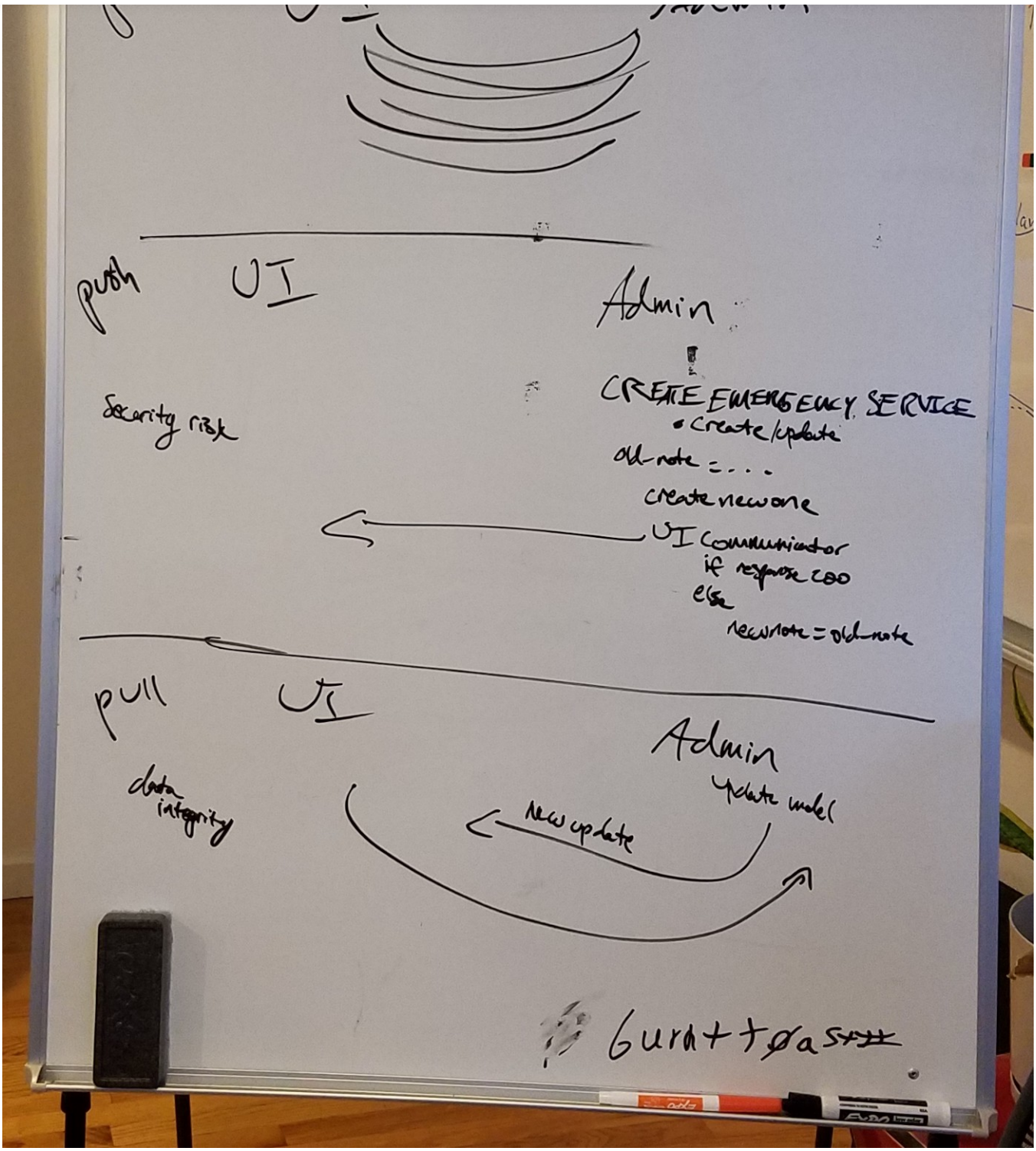
Published in Def Method · Follow

Eric Tillberg · Follow

Nov 3, 2016 · 5 min read

Push, Pull, or Poll

As I approached the end of my apprenticeship with Def Method, my pair and I were met with a deceptively complicated challenge. This challenge is undoubtedly a common one but we struggled to find a good solution because a few presented themselves and we wanted to be mindful of being efficient with our code and not bringing a sledgehammer to a small nail. Ultimately, this problem set me up face-to-face with concurrency, a subject that is introduced relatively early on in the education of a traditional software developer but which I, with a more bootcamp and practitioner-oriented education, had not been exposed to.



This whiteboard shows the three approaches we considered

The problem was as follows: we had two Rails apps, a UI and an Admin app. We had to communicate between these two apps with updates on the Admin side that would inform the text shown to the user on the UI side. The Admin side had its own user interface (which would be used by the administrators of the site). So we proceeded to

the database tables in sync. For the sake of having the whole visual experience load together, we chose to have the UI load the data from its own database instead of making a dynamic request upon loading the page. We didn't want the user to see the page and then have it jolt down to make room for the additional text that was included from the Admin side.

Toward the end of implementing our first idea, where the Admin app would push an update to the UI app, we discovered a potential problem with data integrity in case of a failed push. Specifically, there was the possibility that an administrator updates the Admin app and, for whatever reason, the UI app fails to receive that update to the data. Should this happen, our databases become out of sync. We considered keeping the old data in a variable (in memory on the Admin server) while attempting to push the new data and, in case of failure, we could fall back on the old data. This way our data would remain synced and we could re-attempt a push.

In addition to the data integrity issue, we discovered that there might be a security concern since we had to have our UI app receive PUT requests from our Admin app. How would the UI side verify that the push was coming from the Admin app and not any other place?

The answer to both of these concerns did not come easily so we began fleshing out other options.

The next option we considered was to set up a regularly-scheduled poll of the data. This is perhaps the most hands-off approach but also feels like overkill since the data would likely not be updated *too* often. How often would we need to poll? Every half hour? What if the administrator wanted to see their updates on the UI immediately?

The final approach, and the one we went with, was to have the UI app pull data from the Admin app. This had the obvious first advantage of solving the security issue. Having an app accept data pushes from random sources is a lot more dangerous, but if you're the one pulling, you generally know what you're trying to pull and where from. We thought down this line a little further and deduced that the approach might be to have the Admin app send an alert to the UI app whenever the database was updated.

```
class Admin::EmergencyNotificationsController <
Admin::AdminController

after_action :update_ui, only: [:create, :delete]

...

def update_ui
  UICommunicator.send_update_alert
end
```

Then, the UI app would perform a GET request on the Admin app and pull the updated data.

```
# app/services/admin_communicator.rb

class AdminCommunicator
  def self.get_updated_alert
    HTTParty.get("#{ENV['FAQ_API']}emergency")
  end
end
```

Great! Problem solved, right?

Not so fast. Our final major hurdle to overcome familiarized me with concurrency and multiple processes, words that had heretofore been vague, hand-wavy concepts to me. Sure enough, we had our Admin app tell the UI app that there was an update and wait on a response from the UI app to acknowledge receipt of the alert. In direct response to the alert received from the Admin app, the UI would then ask the Admin app for the new data. But the initial request alerting the UI server from the Admin server hadn't finished! The Admin app was still waiting for a response from the UI app from its initial alert request that there had been an update. But the UI could not respond until it had completed its process which included another request back to the Admin app. That second request, from UI to Admin, could not be completed since the Admin app was hung on waiting for its response to its initial request. A sort of catch-22, the Admin app couldn't respond until it had received its response from the UI but the UI couldn't

seem to work) and thinking about other ways around this problem (background processing), we discovered that Ruby has the built-in capability to create a new thread in your code. All we had to do was execute the initial alert from the Admin app to the UI app in its own thread. Then, we could have the UI app do its thing in peace and pull the data properly without hanging.

So the first code sample above turns into this:

```
# app/controllers/admin/emergency_notifications_controller.rb

class Admin::EmergencyNotificationsController <
Admin::AdminController

def update_ui
  Thread.new do
    UICommunicator.send_update_alert
  end
end
```

The moral of the story? Well, its multifold, but I would summarize it to say that honing the ability to explore multiple approaches pays off handsomely. We didn't give each approach even treatment and, as often as not, made "gut" decisions that may have been questionable, but we made an effort to be somewhat systematic and open-minded. This led us to a pretty elegant solution (i.e., it required very little new code) and gave us deeper understanding of the problem than if we had just thrown Resque at it and called it a day. Also, the more you know about your language (like the fact that `Thread` is a thing in Ruby), the better off you'll be. At the end of the day, we found a solution to the problem but, more to the point, we learned a lot along the way. This is an all-too-rare example of the slow, tortuous path to mastery.