Carlos Rosario and Zoni. And when we bought a freaking car. And when I joined a pre-seed startup. It's been quite a time.

I also want to thank the technical reviewers, Noel Rappin, Chris Gibson, Zach Campbell, Lisa Sheridan, Raul Murciano, Geoff The, and Sean Miller.

PART

I

introduction

1

Why This Book Exists

Rails can scale. But what does that actually mean? And how do we do it? This book is the answer to both of these questions, but instead of using "scalable", which many developers equate with "fast performance", I'm using the word "sustainable". This is really what we want out of our software: the ability to sustain that software over time.

Rails itself is an important component in sustainable web development, since it provides common solutions to common problems and has reached a significant level of maturity. But it's not the complete picture.

Rails has a lot of features and we may not need them all. Or, we may need to take some care in how we use them. Rails also leaves gaps in your application's architecture that you'll have to fill (which makes sense, since Rails can't possibly provide *everything* your app will need).

This book will help you navigate all of that.

Before we begin, I want to be clear about what *sustainability* means and why it's important. I also want to state the assumptions I'm making in writing this, because there is no such thing as universal advice—there's only recommendations that apply in a given context.

1.1 What is Sustainability?

The literal interpretation of sustainable web development is web development that can be sustained. As silly as that definition is, I find it an illuminating restatement.

To *sustain* the development of our software is to ensure that it can continue to meet its needs. A sustainable web app can easily suffer new requirements, increased demand for its resources, and an increasing (or changing) team of developers to maintain it.

A system that is hard to change is hard to sustain. A system that can't avail itself of the resources it needs to function is hard to sustain. A system that only *some* developers can work on is hard to sustain.

Thus, a sustainable application is one in which changes we make tomorrow are as easy as changes are today, for whatever the application might need to do and whoever might be tasked with working on it.

So this defines *sustainability*, but why is it important?

1.2 Why Care About Sustainability?

Most software exists to meet some need, and if that need will persist over time, so must the software. *Needs* are subjective and vague, while software must be objective and specific. Thus, building software is often a matter of continued refinement as the needs are slowly clarified. And, of course, needs have a habit of changing along the way.

Software is expensive, mostly owing to the expertise required to build and maintain it. People who can write software find their skills to be in high demand, garnering some of the highest wages in the world, even at entry levels. It stands to reason that if a piece of software requires more effort to enhance and maintain over time, it will cost more and more and deliver less and less.

In an economic sense, sustainable software minimizes the cost of the software over time. But there is a human cost to working on software. Working on sustainable software is, well, more enjoyable. They say employees quit managers, but I've known developers that quit codebases. Working on unsustainable software just plain sucks, and I think there's value in having a job that doesn't suck. . . at least not all of the time.

Of course, it's one thing to care about sustainability in the abstract, but how does that translate into action?

1.3 How to Value Sustainability

Sustainability is like an investment. It necessarily won't pay off in the short term and, if the investment isn't sound, it won't ever pay off. So it's really important to understand the value of sustainability to your given situation and to have access to as much information as possible to know exactly how to invest in it.

Predicting the future is dangerous for programmers. It can lead to over-engineering, which makes certain classes of changes more difficult in the future. To combat this urge, developers often look to the tenets of agile software development, which have many cute aphorisms that boil down to "don't build software that you don't know you need".

If you are a hired consultant, this is excellent advice. It gives you a framework to be successful and manage change when you are in a situation where you have very little access to information. The strategy of "build for only what you 100% know you need" works great to get software shipped with confidence, but it doesn't necessarily lead to a sustainable outcome.

For example, no business person is going to ask you to write log statements so you can understand your code in production. No product owner is going to ask you to create a design system to facilitate building user interfaces more quickly. And no one is going to require that your database has referential integrity.

The features of the software are merely one input into what software gets built. They are a significant one, to be sure, but not the only one. To make better technical decisions, you need access to more information than simply what someone wants the software to do.

Do you know what economic or behavioral output the software exists to produce? In other words, how does the software make money for the people paying you to write it? What improvements to the business is it expected to make? What is the medium or long-term plan for the business? Does it need to grow significantly? Will there need to be increased traffic? Will there be an influx of engineers? Will they be very senior, very junior, or a mix? When will they be hired and when will they start?

The more information you can get access to, the better, because all of this feeds into your technical decision-making and can tell you just how sustainable your app needs to be. If there will be an influx of less experienced developers, you might make different decisions than if the team is only hiring one or two experienced specialists.

Armed with this sort of information, you can make technical decisions as part of an overall *strategy*. For example, you may want to spend several days setting up a more sustainable development environment. By pointing to the company's growth projections and your teams hiring plans, that work can be easily justified (see the sidebar "Understanding Growth At Stitch Fix" on the next page for a specific example of this).

If you don't have the information about the business, the team, or anything other than what some user wants the software to do, you aren't set up to do sustainable development. But it doesn't mean you shouldn't ask anyway.

People who don't have experience writing software won't necessarily intuit that such information is relevant, so they might not be forthcoming. But you'd be surprised just how much information you can get from someone by asking.

Whatever the answers are, you can use this as part of an overall technical strategy, of which sustainability is a part. As you read this book, I'll talk about the considerations around the various recommendations and techniques. They might not all apply to your situation, but many of them will.

Which brings us to the set of assumptions that this book is based on. In other words, what *is* the situation in which sustainability is important and in which this book's recommendations apply?

Understanding Growth At Stitch Fix

During my first few months at Stitch Fix, I was asked to help improve the operations of our warehouse. There were many different processes and we had a good sense of which ones to start automating. At the time, there was only one application—called HELLBLAZER—and it served up `stitchfix.com`.

If I hadn't been told anything else, the simplest thing to do would've been to make a `/warehouse` route in HELLBLAZER and slowly add features for the associates there. But I *had* been told something else.

Like almost everyone at the company, the engineering team was told—very transparently—what the growth plans for the business were. It needed to grow in a certain way or the business would fail. It was easy to extrapolate from there what that would mean for the size of the engineering team, and for the significance of the warehouse's efficiency. It was clear that a single codebase everyone worked in would be a nightmare, and migrating away from it later would be difficult and expensive.

So, we created a new application that shared HELLBLAZER's database. It would've certainly been faster to add code to HELLBLAZER directly, but we knew doing so would burn us long-term. As the company grew, the developers working on warehouse software were fairly isolated since they worked in a totally different codebase. We replicated this pattern and, after six years of growth, it was clearly the right decision, even accounting for problems that happen when you share a database between apps.

We never could've known that without a full understanding of the company's growth plans, and long-term vision for the problems we were there to solve.

1.4 Assumptions

This book is pretty prescriptive, but each prescription comes with an explanation, and *all* of the book's recommendations are based on some key assumptions that I would like to state explicitly. If your situation differs wildly from the one described below, you might not get that much out of this book. My hope—and belief—is that the assumptions below are common, and that the situation of writing software that you find yourself in is similar to situations I have faced. Thus, this book will help you.

In case it's not, I want to state my assumptions up front, right here in this free chapter.

1.4.1 The Software Has a Clear Purpose

This might seem like nonsense, but there are times when we don't exactly know what the software is solving for, yet need to write some software to explore the problem space.

Perhaps some venture capitalist has given us some money, but we don't yet know the exact market for our solution. Maybe we're prototyping a potentially complex UI to do user testing. In these cases we need to be nimble and try to figure out what the software should do.

The assumption here is that that has already happened. We know generally what problem we are solving, and we aren't going to have to pivot from selling shoes to providing AI-powered podiatrist back-office enterprise software.

1.4.2 The Software Needs To Exist For Years

This book is about how to sustain development over a longer period of time than a few months, so a big assumption is that the software actually *needs* to exist that long!

A lot of software falls into this category. If you are automating a business process, building a customer experience, or integrating some back-end systems, it's likely that software will continue to be needed for quite a while.

1.4.3 The Software Will Evolve

Sometimes we write code that solves a problem and that problem doesn't change, so the software is stable. That's not an assumption I am making here. Instead, I'm assuming that the software will be subject to changes big and small over the years it will exist.

I believe this is more common than not. Software is notoriously hard to get right the first time, so it's common to change it iteratively over a long period to arrive at optimal functionality. Software that exists for years also tends to need to change to keep up with the world around it.

1.4.4 The Team Will Change

The average tenure of a software engineer at any given company is pretty low, so I'm assuming that the software will outlive the team, and that the group of people charged with the software's maintenance and enhancement will change over time. I'm also assuming the experience levels and skill-sets will change over time as well.

1.4.5 You Value Sustainability, Consistency, and Quality

Values are fundamental beliefs that drive actions. While the other assumptions might hold for you, if you don't actually value sustainability, consistency, and quality, this book isn't going to help you.

Sustainability

If you don’t value sustainability as I’ve defined it, you likely didn’t pick up this book or have stopped reading by now. You’re here because you think sustainability is important, thus you *value* it.

Consistency

Valuing consistency is hugely important as well. Consistency means that designs, systems, processes, components (etc.), should not be arbitrarily different. Same problems should have same solutions, and there should not be many ways to do something. It also means being explicit that personal preferences are not critical inputs to decision-making.

A team that values consistency is a sustainable team and will produce sustainable software. When code is consistent, it can be confidently abstracted into shared libraries. When processes are consistent, they can be confidently automated to make everyone more productive.

When architecture and design are consistent, knowledge can be transferred, and the team, the systems, and even the business itself can survive potentially radical change (see the sidebar “Our Uneventful Migration to AWS” on the next page for how Stitch Fix capitalized on consistency to migrate from Heroku to AWS with no downtime or outages).

Quality

Quality is a vague notion, but it’s important to both understand it and to value it. In a sense, valuing quality means doing things right the first time. But “doing things right” doesn’t mean over-engineering, gold-plating, or doing something fancy that’s not called for.

Valuing quality is to acknowledge the reality that we aren’t going to be able to go back and clean things up after they have been shipped. There is this fantasy developers engage in that they can simply “acquire technical debt” and someday “pay it down”.

I have never seen this happen, at least not in the way developers think it might. It is extremely difficult to make a business case to modify working software simply to make it “higher quality”. Usually, there must be some catastrophic failure to get the resources to clean up a previously-made mess. It’s simpler and easier to manage a process by which messes don’t get made as a matter of course.

Quality should be part of the everyday process. Doing this consistently will result in predictable output, which is what managers really want to see. On the occasion when a date must be hit, cut scope, not corners. Only the developers know what scope to cut in order to get meaningfully faster delivery, but this requires having as much information about the business strategy as possible.

When you value sustainability, consistency, and quality, you will be unlikely to find yourself in a situation where you must undo a technical decision you made at the cost of shipping more features. Business people may want software delivered as fast as possible, but they *really* don't want to go an extended period without any features so that the engineering team can "pay down" technical debt.

We know what sustainability is, how to value it, what assumptions I'm making going in, and that values that drive the tactics and strategy for the rest of the book. But there are two concepts I want to discuss that allow us to attempt to quantify just how sustainable our decisions are: opportunity costs and carrying costs.

Our Uneventful Migration to AWS

For several years, Stitch Fix used the platform-as-a-service Heroku. We were consistent in how we used it, as well as in how our applications were designed. We used one type of relational database, one type of cache, one type of CDN, etc.

In our run-up to going public, we needed to migrate to AWS, which is *very* different from Heroku. We had a team of initially two people and eventually three to do the migration for the 100+ person engineering team. We didn't want downtime, outages, or radical changes in the developer experience.

Because everything was so consistent, the migration team was able to quickly build a deployment pipeline and command-line tool to provide a Heroku-like experience to the developers. Over several months we migrated one app and one database at a time. Developers barely noticed, and our users and customers had no idea.

The project lead was so confident in the approach and the team that he kept his scheduled camping trip to an isolated mountain in Colorado, unreachable by the rest of the team as they moved `stitchfix.com` from Heroku to AWS to complete the migration. Consistency was a big part of making this a non-event.

1.5 Opportunity and Carrying Costs

An *opportunity cost* is basically a one-time cost to produce something. By committing to work, you necessarily cut off other avenues of opportunity. This cost can be a useful lens to compare two different approaches when trying to perform a cost/benefit analysis. An opportunity cost we'll take in a few chapters is writing robust scripts for setting up our app, running it, and running its tests. It has a higher opportunity cost than simply writing documentation about how to do those things.

But sometimes an investment is worth making. The way to know if that's true is to talk about the *carrying cost*. A carrying cost is a cost you have to pay all the time every time. If it's difficult to run your app in development, reading the documentation about how to do so and running all the various commands is a cost you pay frequently.

It is carrying costs that most greatly affect sustainability. Each line of code is a carrying cost. Each new feature has a carrying cost. Each thing we have to remember to do is a carrying cost. This is the true value provided by Rails: it reduces the carrying costs of a lot of pretty common patterns when building a web app.

To sustainably write software requires carefully balancing your carrying costs, and strategically incurring opportunity costs that can reduce, or at least maintain, your carrying costs.

If there are two concepts most useful to engineers, it is these two.

The last bit of information I want to share is about me. This book amounts to my advice based on my experience, and you need to know about that, because, let's face it, the field of computer programming is pretty far away from science, and most of the advice we get is nicely-formatted survivorship bias.

1.6 Why should you trust me?

Software engineering is notoriously hard to study and most of what exists about how to write software is anecdotal evidence or experience reports. This book is no different, but I do believe that if you are facing problems similar to those I have faced, there is value in here.

So I want to outline what my experience is that has led to me recommend what I do in this book.

The most important thing to know about me is that I'm not a software consultant, nor have I been in a very long time. For the past ten years I have been a product engineer, working for companies building one or more products designed to last. I was a rank and file engineer at times, a manager on occasion, and most recently, an architect (meaning I was responsible for technical strategy, but I assure you I wrote a *lot* of code).

What this means is that the experience upon which this book is based comes from actually building software meant to be sustained. I have actually done—and seen the long-term results of doing—pretty much everything in this book. I've been responsible for sustainable software several times over my career.

- I spent four years at an energy startup that sold enterprise software. I saw the product evolve from almost nothing to a successful company with many clients and over 100 engineers. While the software was Java-based, much of what I learned about sustainability applies to the Rails world as well.

- I spent the next year and half at an e-commerce company that had reached what would be the peak of its success. I joined a team of almost 200 engineers, many of whom were working in a huge Rails monolith that contained thousands of lines of code, all done "The Rails Way". The team had experienced massive growth and this growth was not managed. The primary application we all worked in was wholly unsustainable and had a massive carrying cost simply existing.
- I then spent the next six and half years at Stitch Fix, where I was the third engineer and helped set the technical direction for the team. By the time I left, the team was 200 engineers, collectively managing a microservices-based architecture of over 50 Rails applications, many of which I contributed to. At that time I was responsible for the overall technical strategy for the team and was able to observe which decisions we made in 2013 ended up being good (or bad) by 2019.

What I don't have much experience with is working on short-term greenfield projects, or being dropped into a mess to help clean it up (so-called "Rails Rescue" projects). There's nothing wrong with this kind of experience, but that's not what this book is about.

What follows is what I tried to take away from the experience above, from the great decisions my colleagues and I made, to the unfortunate ones as well (I pushed hard for both Coffeescript and Angular 1 and we see how those turned out).

But, as they say, your mileage may vary, "it depends", and everything is a trade-off. Hopefully, I can at least clarify the trade-offs and how to think about them, so if you aren't in the same exact situation as me, you can still get value from my experience.

Up Next

This chapter should've given you a sense of what you're in for and whether or not this book is for you. I hope it is!

So, let's move on. Because this book is about Ruby on Rails, I want to give an overview of the application architecture Rails provides by default, and how those pieces relate to each other. From that basis, we can then deep dive into each part of Rails and learn how to use it sustainably.

2

Business Logic (Does Not Go in Active Records)

Much of this book contains strategies and tactics for managing each part of Rails in a sustainable way. But there is one part of every app that Rails doesn't have a clear answer for: the *business logic*.

Business logic is the term I'm going to use to refer to the core logic of your app that is specific to whatever your app needs to do. If your app needs to send an email every time someone buys a product, but only if that product ships to Vermont, unless it ships from Kansas in which case you send a text message. . . this is business logic.

The biggest question Rails developers often ask is: where does the code for this sort of logic go? Rails doesn't have an explicit answer. There is no `ActiveBusinessLogic::Base` class to inherit from nor is there a `bin/rails generate business-logic` command to invoke.

This chapter outlines a simple strategy to answer this question: do not put business logic in Active Records. Instead, put each bit of logic in its own class, and put all those classes somewhere inside `app/` like `app/services` or `app/businesslogic`.

The reasons don't have to do with moral purity or adherence to some object-oriented design principles. They instead relate directly to sustainability by minimizing the impact of bugs found in business logic.

This chapter is going to walk you through the way I think about it. We'll learn that business logic code is both more complex and less stable than other parts of the codebase. We'll then talk about *fan-in* which is a rough measure of the inter-relations between modules in our system. We'll bring those concepts together to understand how bugs in code used broadly in the app can have a more serious impact than bugs in isolated code.

From there, we'll then be able to speak as objectively as possible about the ramifications of putting business logic in Active Records versus putting it somewhere else.

So, let's jump in. What's so special about business logic?

2.1 Business Logic Makes Your App Special...and Complex

Rails is optimized for so-called *CRUD*, which stands for “Create, Read, Update, and Delete”. In particular, this refers to the database: we create database records, read them back out, update them, and sometimes delete them.

Of course, not every operation our app needs to perform can be thought of as manipulating a database table’s contents. Even when an operation requires making changes to multiple database tables, there is often other logic that has to happen, such as conditional updates, data formatting and manipulation, or API calls to third parties.

This logic can often be complex, because it must bring together all sorts of operations and conditions to achieve the result that the domain requires it to achieve.

This sort of complexity is called *necessary complexity* (or *essential* complexity) because it can’t be avoided. Our app has to meet certain requirements, even if they are highly complex. Managing this complexity is one of the toughest things to do as an app grows.

2.1.1 Business Logic is a Magnet for Complexity

While our code has to implement the necessary complexity, it can often be even more complex due to our decisions about how the logic gets implemented. For example, we may choose to manage user accounts in another application and make API calls to it. We didn’t *have* to do that, and our domain doesn’t require it, but it might be just the way we ended up building it. This kind of complexity is called *accidental* or *unnecessary* complexity.

We can never avoid *all* accidental complexity, but the distinction to necessary complexity is important, because we do have at least limited control over accidental complexity. The better we manage that, the better able we are to manage the code to implement the necessarily complex logic of our app’s domain.

What this means is that the code for our business logic is going to be more complex than other code in our app. It tends to be a magnet for complexity, because it usually contains the necessarily complex details of the domain as well as whatever accidentally complexity that goes along with it.

To make matters worse, business logic also tends to change frequently.

2.1.2 Business Logic Experiences Churn

It’s uncommon for us to build an app and then be done with it. At best, the way we build apps tends to be iterative, where we refine the implementation using feedback cycles to narrow in on the best implementation. Software is notoriously hard to specify, so this feedback cycle tends to work the best. And that means changes, usually in the business logic. Changes are often called *churn*, and areas of the app that require frequent changes have *high churn*.

Churn doesn't necessarily stop after we deliver the first version of the app. We might continue to refine it, as we learn more about the intricacies of the problem domain, or the world around might change, requiring the app to keep up.

This means that the part of our app that is special to our domain has high complexity and high churn. *That* means it's a haven for bugs.

North Carolina State University researcher Nachiappan Nagappan, along with Microsoft employee Richard Ball demonstrated this relationship in their paper "Use of Relative Code Churn Measures to Predict System Defect Density"[1], in which they concluded:

> Increase in relative code churn measures is accompanied by an increase in system defect density [number of bugs per line of code]

Hold this thought for a moment while we learn about another concept in software engineering called *fan-in*.

2.2 Bugs in Commonly-Used Classes Have Wide Effects

Let's talk about the inter-dependence of pieces of code. Some methods are called in only one place in the application, while others are called in multiple places.

Consider a controller method. In most Rails apps, there is only one way a controller method gets called: when an HTTP request is issued to a specific resource with a specific method. For example, we might issue an HTTP GET to the URL `/widgets`. That will invoke the `index` method of the `WidgetsController`.

Now consider the method `find` on `User`. *This* method gets called in *many* more places. In applications that have authentication, it's possible that `User.find` is called on almost every request.

Thus, if there's a problem with `User.find`, most of the app could be affected. On the other hand, a problem in the `index` method of `WidgetsController` will only affect a small part of the app.

We can also look at this concept at the class level. Suppose `User` instances are part of most pieces of code, but we have another model called `WidgetFaxOrder` that is used in only a few places. Again, it stands to

[1]https://www.st.cs.uni-saarland.de/edu/recommendation-systems/papers/ICSE05Churn.pdf

reason that bugs in `User` will have wider effects compared to bugs in `WidgetFaxOrder`.

While there are certain other confounding factors (perhaps `WidgetFaxOrder` is responsible for most of our revenue), this lens of class dependencies is a useful one.

The concepts here are called *fan-out* and *fan-in*. Fan-out is the degree to which one method or class calls into other methods or classes. Fan-in is what I just described above and is the inverse: the degree to which a method or class is *called* by others.

What this means is that bugs in classes or methods with a high fan-in—classes used widely throughout the system—can have a much broader impact on the overall system than bugs in classes with a low fan-in.

Consider the system diagrammed in the figure below. We can see that `WidgetFaxOrder` has a low fan-in, while `Widget` has a high one. `WidgetFaxOrder` has only one incoming "uses" arrow pointing to it. `Widget` has two incoming "uses" arrows, but is also related via Active Record to two other classes.

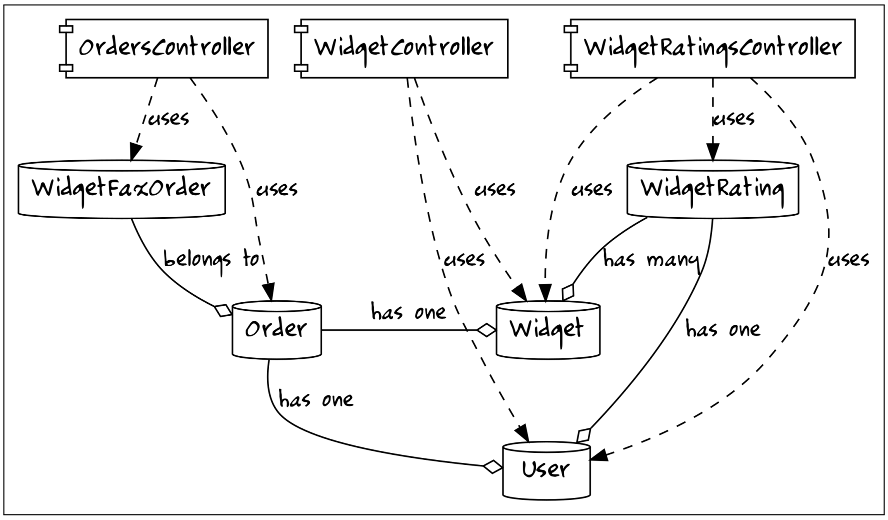

Figure 2.1: System Diagram to Understand Fan-in

Consider a bug in `WidgetFaxOrder`. The figure "Bug Effects of a Low Fan-in Module" on the next page outlines the affected components. This shows that because `WidgetFaxOrder` has a bug, it's possible that `OrdersController` is also buggy, since it relies on `WidgetFaxOrder`. The diagram also shows that it's highly unlikely that any of the rest of the system is affected, because those parts don't call into `WidgetFaxOrder` or any class that does. Thus, we are seeing a worst case scenario for a bug in `WidgetFaxOrder`.

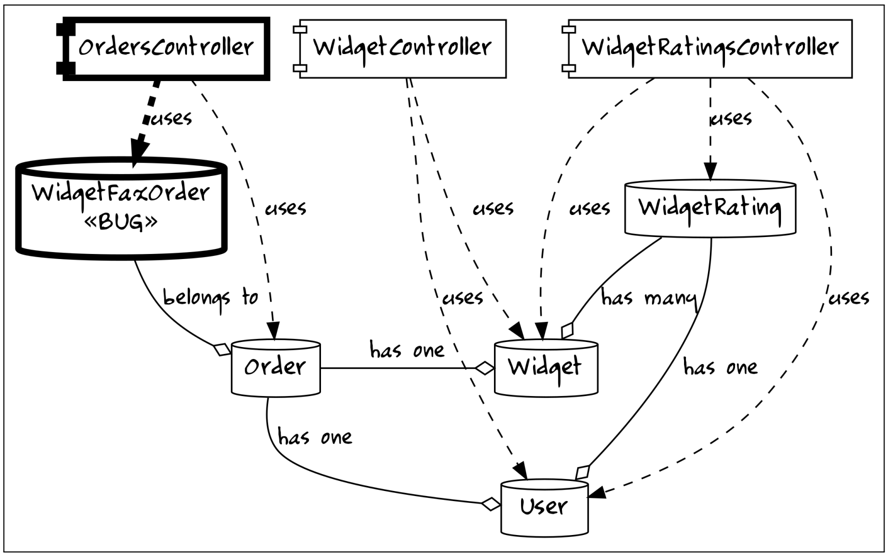

Figure 2.2: Bug Effects of a Low Fan-in Module

Now consider if instead `Widget` has a bug. The figure "Bug Effects of a High Fan-in Module" on the next page shows how a broken `Widget` class could have serious effects throughout the system in the worst case. Because it's used directly by two controllers and possibly indirectly by another through the Active Record relations, the potential for the Widget class to cause a broad problem is much higher than for `WidgetFaxOrder`.

It might seem like you could gain a better understanding of this problem by looking at the method level, but in an even moderately complex system, this is hard to do. The system diagrammed here is vastly simplified.

What this tells me is that the classes that are the most central to the app have the highest potential to cause serious problems. Thus it is important to make sure those classes are working well to prevent these problems.

A great way to do that is to minimize the complexity of those classes as well as to minimize their churn. Do you see where I'm going?

2.3 Business Logic in Active Records Puts Churn and Complexity in Critical Classes

We know that the code that implements business logic is among the most complex code in the app. We know that it's going to have high churn. We know that these two factors mean that business logic code is more likely to have bugs. And we also know that bugs in classes widely used throughout the app can cause more serious systemic problems.

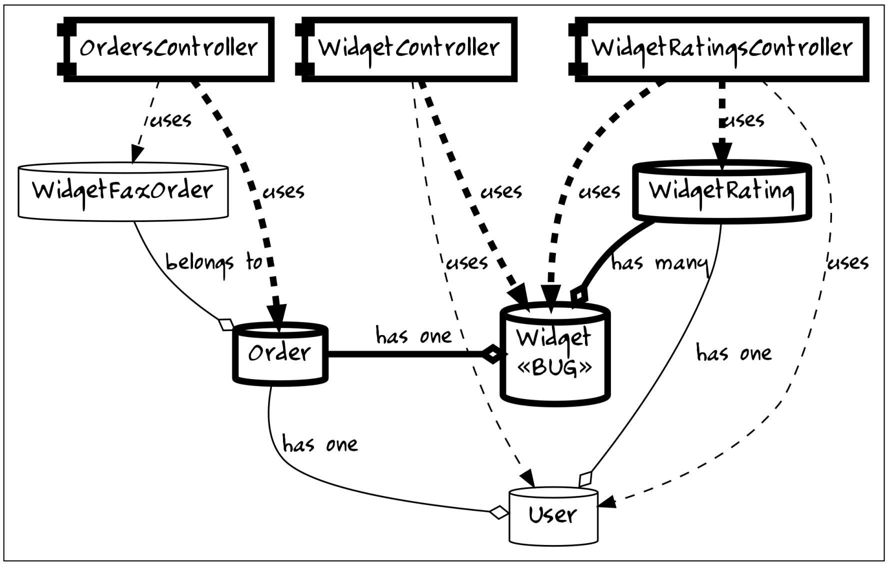

Figure 2.3: Bug Effects of a High Fan-in Module

So why would we put the code most likely to have bugs in the classes most widely used in the system? Wouldn't it be extremely wise to keep the complexity and churn on high fan-in classes—classes used in many places—as low as possible?

If the classes most commonly used throughout the system were very stable, and not complex, we minimize the chances of system-wide bugs caused by one class. If we place the most complex and unstable logic in isolated classes, we minimize the damage that can be done when those classes have bugs, which they surely will.

Let's revise the system diagram to show business logic functions on the Active Records. This will allow us to compare two systems: one in which we place all business logic on the Active Records themselves, and another where that logic is placed on isolated classes.

Suppose that the app shown the diagram has these features:

- Purchase a widget
- Purchase a widget by fax
- Search for a widget
- Show a widget
- Rate a widget
- Suggest a widget rated similar to another widget you rated highly

I've added method names to the Active Records where these might go in the figure "System with Logic on Active Records" on the next page. You might put these methods on different classes or name them differently, but this should look pretty reasonable for an architecture that places business logic on the Active Records.

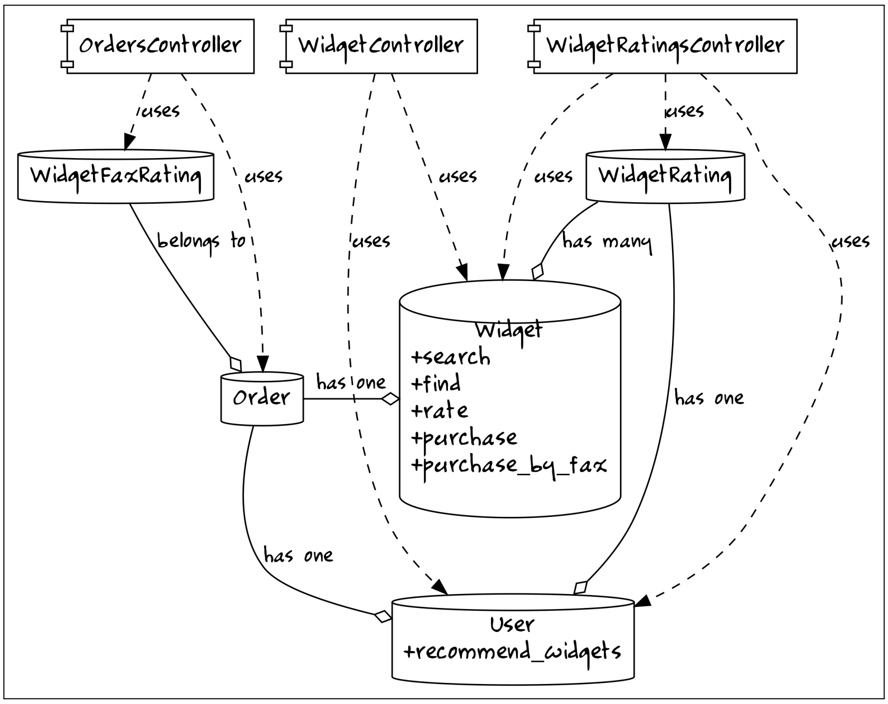

Figure 2.4: System with Logic on Active Records

Now consider an alternative. Suppose that each bit of business logic had its own class apart from the Active Records. These classes accept Active Records as arguments and use the Active Records for database access, but they have all the logic themselves. They form a *service layer* between the controllers and the database. We can see this in the figure below.

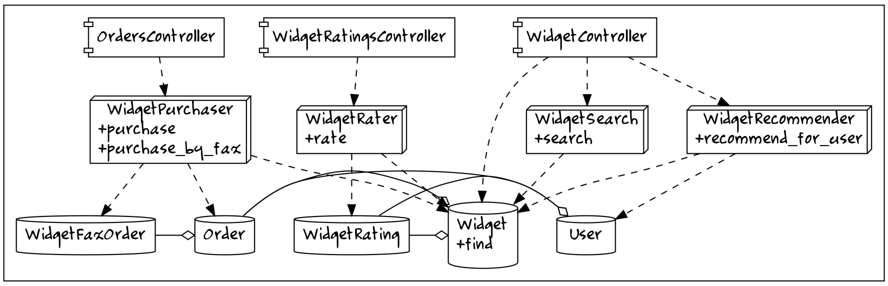

Figure 2.5: System with Business Logic Separated

Granted, there are more classes, so this diagram has more paths and seems more complex, but look at the fan-in of our newly-introduced service layer (the classes in 3-D boxes). All of them have low fan-in. This means that a bug in those classes is likely to be contained. And because those classes are the ones with the business logic—by definition the code likely to contain the most bugs—the effect of those bugs is minimized.

And *this* is why you should not put business logic in your Active Records. There's no escaping a system in which a small number of Active Records are central to the functionality of the app. But we can minimize the damage that can be caused by making those Active Records stable and simple. And to do that, we simply don't put logic on them at all.

There are some nice knock-on effects of this technique as well. The business logic tends to be in isolated classes that embody a domain concept. In our hypothetical system above, one could imagine that `WidgetPurchaser` encapsulates all the logic about purchasing a widget, while `WidgetRecommender` holds the logic about how we recommend widgets.

Both use `Widget` and `User` classes, which don't represent any particular domain concept beyond the attributes we wish to store in the database. And, as the app grows in size and features, as we get more and more domain concepts which require code, the `Widget` and `User` classes won't grow proportionally. Neither will `WidgetRecommender` nor `WidgetPurchaser`. Instead, we'll have new classes to represent those concepts.

In the end, you'll have a system where churn is isolated to a small number of classes, depended-upon by a few number of classes. This makes changes safer, more reliable, and easier to do. That's sustainable.

Let's see an example.

2.4 Example Design of a Feature

Suppose we are building a feature to edit widgets. Here is a rough outline of the requirements around how it should work:

1. A user views a form where they can edit a widget's metadata.
2. The user submits the form with a validation error.
3. The form is re-rendered showing their errors.
4. The user corrects the error and submits the edit again.
5. The system then updates the database.
6. When the widget is updated, two things have to happen:
    1. Depending on the widget's manufacturer, we need to notify an admin to approve of the changes
    2. If the widget is of a particular type, we must update an inventory table used for reporting.
7. The user sees a result screen.
8. Eventually, an email is sent to the right person.

This is not an uncommon amount of complexity. We will have to write a bit of code to make this work, and it's necessarily going to be in several places. A controller will need to receive the HTTP request, a view will need to render the form, a model must help with validation, a mailer will need to be created for the emails we'll send and somewhere in there we have a bit of our own logic.

The figure below shows the classes and files that would be involved in this feature. `WidgetEditingService` is probably sticking out to you.

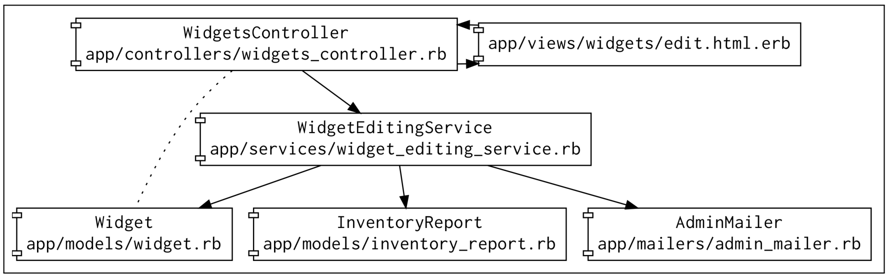
WidgetsController
app/controllers/widgets_controller.rb
app/views/widgets/edit.html.erb
WidgetEditingService
app/services/widget_editing_service.rb
Widget
app/models/widget.rb
InventoryReport
app/models/inventory_report.rb
AdminMailer
app/mailers/admin_mailer.rb

Figure 2.6: Class Design of Feature

Here's what that class might look like:

```
class WidgetEditingService
  def edit_widget(widget, widget_params)
    widget.update(widget_params)

    if widget.valid?
      # create the InventoryReport
      # check the manufacturer to see who to notify
      # trigger the AdminMailer to notify whoever
      # should be notified
    end

    widget
  end
end
```

The code in the other classes would be relatively vanilla Rails stuff. `WidgetsController` looks how you'd expect:

```
class WidgetsController < ApplicationController
  def edit
```

```
    @widget = Widget.find(params[:id])
  end

  def update
    widget  = Widget.find(params[:id])
    @widget = WidgetEditingService.new.edit_widget(
                widget, widget_params
              )
    if @widget.valid?
      redirect_to widgets_path
    else
      render :edit
    end
  end

private
  def widget_params
    params.require(:widget).permit(:name, :status, :type)
  end
end
```

`Widget` will have a few validations:

```
class Widget < ApplicationRecord
  validates :name, presence: true
end
```

`InventoryReport` is almost nothing:

```
class InventoryReport < ApplicationRecord
end
```

`AdminMailer` has methods that just render mail:

```
class AdminMailer < ApplicationMailer
  def edited_widget(widget)
    @wiget = widget
  end
  def edited_widget_for_supervisor(widget)
    @widget = widget
  end
end
```

Note that just about everything about editing a widget is in `WidgetEditingService` (which also means that the test of this class will almost totally specify the business process in one place). `widget_params` and the validations in `Widget` *do* constitute a form of business logic, but to co-locate those in `WidgetEditingService` would be giving up a *lot*. There's a huge benefit to using strong parameters and Rails' validations. So we do!

Let's see how this survives a somewhat radical change. Suppose that the logic around choosing who to notify and updating the inventory record are becoming too slow, and we decide to execute that logic in a background job—the user editing the widget doesn't really care about this part anyway.

The figure below shows the minimal change we'd make. The highlighted classes are all that needs to change.

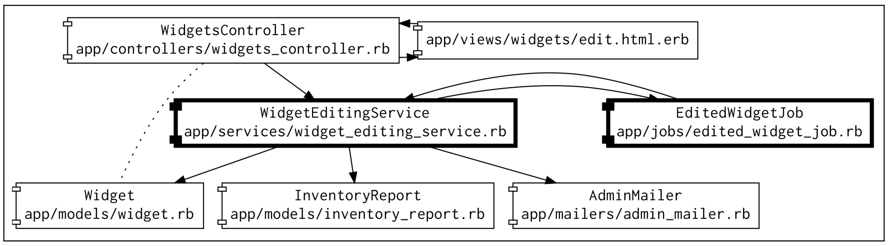
WidgetsController
app/controllers/widgets_controller.rb
app/views/widgets/edit.html.erb
WidgetEditingService
app/services/widget_editing_service.rb
EditedWidgetJob
app/jobs/edited_widget_job.rb
Widget
app/models/widget.rb
InventoryReport
app/models/inventory_report.rb
AdminMailer
app/mailers/admin_mailer.rb

Figure 2.7: Design with a Background Job Added

We might imagine that `WidgetEditingService` is now made up of two methods, one that's called from the controller and now queues a background and a new, second method that the background job will call that contains the logic we are backgrounding.

```
class WidgetEditingService
  def edit_widget(widget, widget_params)
    widget.update(widget_params)

    if widget.valid?
      EditedWidgetJob.perform_later(widget.id)
    end

    widget
  end
  def post_widget_edit(widget)
    # create the InventoryReport
    # check the manufacturer to see who to notify
    # trigger the AdminMailer to notify whoever
    # should be notified
  end
end
```

The `EditedWidgetJob` is just a way to run code in the background:

```
class EditedWidgetJob < ApplicationJob
  def perform(widget_id)
    widget = Widget.find(widget_id)
    WidgetEditingService.new.post_widget_edit(widget)
  end
end
```

As you can see, we're putting only the code in the background job that *has* to be there. The background job is given an ID and must trigger logic. And that's all it's doing.

I'm not going to claim this is beautiful code. I'm not going to claim this adheres to object-oriented design principles…whatever those are. I'm also not going to claim this is how DHH would do it.

What I will claim is that this approach allows you to get a *ton* of value out of Rails, while also allowing you to consolidate and organize your business logic however you like. And this will keep that logic from getting intertwined with HTTP requests, email, databases, and anything else that's provided by Rails. And *this* will help greatly with sustainability.

Do note that the "service layer" a) can be called something else, and b) can be designed any way you like yet still reap these benefits. While I would encourage you to write boring procedural code as I have done (and I'll make the case for it in "Business Logic Class Design" on page **??**), you can use any design you like.

Up Next

This will be helpful context about what's to come. Even when isolating business logic in standalone classes, there's still gonna be a fair bit of code elsewhere in the app. A lot of it ends up where we're about to head: the view. And the first view of your app that anyone ever sees is the URL, so we'll begin our deep-dive into Rails with routes.

3

Jobs

One of the most powerful tools to make your app high-performing and fault-tolerant is the background job. Background jobs bring some complexity and carrying cost to the system, so you have to be careful not to swap one sustainability problem for another.

This chapter will help you navigate this part of Rails. We'll start by understanding exactly what problems background jobs exist to solve. We'll then learn why you must understand exactly how your chosen job backend (Sidekiq, Resque, etc.) works. We'll set up Sidekiq in our example app, since Sidekiq is a great choice if you don't have specific requirements otherwise.

We'll then learn how to use, build, and test jobs. After all that we'll talk about a big source of complexity around background jobs, which is making them idempotent. Jobs can and will be automatically retried and you don't usually want their effects to be repeated. Achieving idempotency is not easy or even possible in every situation.

Let's jump into it. What problems do background jobs solve?

3.1 Use Jobs To Defer Execution or Increase Fault-Tolerance

Background jobs allow you to run code outside a web request/response cycle. Sometimes you do this because you need to run some batch process on a schedule. There are two other reasons we're going to focus on, since they lead to the sort of complexity you have to carefully manage. Background jobs can allow moving non-critical code to outside the request/response cycle as well as encapsulate flaky code that may need several retries in order to succeed.

Both of these situations amount to deferring code that might take too long to a background job to run later. The reason this is important has to do with how your Rails app is set up in production.

3.1.1 Web Workers, Worker Pools, Memory, and Compute Power

In development, your Rails app uses the Puma[1] web server. This server receives requests and dispatches them to your Rails app (this is likely how it works in production as well). When a request comes in, Puma allocates a *worker* to handle that request. That worker works on only that request until a response is rendered—it can't manage more than one request at a time.

When the response is rendered, the worker can work on another request. Puma keeps these workers in a *pool*, and that pool has a finite limit. This is because each worker consumes memory and CPU just merely existing, and because memory and CPU are finite resources, there can only be so many workers per server.

What if all workers are handling requests? What happens to a new request that comes in when there is no worker to handle it?

It depends. In some configurations, the new request will be denied and the browser will receive an HTTP 503 (resource unavailable). In other configurations that request will be placed in a queue (itself a finite resource) to be handled whenever a worker becomes available. In this case the request will appear to be handled more slowly than usual.

While you can increase the overall number of workers through complex mechanisms such as load balancers, there is always going to be a finite amount of resources to process requests. Often this limit is financial, not technical, since more servers and more infrastructure cost more money and it may not be worth it.

Another solution to the problem of limited workers is to reduce the amount of work those workers have to do. If your controller initiates a business process that takes 500ms normally, but can be made to defer 250ms of that process into a background job, you will have doubled your worker capacity[2].

One particular type of code that leads to poor performance—and thus is a good target for moving to a background job—is code that interacts with third party APIs, such as sending email or processing payments.

3.1.2 Network Calls and Third Parties are Slow

Although our app doesn't have the ability to charge users to purchase widgets, you might imagine that it could, and that means integrating with a payment processor. And *this* means making a network call over the Internet. Although network calls within our data center can fail, network calls over the Internet are so likely to fail that you have to handle that failure as a first-order issue.

[1]https://puma.io

[2]Yes, this is vastly oversimplified, but the point stands.

Of course, network calls that fail don't fail immediately. They often fail after an interminable amount of time. Or not. Sometimes the network is just slow and a successful result eventually comes back.

Background jobs can help solve this problem. The figure below outlines how this works.

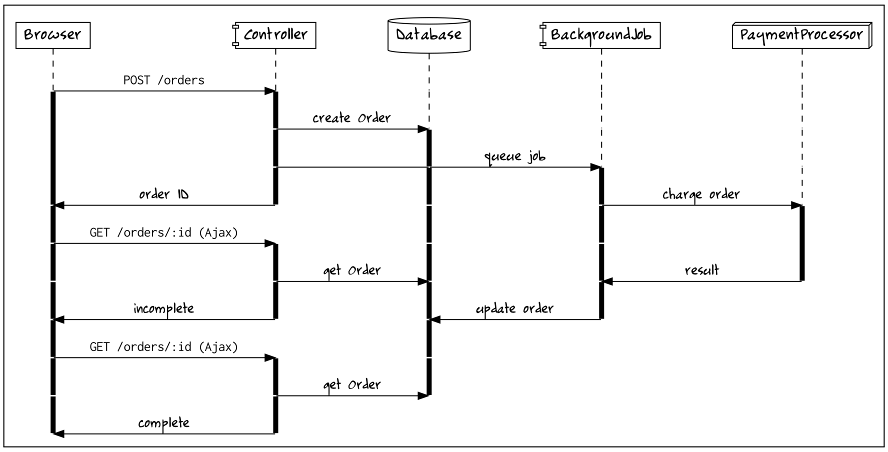

Figure 3.1: Performing Slow Code in Background Jobs

In the figure, you can see that the initial POST to create an order causes the controller to insert an order into the database then queue a background job to handle communicating with the payment processor. While that's happening, the controller returns the order ID to the browser.

The browser then uses Ajax to poll the controller's show method to check on the status of the order. The show method will fetch the order from the database to see if it's been processed. Meanwhile, the background job waits for the payment processor until it receives a response. When it does, it updates the order in the database. Eventually, the browser will ask about the order and receive a response that it's completed.

This may seem complex, but it allows the web workers (which are executing only the controller code in this example) to avoid waiting on the slow payment provider.

This design can also handle transient errors that might happen communicating with the third party. The job can be automatically retried without having to change how the front-end works.

3.1.3 Network Calls and Third Parties are Flaky

Network calls fail. There's just no way to prevent that. The farther away another server is from *your* server, the more likely it is to fail, and even at small scale, network failures happen frequently.

In most cases, network failures are transient errors. Retrying the request usually results in a success. But retrying network requests can take a while, since network requests don't fail fast. Your background jobs can handle this.

The figure below shows how this might work.

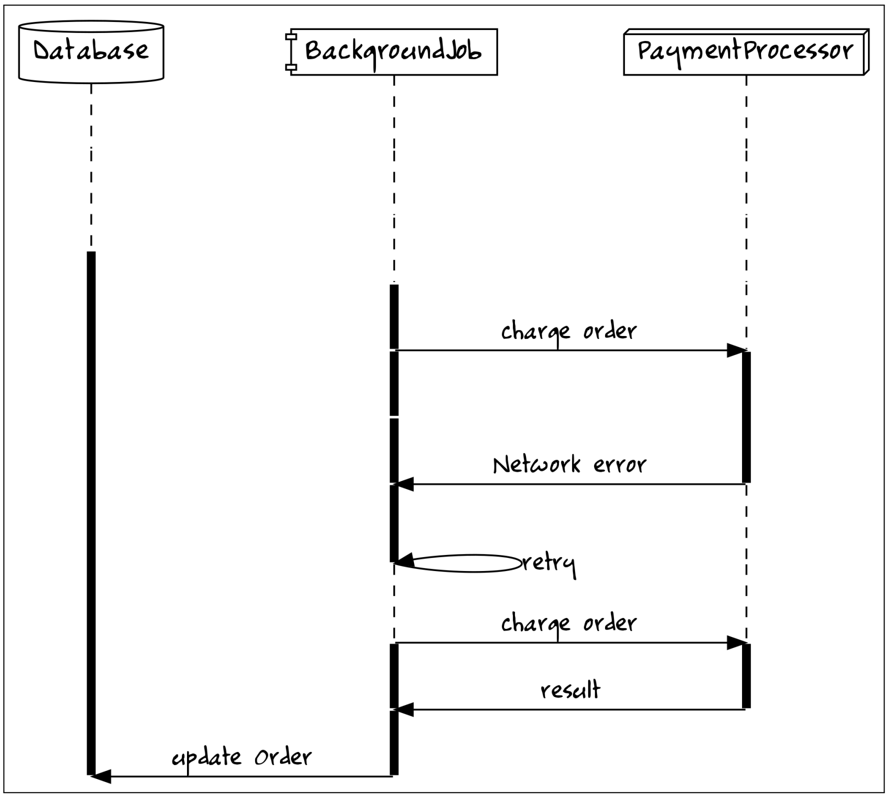

Figure 3.2: Retrying a Failed Job

When our job encounters a network error, it can retry itself. During this retry, the front-end is still diligently asking for an update. In this case it waits a bit longer, but we don't have to re-architect how the entire feature works.

This might all seem quite complex and, well, it is. The rest of this chapter will identify sources of complexity and strategies to work around them, but it's important that you use background jobs only when needed.

3.1.4 Use Background Jobs Only When Needed

At a certain scale, the benefits of background jobs outweigh their complexity, and you'd be wise to use them as much as possible. You likely aren't at that scale now, and might never be. Thus, you want to be judicious when you use background jobs.

The two main problems that happen when you do all processing in the request are over-use of resources and failures due to network timeouts. Thus, your use of background jobs should be when you cannot tolerate these failures at whatever level you are seeing them.

This can be hard to judge. A guideline that I adopt is to always communicate with third parties in a background job, because even at tiny scale, those communications will fail.

For all other code, it's best to monitor its performance, set a limit on how poor the performance is allowed to get, and use background jobs when performance gets bad (keeping in mind that background jobs aren't the only solution to poor performance). For example, you might decide that the 90th percentile of controller action response times should always be under 500ms.

When you *are* going to use background jobs, you need to understand how the underlying system actually works to avoid surprises.

3.2 Understand How Your Job Backend Works

Rails includes a library called Active Job that provides an abstraction layer over queueing and implementing jobs. Since it is not a job queueing system itself, it unfortunately does not save you from having to understand whatever system—called a *backend*—you have chosen. Be it Sidekiq, Sucker Punch, Resque, or something else, each job backend has different behaviors that are critical to understand.

For example, Resque does not automatically retry failed jobs, but Sidekiq does. Que uses the database to store jobs, but Sidekiq uses Redis (meaning you need to have a Redis database set up to use Sidekiq and also understand what a Redis database actually is). And, of course, the default queuing system in Rails is nothing, so jobs don't run in the background without setting something up.

Here is what you need to know about the job backend you are using:

- How does queueing work?
  - How are the jobs themselves stored?
  - Where are they stored?
  - How are the arguments to the jobs encoded while jobs wait to execute?
- What happens when a job fails?
- How can you observe what's happening in the job backend?

3.2.1 Understand Where and How Jobs (and their Arguments) are Queued

When you queue a job with Sucker Punch, the job is stored in memory. Another process with access to that memory will pluck the job out of an internal queue and execute it. If you use Sidekiq, the job goes into Redis. The job class and the arguments passed to it are converted into JSON before storing, and converted back before the job runs.

It's important to know where the jobs are stored so you can accurately predict failure modes. In the case of Sucker Punch, if your app's process dies for some reason, any unprocessed job is gone without a trace.

In the case of Sidekiq (or Resque), you may lose jobs if Redis goes down, depending on how Redis is configured. If you are also using that Redis for caching, you then run the risk of using up all of the storage available on caching and will be unable to queue jobs at all.

You also need to know the mechanism by which the jobs are stored wherever they are stored. For example, when you queue a job for Sidekiq, it will store the name of the job class as a string, and all of the arguments as an array. Each argument will be converted to JSON before being stored. When the job is executed, those JSON blobs will be parsed into hashes.

This means that if you write code like this:

```
ChargeMoneyForWidgetJob.perform_async(widget)
```

The code in `ChargeMoneyForWidgetJob` will not be given a `Widget`, but instead be given a `Hash` containing whatever results from calling `to_json` on a `Widget`. Many developers find this surprising, and this is precisely why you have to understand how jobs are stored.

You also need to know what happens when jobs fail.

3.2.2 Understand What Happens When a Job Fails

When a job encounters an exception it doesn't rescue, it fails. Unlike a web request in a similar situation, which sends an HTTP 500 to the browser, the job has no client to report its failure to. Each job backend handles this situation differently by default, and has different options for modifying the default behavior.

For example, Sucker Punch does nothing by default, and failed jobs are simply discarded. Sidekiq will automatically retry them for a period of time before discarding them. Resque will place them into a special failed queue and hope you notice.

As discussed above, the ability to retry in the face of failures is one of the reasons to place code in a background job. My advice is to understand how failure is managed and then configure your jobs system and/or jobs to automatically retry a certain number of times before loudly notifying you of the job failure.

It's common for job backends to integrate with exception notification services like Bugsnag or Rollbar. You need to understand exactly how this integration works. For example, Resque will notify you once before placing the job in the failed queue. Sidekiq will notify you every time the job fails, even if that job is going to be retried.

I can't give specific advice, because it depends on what you have chosen, but you want to arrange for a situation in which you are notified when a job that should complete has failed and won't be retried. You *don't* want notification when a job fails and will be retried, nor do you need to know if a job fails whose failure doesn't matter.

Failure is a big part of the next thing you need to know, which is how to observe the behavior of the job backend.

3.2.3 Observe the Behavior of Your Job Backend

When a job fails and won't be retried, you need a way to examine that job. What class was it? What were the arguments passed to it? What was the reason for failure? You also need to know how much capacity you have used storing jobs, as well as how many and what type of jobs are waiting to be processed. You may also wish to know what jobs have failed and *will* be retried, and when they might get retried.

Many job backends come with a web UI that can tell you this. Some also include programmatic APIs you can use to inspect the job backend. Familiarize yourself with whatever is provided and make sure you use it. If there is a web UI, make sure only authorized users can access it, and make sure you understand what it's showing you.

The more you can connect your job backend's metrics to a monitoring system, the better. It can be extremely hard to diagnose problems that result from the job backend failing if you can't observe its behavior.

I have personally used Que, Resque, Sucker Punch, and Sidekiq. Of those four, Sidekiq is the best choice for most situations and if you aren't sure which job backend to use, choose Sidekiq.

We'll need to write some job code later on, so we need some sort of backend set up. Let's set up Sidekiq.

3.3 Sidekiq is The Best Job Backend for Most Teams

I'm going to go quickly through this setup. Sidekiq's documentation is great and can provide you with many details about how it works. This point of this chapter is to talk about job code, not Sidekiq, but we need something set up, and I want to use something that is both realistic and substantial. You are likely to encounter Sidekiq in the real world, and you are very likely to encounter a complex job backend configuration.

First, we'll add the Sidekiq gem to `Gemfile`:

```
# Gemfile

  # lograge changes Rails' logging to a more
  # traditional one-line-per-event format
  gem "lograge"
→
→ # Sidekiq handles background jobs
→ gem "sidekiq"

  # Bundle edge Rails instead: gem 'rails', github: 'rails/rail. . .
  gem 'rails', '~> 6.0.3', '>= 6.0.3.4'
```

Then install it:

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

We will also need to create the binstub so we can run it if we need to:

```
> bundle binstub sidekiq
The dependency tzinfo-data (>= 0) will be unused by any of t. . .
```

Sidekiq assumes Redis is running on `localhost` by default. Assuming you are using the Docker-based setup I recommended, our Redis is running on port 6379 of the host `redis`, so we need to tell Sidekiq about that. Remembering what we learned in "Using The Environment for Runtime Configuration" on page **??**, we want this URL configured via the environment. Let's add that to our two `.env` files.

First, is `.env.development`:

```
# .env.development

  DATABASE_URL="
    postgres://postgres:postgres@db:5432/widgets_development"
→ SIDEKIQ_REDIS_URL=redis://redis:6379/1
```

The value `redis` for the host comes from key used in the `docker-compose.yml` file to set up Redis. For the test environment, we'll do something similar, but instead of `/1` we'll use `/2`, which is a different logical database inside the Redis instance.

```
# .env.test

  DATABASE_URL=postgres://postgres:postgres@db:5432/widgets_tes. . .
→ SIDEKIQ_REDIS_URL=redis://redis:6379/2
```

Note that we put "SIDEKIQ" in the name to indicate the purpose of this Redis. You should not use the same Redis instances for both job queueing and caching if you can help it. The reason is that it creates a single point of failure for two unrelated activities. You don't want a situation where you start aggressively caching and use up your storage preventing jobs from being queued.

Now, we'll create an initializer for Sidekiq that uses this new enviornment variable:

```
# config/initializers/sidekiq.rb

Sidekiq.configure_server do |config|
  config.redis = {
    url: ENV.fetch("SIDEKIQ_REDIS_URL")
  }
end

Sidekiq.configure_client do |config|
  config.redis = {
    url: ENV.fetch("SIDEKIQ_REDIS_URL")
  }
end
```

Note that we used `fetch` because it will raise an error if the value `SIDEKIQ_REDIS_URL` is not found in the environment. This will alert us if we forget to set this in production.

We don't need to actually *run* Sidekiq in this chapter, but we should set it up. This is going to require that `bin/run` start two simultaneous processes: the Rails server we are already using and the Sidekiq worker process. To do *that* we'll use Foreman[3], which we'll add to the development and test sections of our `Gemfile`:

```
# Gemfile

    # We use Factory Bot in place of fixtures
    # to generate realistic test data
    gem "factory_bot_rails"
→
→   # Foreman runs all processes for local development
→   gem "foreman"

    # We use Faker to generate values for attributes
    # in each factory
```

We can install it:

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

We also need to create a binstub in `bin/` for it:

```
> bundle binstub foreman
The dependency tzinfo-data (>= 0) will be unused by any of t. . .
```

Foreman uses a "Procfile" to know what to run. The Procfile lists out all the processes needed to run our app. Rather than create this file, I prefer to generate it inside `bin/run`. This centralizes the way we run our app to a single file, which is more mangeable as our app gets more complex. I also prefer to name this file `Procfile.dev` so it's clear what it's for (services like Heroku use `Procfile` to know what to run in production). Let's replace `bin/run` with the following:

```
# bin/run

#!/usr/bin/env bash

set -e
```

[3]https://ddollar.github.io/foreman/

```
echo "[ bin/run ] Rebuilding Procfile.dev"
echo "# This is generated by bin/run. Do not edit" > Procfile.dev
echo "# Use this via bin/run" >> Procfile.dev
# We must bind to 0.0.0.0 inside a
# Docker container or the port won't forward
echo "web: bin/rails server --binding=0.0.0.0" >> Procfile.dev
echo "sidekiq: bin/sidekiq" >> Procfile.dev

echo "[ bin/run ] Starting foreman"
bin/foreman start -f Procfile.dev -p 3000
```

We'll also add `Procfile.dev` to our `.gitignore` file:

```
# .gitignore

  # The .env file is used for both dev and test
  # and creates more problems than it solves
  .env
→
→ # Procfile.dev is generated, so should not be checked in
→ Procfile.dev

  # .env.*.local files are where we put actual
  # secrets we need for dev and test, so
```

Now, when we run our app with `bin/run`, Sidekiq will be started as well and any code that requires background job processing will work in development.

Let's talk about how to queue jobs and how to implement them.

3.4 Queue Jobs Directly, and Have Them Defer to Your Business Logic Code

Once you know how your job backend works and when to use a background job, how do you write one and how do you invoke it?

Let's talk about invocation first.

3.4.1 Do Not Use Active Job - Use the Job Backend Directly

Active Job was added to Rails in recent years as a single abstraction over background jobs. This provides a way for library authors to interact with background jobs without having to know about the underlying backend.

Active Job does a great job at this, but since you *aren't* writing library code, it creates some complexities that won't provide much value in return. Since Active Job doesn't alleviate you from having to understand your job backend, there isn't a strong reason to use it.

The main source of complexity is the way in which arguments to jobs are handled. As discussed above, you need to know how those arguments are serialized into whatever data store your job system is using. Often, that means JSON.

This means that you can't pass an Active Record directly to a job since it won't serialize/de-serialize properly:

```
> bin/rails c
rails-console> require "pp"
rails-console> widget = Widget.first
rails-console> pp JSON.parse(widget.to_json) ; nil
{"id"=>1,
 "name"=>"Stembolt",
 "price_cents"=>102735,
 "widget_status_id"=>2,
 "manufacturer_id"=>11,
 "created_at"=>"2020-05-24T22:02:54.571Z",
 "updated_at"=>"2020-05-24T22:02:54.571Z"}
=> nil
```

Before Active Job, the solution to this problem was to pass the widget ID to the job, and have the job look up the `Widget` from the database. Active Job uses globalid[4] to automate this process for you. But only for Active Records and only when using Active Job.

That means that when you are writing code to queue a job, you have to think about what you are passing to that job. You need to know what type of argument is being passed, and whether or not it uses globalid. I don't like having to think about things like this while I'm coding and I don't see a lot of value in return for doing so.

Unless you are using multiple job backends—which will create a sustainability problem for you and your team—use the API of the job backend you have chosen. That means that your arguments should almost always be basic types, in particular database identifiers for Active Records.

Let's see that with our existing widget creation code. We'll move the logic around emailing finance and admin to a background job called `PostWidgetCreationJob`, which we'll write in a moment. We'll use it like so:

[4]https://github.com/rails/globalid

```
# app/services/widget_creator.rb

      widget.save
      if widget.invalid?
        return Result.new(created: false, widget: widget)
      end
× #     if widget.price_cents > 7_500_00
× #       FinanceMailer.high_priced_widget(widget).deliver_now
× #     end
  # XXX
× #     if widget.manufacturer.created_at.after?(60.days.ago)
× #       AdminMailer.new_widget_from_new_manufacturer(widget).
× #         deliver_now
× #     end
  # XXX
× #     Result.new(created: widget.valid?, widget: widget)
→     PostWidgetCreationJob.perform_async(widget.id)
→     Result.new(created: widget.valid?, widget: widget)
    end

    class Result
```

`perform_async` is Sidekiq's API, and we have to pass `widget.id` for reasons stated above. We'll talk about where the code we just removed goes next.

3.4.2 Job Code Should Defer to Your Service Layer

For all the reasons we don't want business logic in our controllers, we don't want business logic in our jobs. And for all the reasons we want to convert the raw data types being passed into richly-typed objects in our controllers, we want to do that in our jobs, too.

We passed in a widget ID to our job, which means our job should locate the widget. After that, it should defer to another class that implements the business logic.

Since this is still widget creation and the job is called `PostWidgetCreationJob`, we'll create a new method on `WidgetCreator` called `post_widget_creation` and have the job trigger that.

Let's write the job code and then fill in the new method. Since we aren't using Active Job, we can't use `bin/rails g job`. We also can't use `ApplicationJob` in its current form, so let's replace it with one that works for Sidekiq.

```
# app/jobs/application_job.rb

# Do not inherit from ActiveJob. All jobs use Sidekiq
class ApplicationJob
  include Sidekiq::Worker

  sidekiq_options backtrace: true
end
```

Now, any job we create that extends `ApplicationJob` will be set up for Sidekiq and we won't have to include `Sidekiq::Worker` in every single class. We could customize the output of `bin/rails g job` by creating the file `lib/templates/rails/job/job.rb.tt`, but we aren't going to use this generator at all. The reason is that our job class will be very small and we won't write a test for it.

Here's what `PostWidgetCreationJob` looks like:

```
# app/jobs/post_widget_creation_job.rb

class PostWidgetCreationJob < ApplicationJob
  def perform(widget_id)
    widget = Widget.find(widget_id)
    WidgetCreator.new.post_widget_creation_job(widget)
  end
end
```

This means we need to create the method `post_widget_creation_job` in `WidgetCreator`, which will contain the code we removed from `create_widget`:

```
# app/services/widget_creator.rb

      Result.new(created: widget.valid?, widget: widget)
    end

→   def post_widget_creation_job(widget)
→     if widget.price_cents > 7_500_00
→       FinanceMailer.high_priced_widget(widget).deliver_now
→     end
→
→     if widget.manufacturer.created_at.after?(60.days.ago)
→       AdminMailer.new_widget_from_new_manufacturer(widget).
→         deliver_now
→     end
→   end
→
    class Result
      attr_reader :widget
      def initialize(created:, widget:)
```

Our app should still work, but we've lost the proof of this via our tests. Let's talk about that next.

3.5 Job Testing Strategies

In the previous section, I said we wouldn't be writing a test for our Job. Given the implementation, I find a test that the job simply calls a method to have low value and high carrying cost. But, we do need coverage that whatever uses the job is working correctly.

There are three approaches to take regarding testing code that uses jobs, assuming your chosen job backend supports them. You can run jobs synchronously inline, you can store jobs in an internal data structure, executing them manually inside a test, or you can allow the jobs to actually go into a real queue to be executed by the real job system.

Which one to use depends on a few things.

Executing jobs synchronously as they are queued is a good technique when the jobs have simple arguments using types like strings or numbers *and* when the job is incidental to the code under test. Our widget creation code falls under this category. There's nothing inherent to widget creation that implies the use of jobs.

Queuing jobs to an internal data structure, examining it, and then executing the jobs manually is more appropriate if the code you are testing is inherently about jobs. In this case, the test serves as a clear set of assertions about what jobs get queued when. A complex batch process whereby you need to fetch a lot of data, then queue jobs to handle it, would be a good candidate for this sort of approach.

This approach is also good when your job arguments are somewhat complex. The reason is that queuing the jobs to an internal structure usually serializes them, so this will allow you to detect bugs in your assumptions about how arguments are serialized. It is *incredibly* common to pass in a hash with symbols for keys and then erroneously expect symbols to come out of the job backend (when, in fact, the keys will likely be strings).

The third option—using the job backend in a production-like mode—is expensive. It requires running a worker to process the jobs outside of your tests (or having your test trigger that worker somehow) and requires that the job data storage system be running *and* be reset on each new test run, just as Rails resets the database for you.

I try to avoid this option if possible unless there is something so specific about the way jobs are queued and processed that I can only detect it by running the actual job backend itself.

For our code, the first approach works, and Sidekiq provides a way to do that. We will require `"sidekiq/testing"` in `test/test_helper.rb` and then call `Sidekiq::Testing.inline!` around our test.

First, however, let's make sure our test is actually failing:

```
> bin/rails test test/services/widget_creator_test.rb || echo  \
  Test Failed
Run options: --seed 17819

# Running:

...F

Failure:
WidgetCreatorTest#test_finance_is_notified_for_widgets_price. . .
Expected: 1
  Actual: 0

rails test test/services/widget_creator_test.rb:44

...F

Failure:
WidgetCreatorTest#test_email_adming_staff_for_widgets_on_new. . .
Expected: 1
  Actual: 0

rails test test/services/widget_creator_test.rb:126



Finished in 0.712802s, 11.2233 runs/s, 30.8641 assertions/s.
8 runs, 22 assertions, 2 failures, 0 errors, 0 skips
Test Failed
```

Good. It's failing in the right ways. You can see that the expected effects of the code we removed aren't happening and this causes the test failures. When we set Sidekiq up to run the job we are queuing inline, the tests should start passing.

Let’s start with `test/test_helper.rb`:

```
# test/test_helper.rb

  ENV['RAILS_ENV'] ||= 'test'
  require_relative '../config/environment'
  require 'rails/test_help'
→
→ # Set up Sidekiq testing modes. See
→ # https://github.com/mperham/sidekiq/wiki/Testing
→ require "sidekiq/testing"

  require "support/confidence_check"
```

Sidekiq’s default behavior is the second approach of queueing jobs to an internal data structure. To run them inline, we’ll use `Sidekiq::Testing.inline!`. We’ll add this to the `setup` block in `test/services/widget_creator_test.rb`:

```
# test/services/widget_creator_test.rb


  class WidgetCreatorTest < ActiveSupport::TestCase
    setup do
→     Sidekiq::Testing.inline!
      ActionMailer::Base.deliveries = []
      @widget_creator = WidgetCreator.new
      @manufacturer = FactoryBot.create(:manufacturer,
```

We need to undo this setting after our tests run in case other tests are relying on the default (which they shouldn’t, but it’s still a good idea to undo anything done in a `setup` block):

```
# test/services/widget_creator_test.rb

      FactoryBot.create(:widget_status)
      FactoryBot.create(:widget_status, name: "Fresh")
    end
→   teardown do
→     Sidekiq::Testing.fake!
→   end
    test "widgets have a default status of 'Fresh'" do
      result = @widget_creator.create_widget(Widget.new(
        name: "Stembolt",
```

Now, our test should pass:

```
> bin/rails test test/services/widget_creator_test.rb
Run options: --seed 43635

# Running:

........

Finished in 0.632866s, 12.6409 runs/s, 47.4034 assertions/s.
8 runs, 30 assertions, 0 failures, 0 errors, 0 skips
```

To use the second testing strategy—allowing the jobs to queue and running them manually—consult your job backend's documentation. Sidekiq provides methods to do all this for you if you should choose.

Now that we've seen how to make our code work using jobs, we have to discuss another painful reality about background jobs, which is retries and idempotence.

3.6 Jobs Will Get Retried and Must Be Idempotent

One of the reasons we use background jobs is to allow them to be retried automatically when a transient error occurs. While you could build up a list of transient errors and only retry them, this turns out to be difficult, because there are a lot of errors that one would consider transient. It is easier to configure your jobs to automatically retry all errors (or at least retry them several time before finally failing).

This means that code executed from a job must be idempotent: it must not have its effect felt more than once, no matter how many times it's executed.

Consider this code that updates a widget's `updated_at`[5]

```
def touch(widget)
  widget.updated_at = Time.zone.now
```

[5]I realize you would never actually write this, but idempotence is worth explaining via a trivial example as it is not a concept that comes naturally to most.

```
  widget.save!
end
```

Each time this is called, the widget's `updated_at` will get a new value. That means this method is not idempotent. To make it idempotent, we would need to pass in the date:

```
def touch(widget, updated_at)
  widget.updated_at = updated_at
  widget.save!
end
```

Now, no matter how many times we call `touch` with the same arguments, the effect will be the same.

The code initiated by our jobs must work similarly. Consider a job that charges someone money for a purchase. If there were to be a transient error partway through, and we retried the entire job, the customer could be charged twice. *And* we might not even be aware of it unless the customer noticed and complained!

Making code idempotent is not easy. It's also—you guessed it—a trade-off. The `touch` method above probably won't cause any problems if it's not idempotent. But charging someone money will. This means that you have to understand what might fail in your job, what might happen if it's retried, how likely that is to happen, and how serious it is if it does.

This means that your job is going to be idempotent with respect to some failure modes, and not to others. This is OK if you are aware of it and make the conscious decision to allow certain scenarios to not be idempotent.

Let's examine the job we created in the last section. It's called `post_widget_creation_job` in `WidgetCreator`, which looks like so:

```
def post_widget_creation_job(widget)
  if widget.price_cents > 7_500_00
    FinanceMailer.high_priced_widget(widget).deliver_now
  end

  if widget.manufacturer.created_at.after?(60.days.ago)
    AdminMailer.new_widget_from_new_manufacturer(widget).
      deliver_now
  end
end
```

When thinking about idempotence, I like to go through each line of code and ask myself what would happen if the method got an error on that line and the entire thing started over. I don't worry too much initially how likely that line is to fail or why it might.

For example, if line 2 fails, there's no problem, because nothing has happened but if line 7 fails—depending on how—we could end up sending the emails twice.

Another thing I will do is ask myself what might happen if the code is retried a long time later. For example, suppose line 3 fails and the mail isn't sent to the finance team. Suppose that the widget's price is updated before the failure is retried. If the price is no longer greater than $7,500, the mail will *never* get sent to the finance team!

How we deal with this greatly depends on how serious it is if the code doesn't execute or executes many times. It also can depend on how much control we really have. See the sidebar "Idempotent Credit Card Charging" below for an example where a third party doesn't make it easy to create idempotent code.

Idempotent Credit Card Charging

The code to charge customers at Stitch Fix was originally written to run in the request cycle. It was ported from Python to Ruby by the early development team and left alone until we all realized it was the source of double-charges our customer service team identified.

We moved the code to a background job, but knew it had to be idempotent. Our payment processor didn't provide any guarantees of idempotency, and would often decline a retried charge that had previously succeeded. We implemented idempotency ourselves and it was... pretty complex.

Whenever we made a charge, we'd send an idempotency key along with the metadata. This key represented a single logical charge that we would not want to have happen more than once.

Before making a charge, we would fetch all the charges we'd made to the customer's credit card. If any charge had our idempotency key, we'd know that the charge had previously gone through but our job code had failed before it could update our system. In that case, we'd fetch the charge's data and update our system.

If we *didn't* see that idempotency key, we'd know the charge hadn't gone through and we'd initiate it. Just explaining it was difficult, and the code even more so. And the tests! This was hard to test.

Let's turn our attention to two problems with the code. First is that we might not send the emails at all if the widget is changed between retries. Second is that a failure to send the admin email might cause us to send the finance email again.

You might think we could move the logic into the mailers and have the mailers use background jobs. I don't like having business logic in mailers as we'll discuss in "Mailers" on page **??**, so let's think of another way.

We could use two jobs instead of one. We could have one job do the finance check (and receive the price as an argument instead of the widget) and another do the manufacturer check (receiving the manufacturer creation date instead of the widget or manufacturer).

Let's try that. We'll remove the job we just created in favor of two new jobs: `HighPricedWidgetCheckJob` and `WidgetFromNewManufacturerCheckJob`. We'll remove `PostWidgetCreationJob`:

```
> rm app/jobs/post_widget_creation_job.rb
```

We'll replace our use of that job in `WidgetCreator` with the two new jobs:

```
# app/services/widget_creator.rb

      end
  # XXX
  # XXX
→     HighPricedWidgetCheckJob.perform_async(
→         widget.id, widget.price_cents)
→     WidgetFromNewManufacturerCheckJob.perform_async(
→         widget.id, widget.manufacturer.created_at)
      Result.new(created: widget.valid?, widget: widget)
    end
```

We'll now replace `post_widget_creation` with two methods that these jobs will call.

```
# app/services/widget_creator.rb

          widget.id, widget.manufacturer.created_at)
      Result.new(created: widget.valid?, widget: widget)
    end

× #   def post_widget_creation_job(widget)
× #     if widget.price_cents > 7_500_00
× #       FinanceMailer.high_priced_widget(widget).deliver_now
× #     end
```

```
  # XXX
× #    if widget.manufacturer.created_at.after?(60.days.ago)
× #      AdminMailer.new_widget_from_new_manufacturer(widget).
× #        deliver_now
× #    end
× #  end
  # XXX
× #  class Result
→   def high_priced_widget_check(widget_id, original_price_cents)
→     if original_price_cents > 7_500_00
→       widget = Widget.find(widget_id)
→       FinanceMailer.high_priced_widget(widget).deliver_now
→     end
→   end
→
→   def widget_from_new_manufacturer_check(
→       widget_id, original_manufacturer_created_at)
→     if original_manufacturer_created_at.after?(60.days.ago)
→       widget = Widget.find(widget_id)
→       AdminMailer.new_widget_from_new_manufacturer(widget).
→         deliver_now
→     end
→   end
→   class Result
      attr_reader :widget
      def initialize(created:, widget:)
        @created = created
```

And now, the jobs, starting with `HighPricedWidgetCheckJob`

```
# app/jobs/high_priced_widget_check_job.rb

class HighPricedWidgetCheckJob < ApplicationJob
  def perform(widget_id, original_price_cents)
    WidgetCreator.new.high_priced_widget_check(
        widget_id,
        original_price_cents)
  end
end
```

For `WidgetFromNewManufacturerCheckJob`, we have to deal with several issues we discussed above. Remember that parameters passed to jobs get serialized into JSON and back—at least when using Sidekiq. In our case, we are now passing in a `Date` to the job. JSON has no data type to store a date. That means that although we passed `widget.manufacturer.created_at` to `perform_async`, what will be passed to our job's `perform` method will *not* be a date time. It will be a string.

Because our service layer should not be parsing strings (or hashes or whatever) into real data types, but expect to receive properly typed values, we will convert it in the job itself. Like a controller, the job code is the right place to do these sorts of conversions. Fortunately, `Date.parse` will do the right thing:

```ruby
# app/jobs/widget_from_new_manufacturer_check_job.rb

class WidgetFromNewManufacturerCheckJob < ApplicationJob
  def perform(widget_id, original_manufacturer_created_at)
    WidgetCreator.new.widget_from_new_manufacturer_check(
        widget_id,
        Date.parse(original_manufacturer_created_at))
  end
end
```

Our tests should still pass, *and* give us coverage of the date-parsing we just had to do[6].

```
> bin/rails test test/services/widget_creator_test.rb
Run options: --seed 41159

# Running:

........

Finished in 0.642998s, 12.4417 runs/s, 46.6565 assertions/s.
8 runs, 30 assertions, 0 failures, 0 errors, 0 skips
```

Wow. This is a huge amount of new complexity. What's interesting is that it revealed some domain concepts that we might not have been aware of. If it's important to know the original price of a widget, we could store that explicitly. That would save us some trouble around the finance mailer.

[6]I actually didn't catch this the first time I wrote this chapter. Later parts of the book compare the manufacturer created date to another and, even though it was really a string, the tests all seemed to pass, because I was using < to do the comparison. I changed it to use `before?` after some reader feedback and discovered it was a string. Even after understanding how jobs get queued in detail, and having directly supported a lot of Resque jobs (which do the same JSON-encoding as Sidekiq) for almost eight years, I still got it wrong. Write tests, people.

Similarly, if it's important to know the original manufacturer of a widget, that, too, could be stored explicitly.

Perhaps you don't think that these emails are important enough to warrant this sort of paranoia. Perhaps you can think of some simpler ways to achieve what we achieved here. Perhaps you are right. Still, the point remains that if there *is* some bit of logic that you you need to execute exactly once, making that happen is going to require complexity.

Make no mistake, this is accidental complexity with a carrying cost. You absolutely have to weigh this against the carrying cost of doing it differently. I can tell you that when jobs aren't idempotent, you create a support burden for your team and customers and *this* can have a real cost on team morale. No one wants to be interrupted to deal with support.

This is why design is hard! But it helps to see what it actually looks like to deal with idempotency. I have certainly refactored code to this degree, seen that it was not the right trade-off and reverted it. Don't be afraid to revert it all back to how it was if the end result is going to be less sustainable than the original.

Up Next

We're just about done with our tour of Rails. I want to spend the next chapter touching on the other *boundary* classes that we haven't discussed, such as mailers, rake tasks, and mailboxes.