# Adding a Rake Task to Remove Unused Subscriber Lists

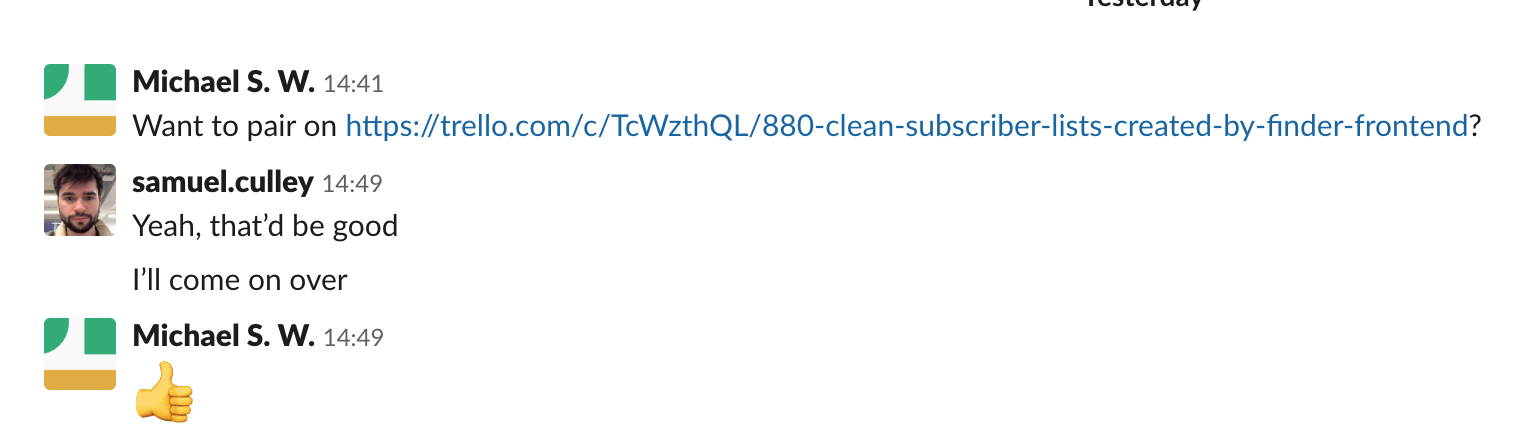
In this piece of evidence I will refer to standard points, but also include the corresponding number for that standard point in square brackets.

The language that I used during this ticket was Ruby, and there were also some command line functions.

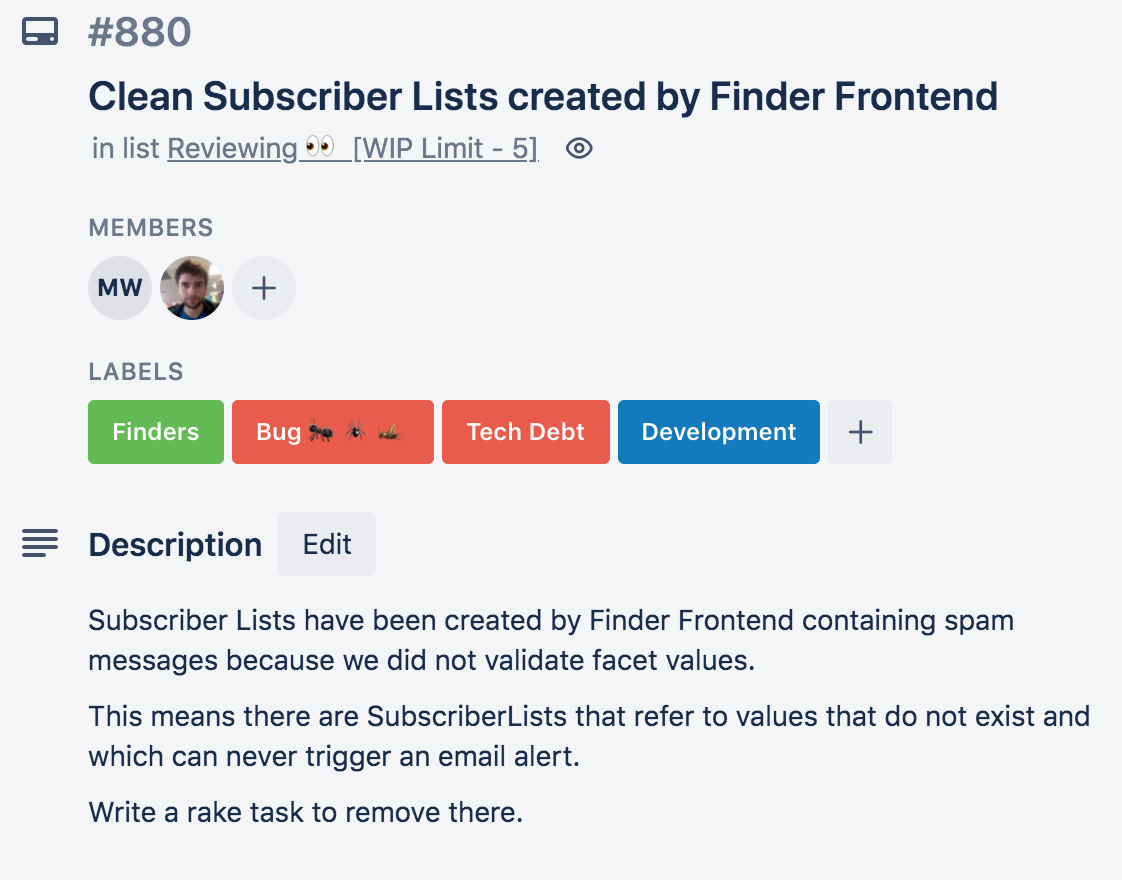
Part of our team’s work involves monitoring the email alert system that we use for our search tools. This functionality lets users subscribe to a particular search query, and they will receive updates by email whenever a new item is published that matches their search. For example, someone might want to find out any news about a particular court case, so they would subscribe to a court judgements email alert, which would trigger if something that matched that was published.

Because we have to maintain this system, we also need ways to ensure that it’s not running too slowly, missing out on alerts, or misleading anyone when they subscribe.

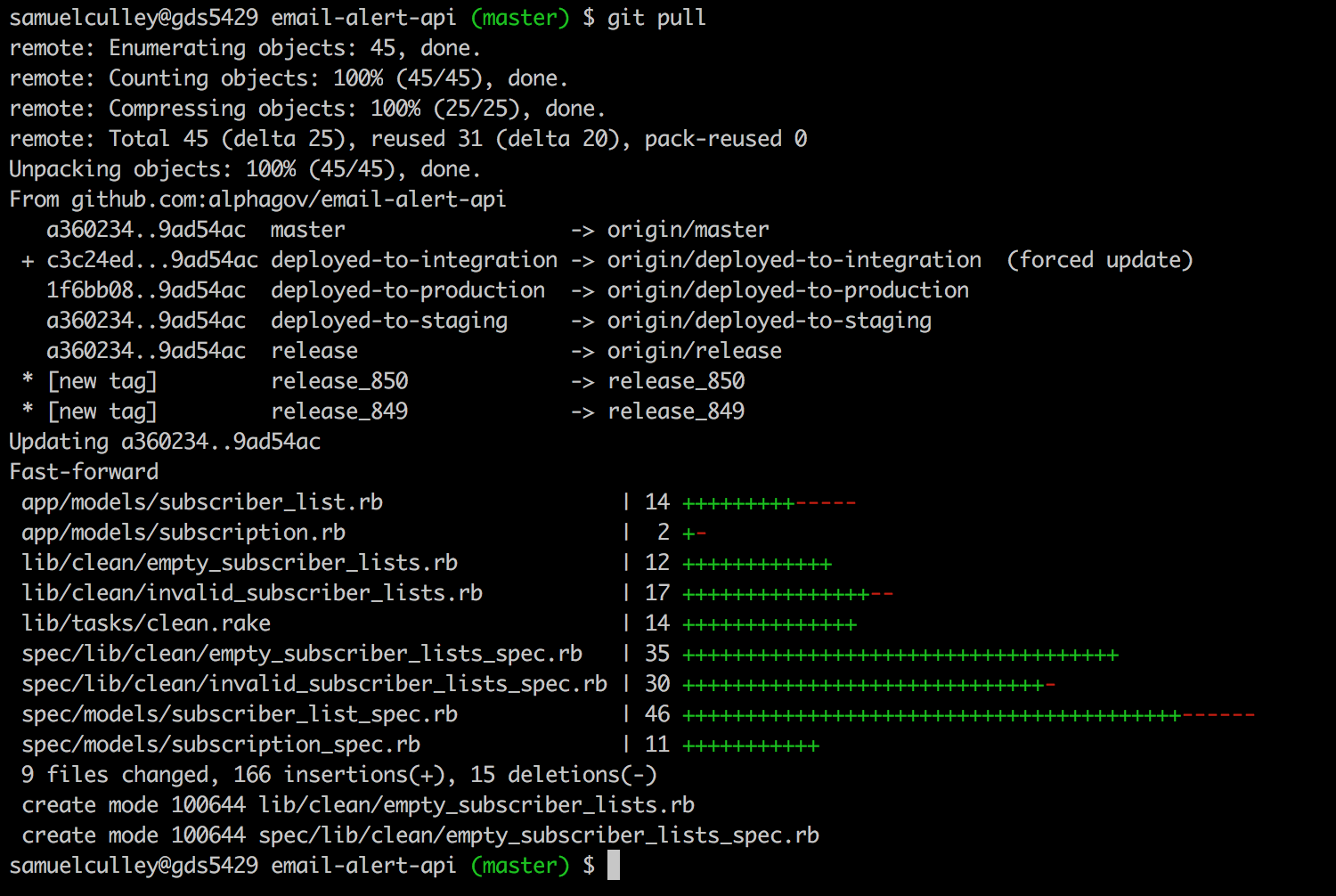
This ticket initially started with me asking during our morning standup if anyone had any work that I’d be able to pair on today, as I had finished my previous ticket. I ended up receiving a pairing request from one of the developers on my team, which I accepted.



I then added myself to the Trello ticket, and read through the description to learn a bit more about the context for the request. This required me to analyse what the issue was, and what needed to happen. I understood that this was a bug fix, and related to the ‘Email Alert API’ part of the application I was working on. This meant I now knew which Github repository it was that I needed to look at. It had currently been picked up by a team member, but hadn’t had any work done on it.

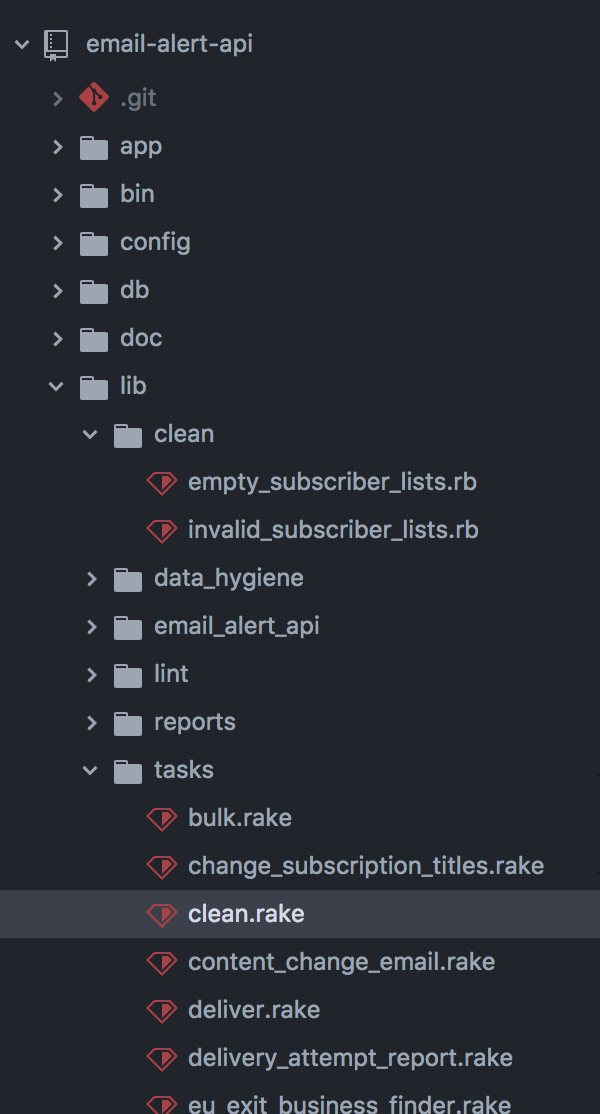


Because no work had been done on this ticket, I was able to pull the repository from its master branch, using the command line. The screenshot below shows me using the ‘git pull’ command, and updating my local version of the repository.

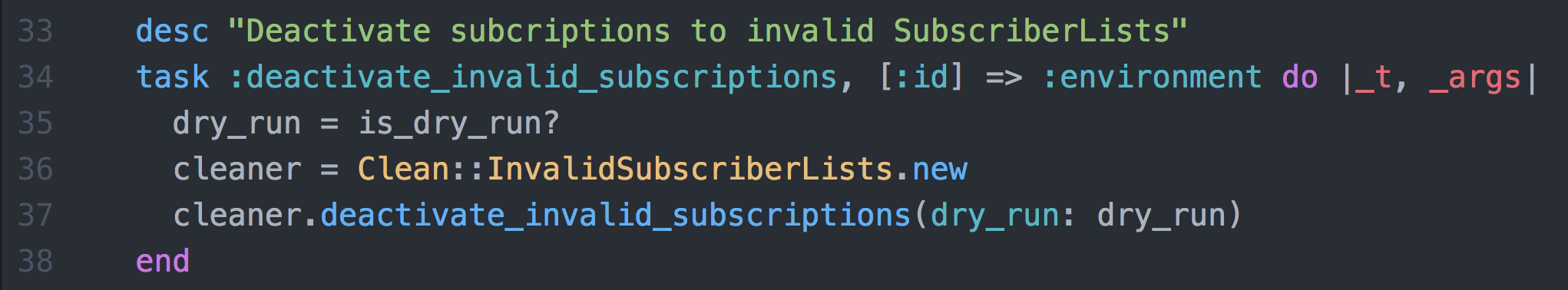


I then made a new branch so that my work could be tracked independently. I decided to name this ‘remove-unsubbed-subscriptions’ to keep it fairly concise, but easily understood in the context of the work. I agreed this with the developer I was pairing with, as I knew that this would be seen by a wider range of people.

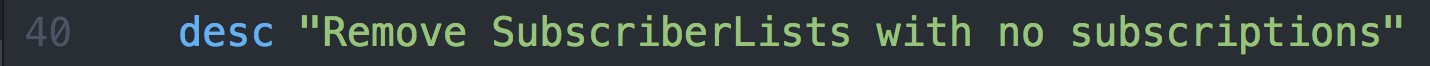
I then opened up the repository in my text editor. After speaking with my pair, I navigated to the appropriate file that needed to be changed. Knowing that this change required a new Rake task to be made, I learned that this meant that within the ‘lib’ folder, there would be a list of pre-existing ‘tasks.’ Using the heading of the ticket, I recognised that the pre-existing ‘clean’ task would be the appropriate place to put a new sub-task, rather than creating a new keyword task.



I checked the contents of this file, and found another task that was similar to the one that I wanted to implement. I decided to use this as a framework for the new task that I wanted to create. This meant that I needed to delve a bit deeper to understand what this task did.



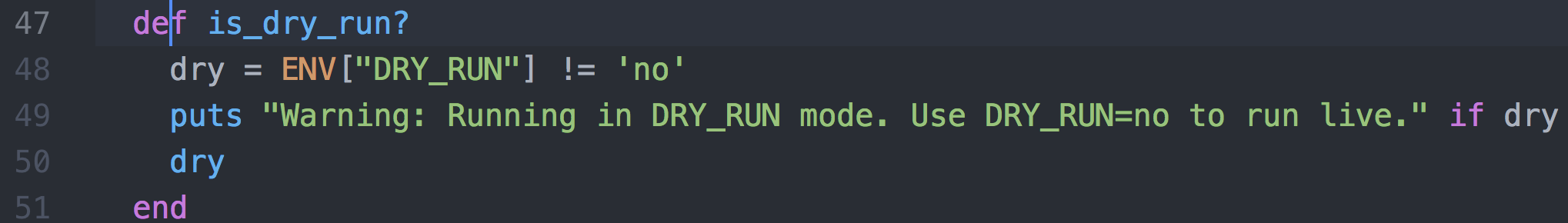
First, I took apart the top two lines. The first of these was a description line, that described the function of the task. This was easy to redo for my new task, so I created a new entry in the file.



I then looked at line 34, which was the name of the task itself. I knew that this would be what was entered on the command line when running the task, so I wanted to make it an imperative name. After discussing with my pair partner, I called it ‘remove\_empty\_subscriberlists’. I also included the required syntax at the end, which would enable code beneath this to be executed when the task is called.

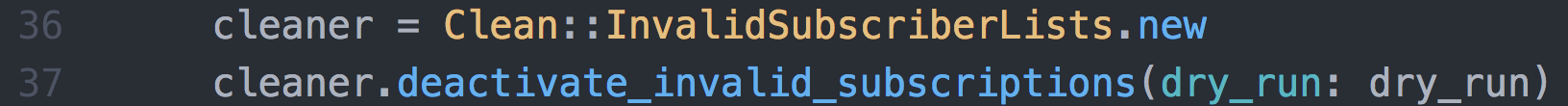


I then look at line 35 - this line set up a variable that would contain a boolean value - but I needed to look at the ‘is\_dry\_run?’ function to see exactly how this would work. I used the search function in my text editor to find it, and it was contained at the bottom of the file. I understood that the context for this meant that if a task was called as a ‘dry\_run’, it shouldn’t actually complete, but should still go through sufficient processing to make sure it could run. The function itself checks an environment variable to see whether it’s a dry run or not, and the output of the function will be true or false depending on this. This meant that the task I was creating would need to have a ‘dry\_run’ mode as well as its standard mode.

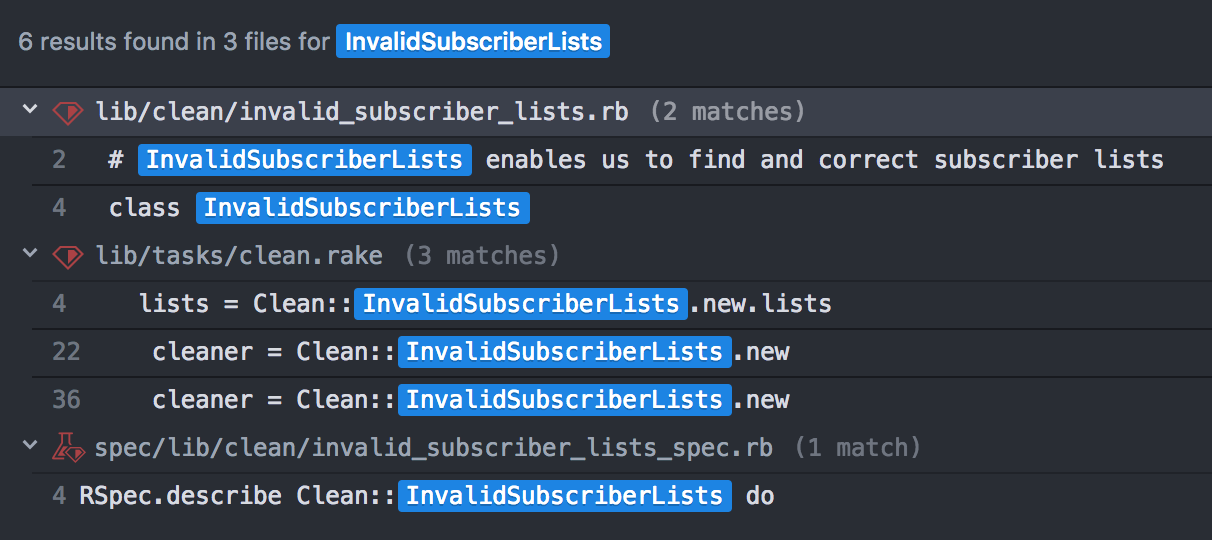


This also then meant that I could simply include the same code as the pre-existing task, as it would make sense to use the same dry run environment variable check. I knew that reusing code was the most effective way to do this.

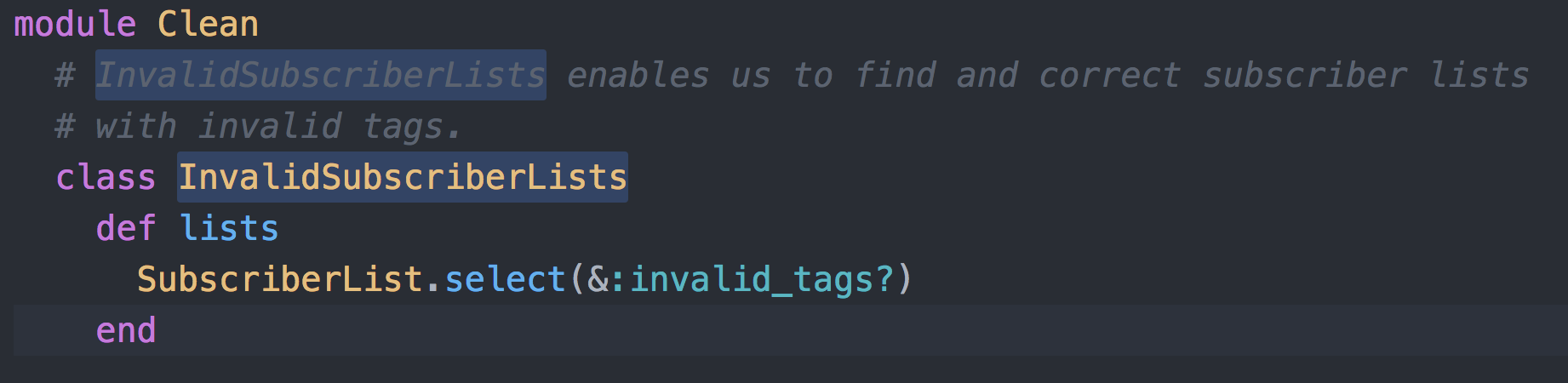
I then looked at lines 36 and 37. These showed a new class object being created and assigned to the ‘cleaner’ variable. I didn’t initially know what this new object was, but I recognised the key phrases I’d need to find were ‘Clean’ and ‘InvalidSubscriberLists’. On line 37, the new object has a method called on it, which took a ‘dry\_run’ parameter that was set up earlier in the task. This suggested that the dry run status was being used by this method to affect what happens when it runs.



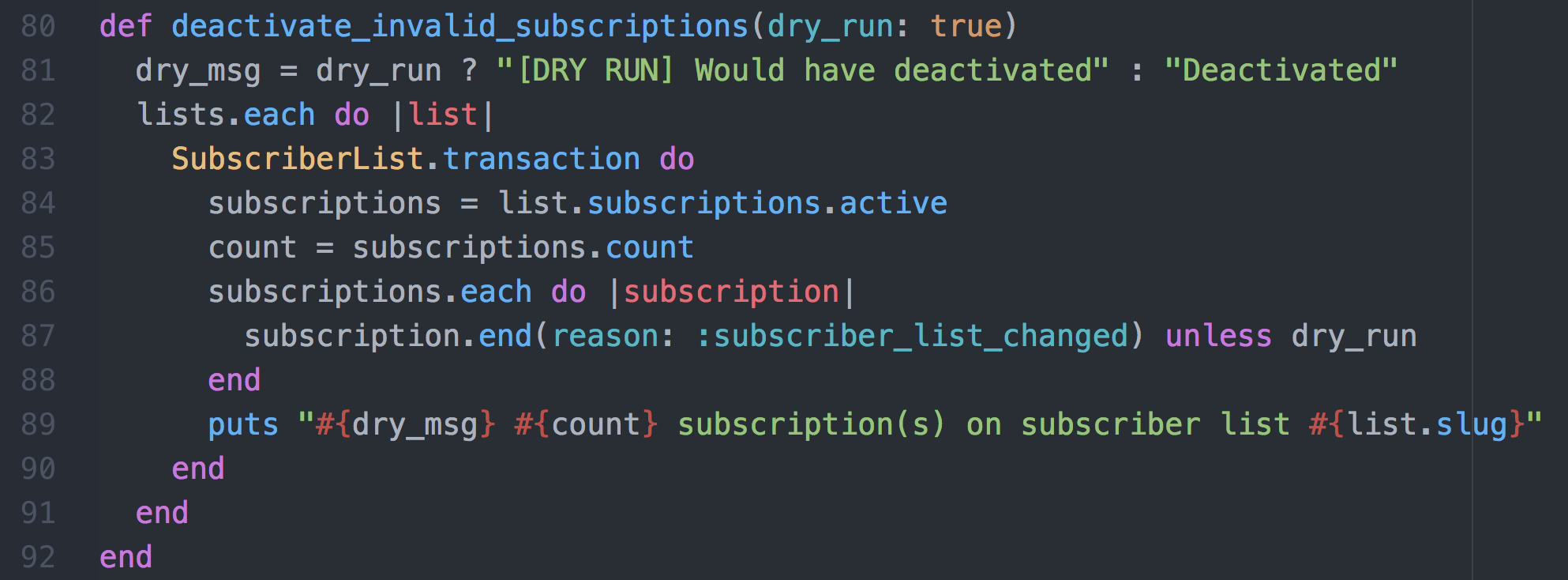
Using my project search tool, I put in the ‘InvalidSubscriberLists’ to try and find the areas of the project where this term is used. I found a range of results, but I noticed that on line 4 of the first file there was a class definition for ‘InvalidSubscriberLists’. This seemed like a good place to go to see how this class worked. I also started thinking about how I would construct my own class to handle the task that I wanted to create, for removing unsubscribed lists.



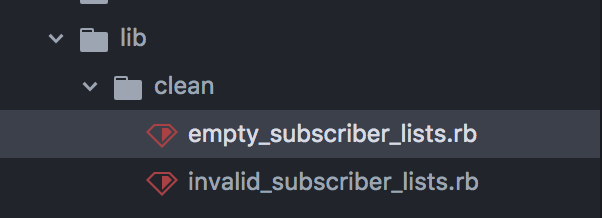
Opening this file up, I started looking at how it was structured, and additionally how the ‘dry\_run’ parameter was used. I noticed that this was all encapsulated in a ‘Clean’ module, which explained the usage earlier (Clean::InvalidSubscriberLists). I also recognised the ‘select’ method being used on the ‘SubscriberList’ object, which would be pulling out all the results from the Subscriber Lists database that would return true when the ‘invalid\_tags?’ method was called on them. I knew that I’d need something similar for me to find all the subscriber lists that didn’t have any subscribers, so this syntax would be useful later.



The rest of the code in this class was specifically for removing invalid lists, which didn’t interest me too much. The only remaining aspect that I was interested in was seeing how the ‘dry\_run’ parameter affected the output. Looking at the method that was being called in the task, I saw how the parameter ‘dry\_run’ was set as ‘true’ by default. I also noticed that on line 81 the dry\_run value is used to set up different strings in the ‘dry\_msg’ variable. On line 87 subscriptions are ended whenever it is not a dry run. Finally, on line 89 the message is printed out to the console with the ‘dry\_msg’ included.

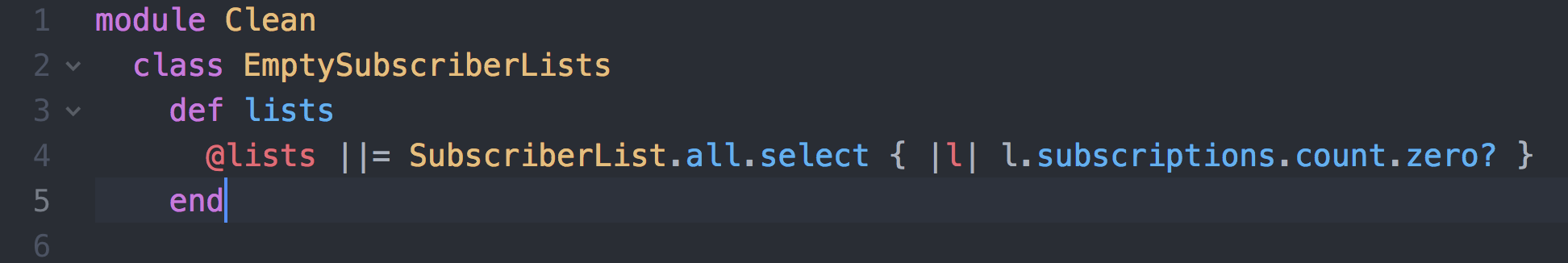


It was now time to create my own class to run the task I was setting up. I decided to locate it in the same directory as the InvalidSubscriberLists class, but use the same module, ‘Clean’. I created a new file called ‘empty\_subscriber\_lists’, which would contain a class of the same name.

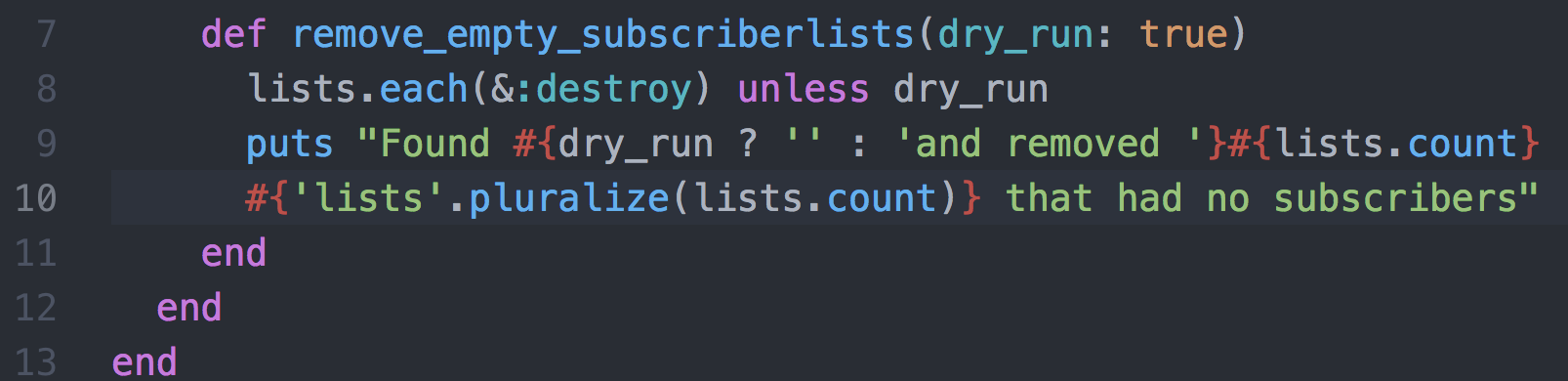


I then set up the module ‘Clean’ and the class ‘EmptySubscriberLists’. I wanted to have two methods in this class, one that would create a set of empty subscriber lists, and another that would remove them. I also wanted to make sure that a dry\_run variable could be ingested by my removing method, so that I could vary whether or not it would actually remove them when it was run.

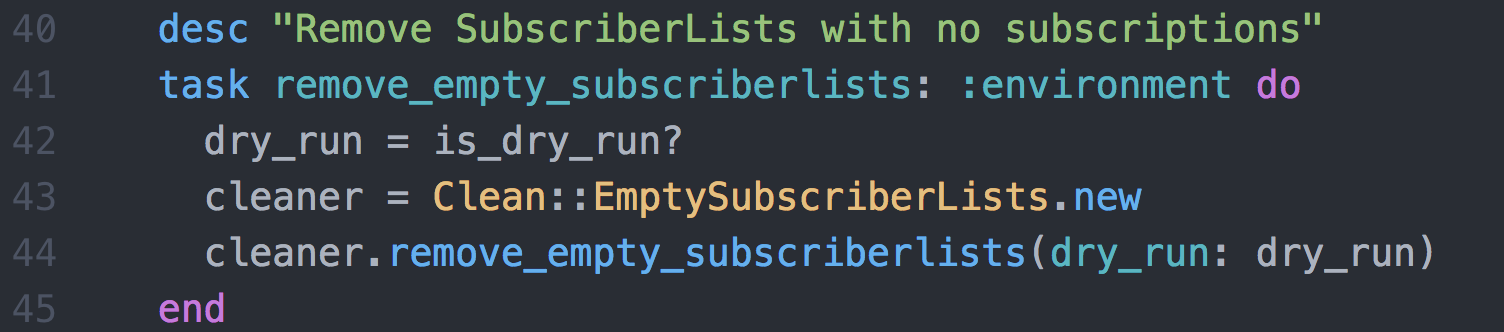
The code below shows the method that I made within the class that would create a list of subscriber lists that had no subscriptions. The method first tries to use the instance variable ‘@lists’, but if this is not available for this instance of the class, it then sets a new instance up. This uses the SubscriberList object, and selects all of the entries that have a subscription count of zero. This will end up as a data structure that contains all of the subscription lists that have no subscriptions, which is still tied to the database that it was drawn from.



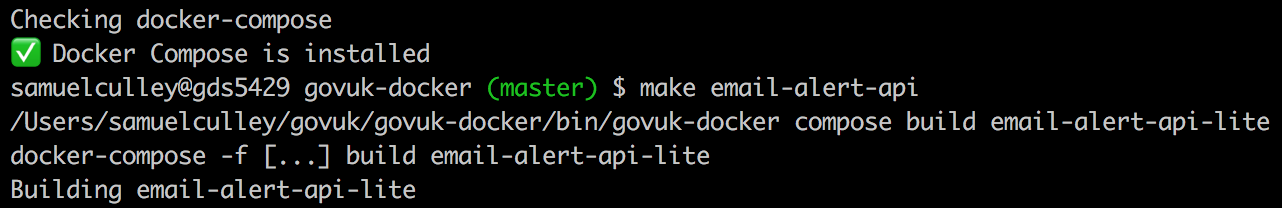
The code below shows the method that I made that would remove all of the unsubscribed lists. I gave the method a name that described what it would do, and made sure it had a dry\_run parameter that defaulted to true. I then iterated through each item in the @lists variable by using the ‘lists’ method, and destroyed them unless it was a dry run. Lastly, I printed out a statement that would accurately describe either how many lists were found and removed, or if it was a dry run, just how many were found.



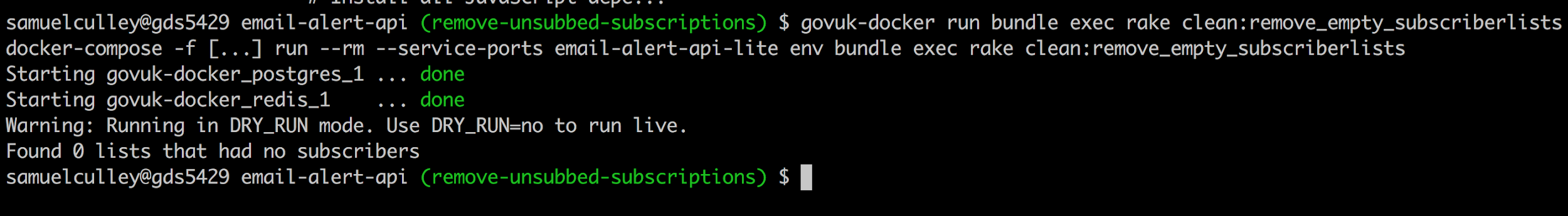
With this class completed, I then return to the rake task file to finish setting up the task, using the newly built class. I used a similar format to the previous task, by assigning a ‘dry\_run’ variable, creating a ‘cleaner’ object, ‘and calling the remove\_empty\_subscriberlists’ method on it, passing in the ‘dry\_run’ var as a parameter.



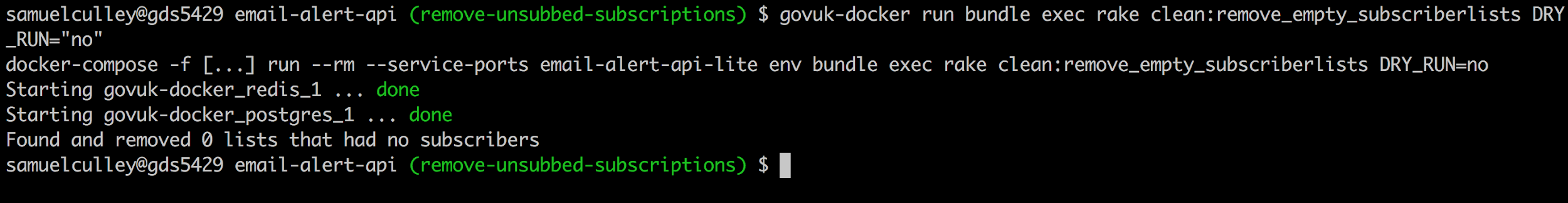
Now that I’d written the code for the task, I wanted to run it to see how it performed. All of this work was being done in a development environment, which meant that there wasn’t any real data at stake. However, this also meant that there wasn’t any existing data that could be manipulated, making it hard to see whether the task was having any effect. I first ran the task locally to see if there were any immediate issues. This required me to set up a docker container for this application, which I did using my organisation’s docker tools. The third line of the screenshot below shows my running a command that will create a docker file for this application, as well as covering any dependencies that would be needed.



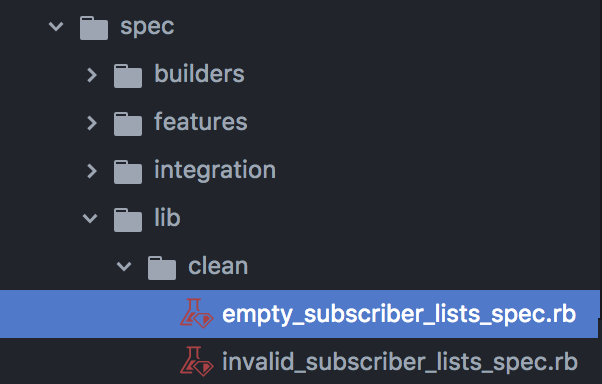
To then run the task in the docker container, I used the command on the first line below. This started up the docker container, and executed the rake task. The results show that the task was run in ‘DRY\_RUN mode’, which should mean that the task should only print out how many lists had no subscribers. Because I was in my own sandbox environment though, there were no lists anyway.



Additionally, when I set the dry run environment variable, the message that was printed showed that it also tried to remove lists as well. Overall this told me that my code was at least in some form of working state, although I still couldn’t tell if it would actually remove subscriber lists if they did exist.



Before I moved this to our integration environment, I first wanted to write an rspec test that would cover this task. I looked at the other tests that had been created already for the other similar task, and found that there was one set up in the spec/lib/clean directory. I then created another file that would contain the tests for my new class.

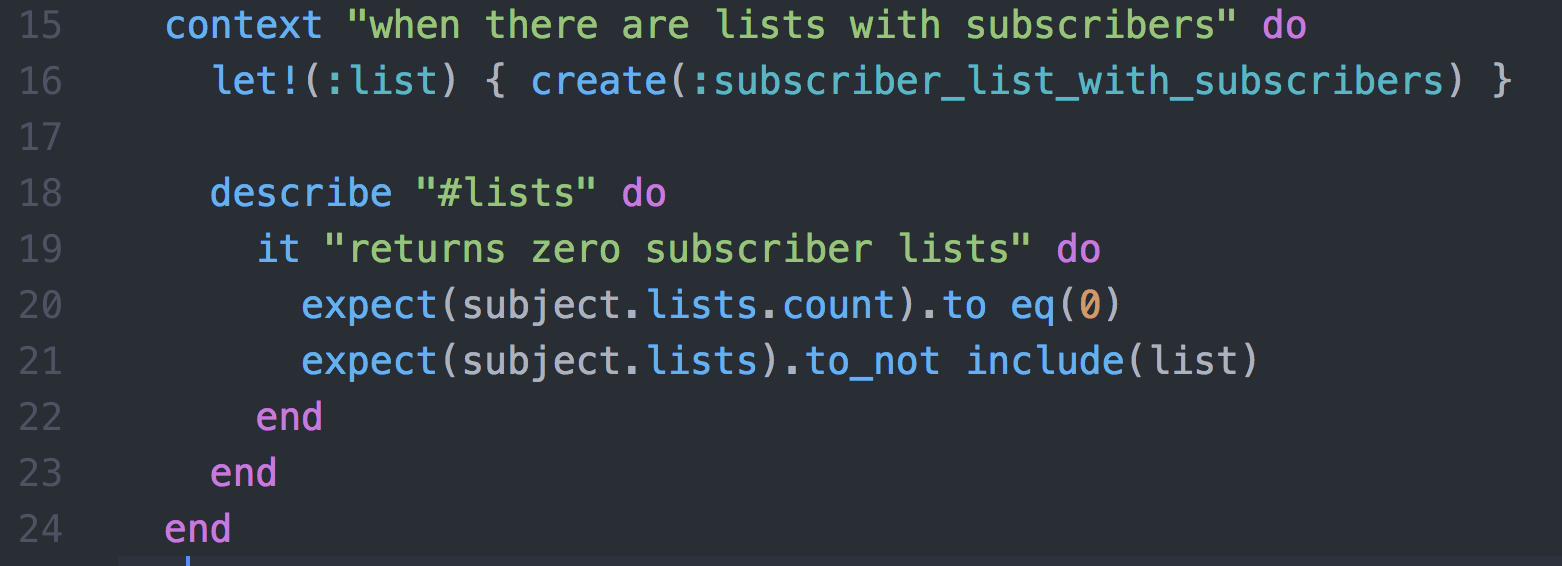


I copied the setup from the other test, so that I could ensure it would be run as part of the test suite. This ended up only really being lines 1 and 2, where the test is linked to the ‘EmptySubscriberLists’ class, and a new instance is made under the name ‘subject’.

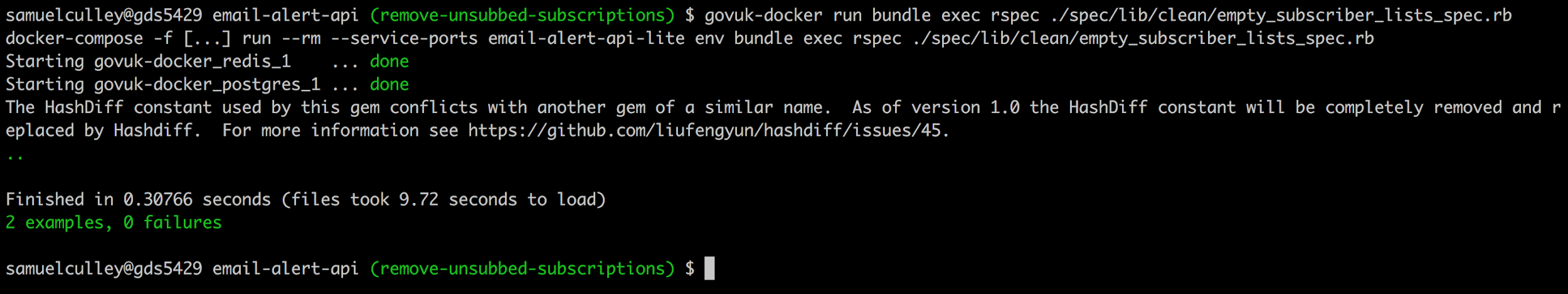
I was able to reuse some code from the other test set, and in particular made use of some of the ‘factories’ that had already been made for this type of content. This included creating new ‘subscriber\_list’, which would behave like an entry in the database. This meant I could simply set up a new ‘list’ item, and then create a test that ran the ‘.lists’ method on my class. The new subscriber list would have no subscriptions, so it should correctly identify this, and return the test list.



Additionally, I wanted to do the reverse, where a subscriber list with subscriptions was created, and my class wouldn’t include them in its ‘.lists’ method.



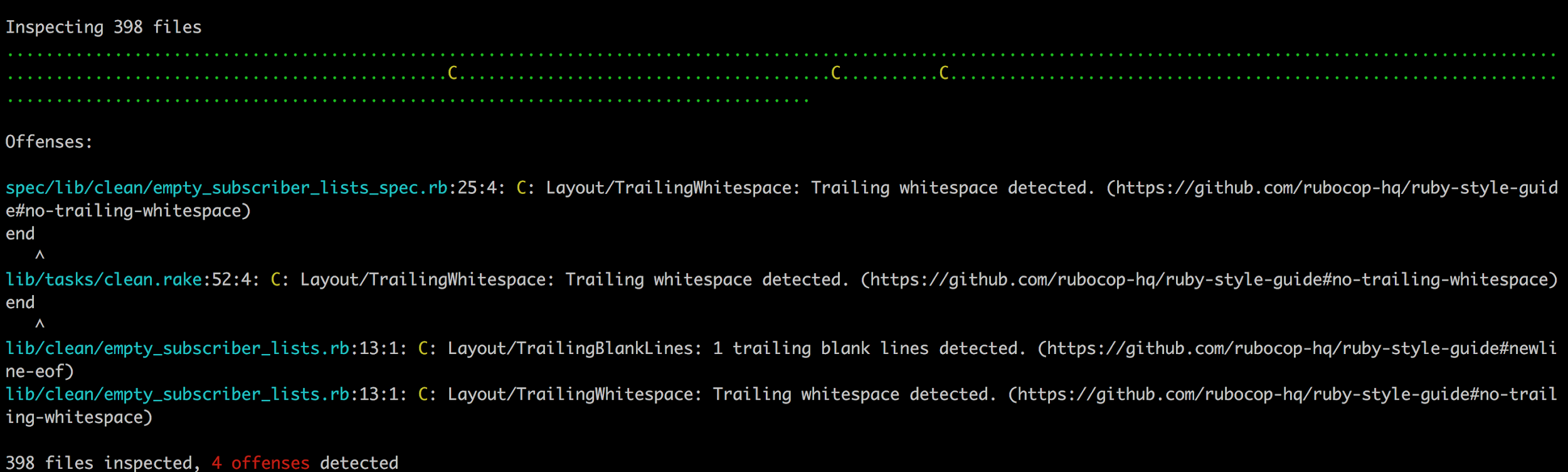
I then ran the tests from the command line, specifying the exact file. The first line contains the full command, where I run the test via the docker container, and specify the directory in which my test is located. This returned 2 successful tests, and no failures.



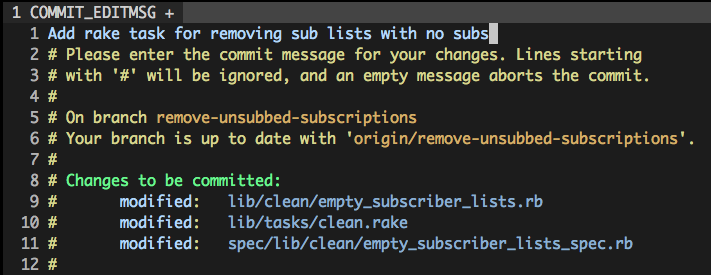
To move this to the integration environment, I had to make a pull request to the repository. Before doing this, I wanted to make sure that all the pre-existing tests were still passing, and that any other cleanup was completed. I ran the following command to see if this was the case:



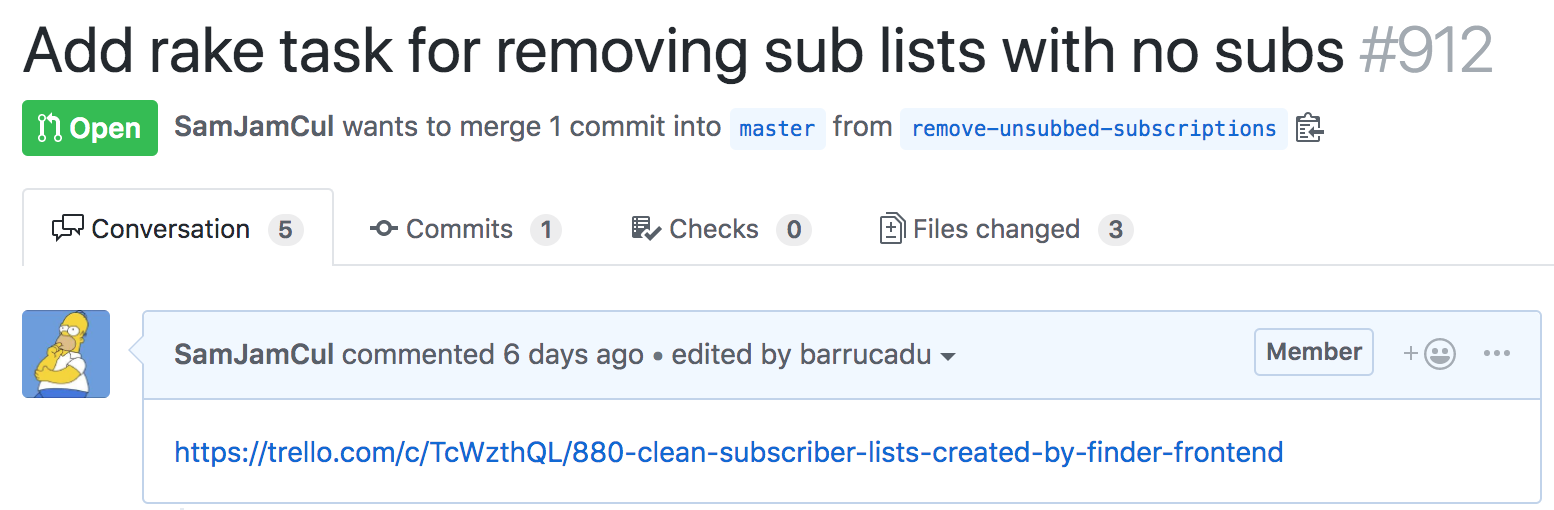
All of the automated tests passed, but the linter picked up some issues with some of the code I’d written. This was fairly easy to fix, so I went through and removed the offending spaces and line breaks.

****

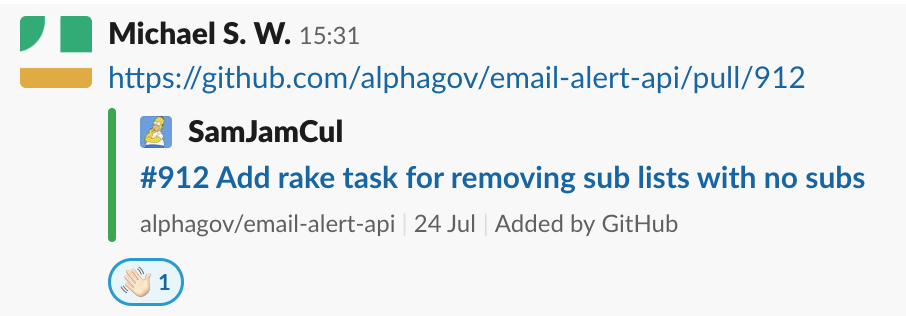
Once this was done, I made a commit of all the work that I’d done. I titled it using my organisations suggested imperative phrasing, and kept it under the maximum length so that it wouldn’t be truncated when being shown in github.



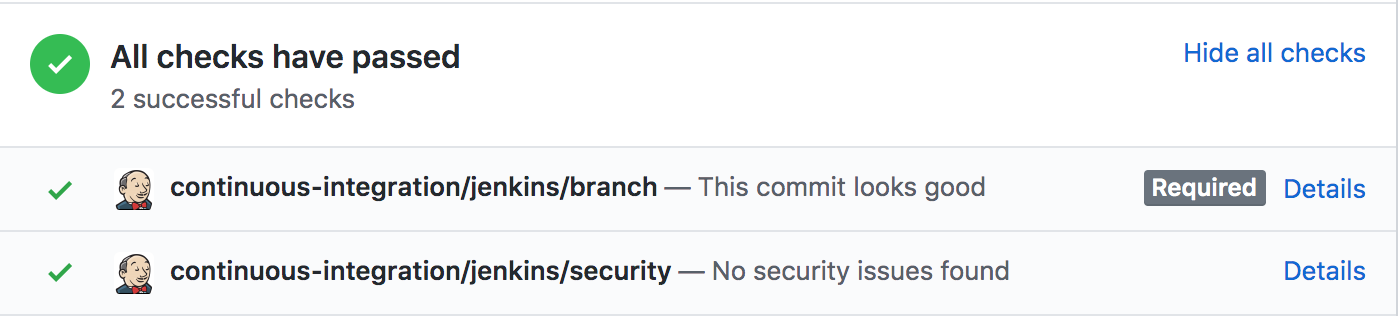
With the commit in place, I pushed this up to the remote repository, creating a corresponding branch and pull request in the process. I used the ‘git push --set-upstream origin’ command to do this. I then opened a pull request, linking to the Trello card that contained the original request. I also moved the trello card into the corresponding reviewing column.



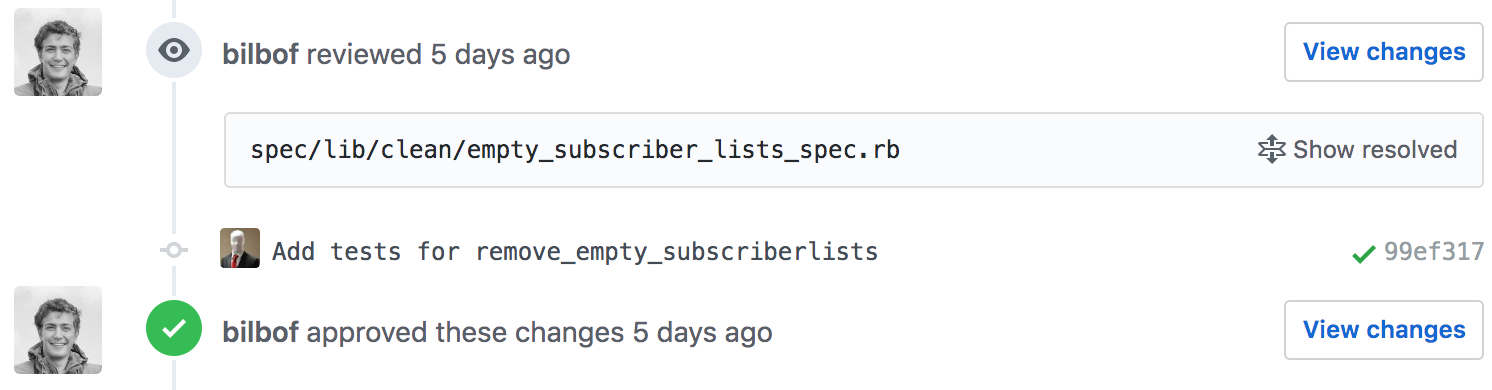
In order for the branch to be merged into master, I needed to get two things - an approving review from another developer, and to ensure that the automated tests were passing. Me and my pair partner made a request on our team slack channel for reviews.



Whenever a branch is pushed to the remote repository, there is a set of Jenkins tasks that run automatically which ensure that the branch is safe to merge. This includes running the rspec suite of tests, but also things like the linter to make sure the styling and formatting quality is maintained. In this case, both passed without any issues.



Once I’d obtained an approving review, I was able to merge the changes into the master branch. From this point onwards, the deployment was handled by automated processes and went through some additional checks to make sure it would be safe for live.



## Summary

The key challenge for this work was understanding the code, and implementing the task in a way that matched the pre-existing tasks. This required me to interpret and understand a decent chunk of the codebase, and navigate through several areas to work out how things fit together. Although I think this went quite well, it would definitely be more difficult if things like function and class names weren’t as clear. This made me appreciate working on a codebase that has quite clear style guidelines and tests that keep things organised.

I’ve also learned about the testing concept of a ‘factory’ which can be used to repeatedly generate more complicated structures, especially when a double wouldn’t really be sufficient. I think this has been one of the key things I’ve learned, as it made my work to generate a test much easier, and it meant that I didn’t need to work out how to substitute test data that exactly matched the criteria, as it had already been done.

The ticket raised was closed, and the subscriber lists that had no subscribers have been cleaned up using this task. There were almost 4000 of them, so it will make regular sweeps of the subscription database run a bit faster!