The Pragmatic Programmers

Rails Scales!

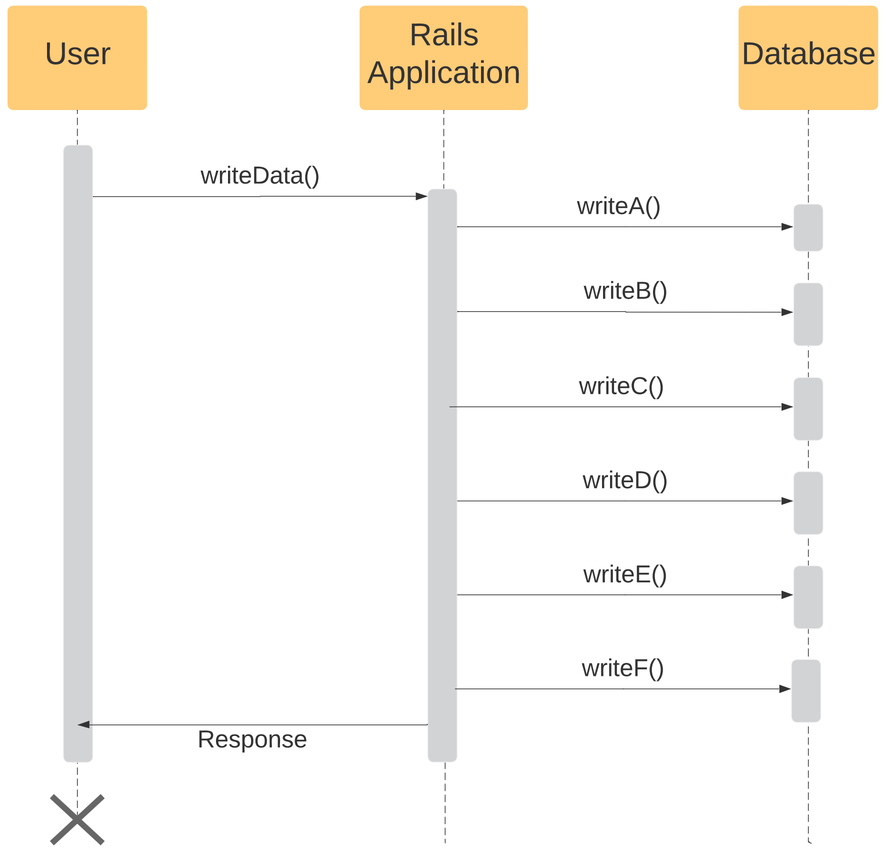
Practical Techniques for Performance and Growth

Cristian Planas

edited by Michael Swaine

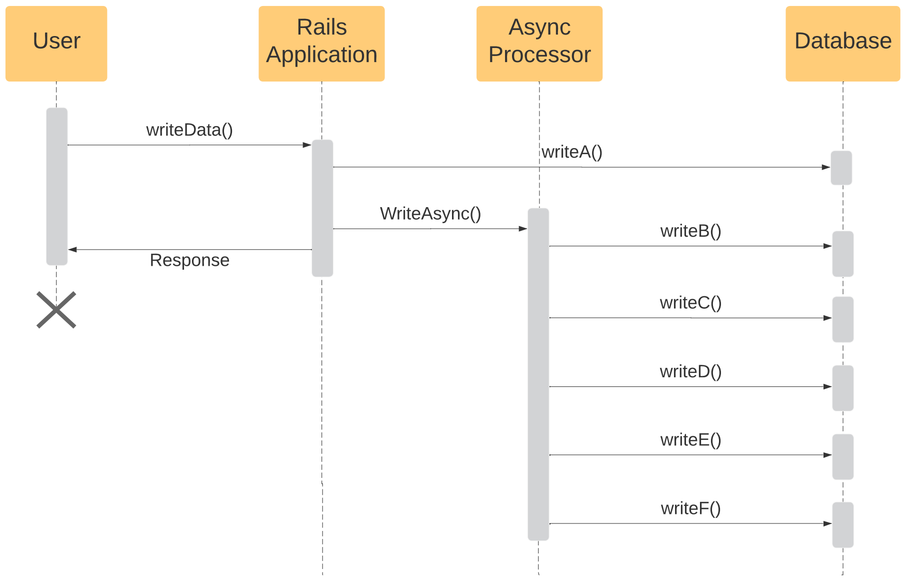
Improving your Response Time with Asynchronous Processing

Feature bloat is a common fate for applications that have been around for a while; this may be even more common for Rails applications, given the speed of development that Rails provides to its users. Even if the product has not added new *features* for the customers, it's probable that the application complexity has increased in other ways. Take the following diagram as an example: years ago you had a simple function (`writeData()`) that performed something simple (`writeA()`). Time passed, and complexity piled up: now the same action does `writeB()`, `writeC()`, `writeD()`, `writeE()`, and `writeF()`. All those actions can be whatever: recalculating statistics, emitting events, generating derived data, sending emails, or anything that makes sense in the domain of the application.



From a product perspective this may be fine and dandy, but from a performance perspective… it may not. Of course, all those new actions could be very important, and they may be also extremely optimized so they are executed as fast as possible. Nevertheless, `writeData()` is way slower than it used to be when it only executed `writeA()`. This can be particularly problematic if it ends up affecting the user experience (think a slow endpoint).

What to do? Well, even if you have decided that everything that this function does is completely necessary, do you really need it to happen synchronously with its primary action? Could you perform `writeB()`, `writeC()`, etcetera asynchronously? Of course, you may decide that the application needs synchronicity. For example, you may decide that emitting a Kafka event needs to be performed in the SQL transaction of a database change, so that change can be canceled if the event creation fails. However, making some actions asynchronous can make your function look like this:



In this section, we will be moving part of the logic of an endpoint to an asynchronous processor.

Before moving on to the implementation, I would like to share with you my experience breaking down actions into multiple asynchronous processes. In my opinion, software engineers start their career with a very idealized view of how an application should run: and this view includes total synchronicity. As one gets experience, the necessity of accepting certain trade-offs becomes apparent. Still, one concern remains: how will users react to the effects of their actions becoming asynchronous? My advice is to be deeply empathic to your users, but also to respect their intelligence. Your users will expect the primary change that they executed to take place immediately (at least to them; “read your own write” can be a solution here), but the effects of their actions can happen asynchronously. Having clear targets (this effect will take place in under a second 99.99% of the time) is important. Having a clear product direction is even more important: designing a system asynchronous from the start is way easier than transitioning a synchronous system to being asynchronous.

Processing Asynchronously with Background Jobs

Rails 4.2 included a new piece of the *Active* ecosystem: ActiveJob.

For engineers who had been using Rails for a while, this action by the Rails core team was a nod to the reality that many, many Rails teams around the world already were using asynchronous jobs. Maybe this is why ActiveJob is agnostic on the precise job runner you use. From the Rails documentation[7]:

> Active Job is a framework for declaring jobs and making them run on a variety of queuing backends. The main point is to ensure that all Rails apps will have a job infrastructure in place. We can then have framework features and other gems built on top of that, without having to worry about API differences between various job runners such as Delayed Job and Resque. Picking your queuing backend becomes more of an operational concern, then. And you'll be able to switch between them without having to rewrite your jobs.

In other words, ActiveModel offers us a common API for background jobs: still, the architecture behind ActiveJob is completely customizable. There are some quite popular gems at your disposal. For example, resque[8], Delayed::Job[9], or bunny[10]. Still, the most used and the one that we are going to employ in this exercise is sidekiq[11].

7. https://guides.rubyonrails.org/active_job_basics.html
8. https://github.com/resque/resque
9. https://github.com/collectiveidea/delayed_job
10. https://github.com/ruby-amqp/bunny
11. https://github.com/sidekiq/sidekiq