Open Source Chronicles: Adventures in the Zulip Realm

An edutainment report by Ben Reeves (bgreeves) and Alyssa Wagenmaker (acwagen)

Abstract

Now this is a story all about how our work plan got flipped turned upside down, and we'd like to take a minute, just sit right there, we'll tell you how we became contributors to a project called Zulip.

In this report, we will document our adventure in open source software engineering, telling the tales of incidental bug discovery, tragic development environment issues, and heartwarming collaboration between developers from across the country, and starring some of your favorite software engineering stars: Test Coverage, Requirements Elicitation, Perverse Incentives, Risk, and Effort Estimation.

About Zulip

Zulip is an open-source chat service similar to Slack or IRC, but not a substitute. It is unique in the fact that it subdivides streams (what one would equate with a Slack channel) into topics for more fine-grained chatting. It has a Python (Django) backend with a JavaScript web frontend, as well as a React Native mobile app and a desktop client written in Electron. It is suitable for any scale: from personal use to large companies (they boast that it scales to thousands of users).

GitHub: https://github.com/zulip/zulip
Website: https://zulip.org/

Project Context

Zulip is an open source alternative to group chat applications like Slack. It was originally founded as a startup in 2012, but was acquired by Dropbox soon after. The Zulip product is marketed towards businesses and other large organizations, and allows individuals to be able to work together from anywhere, at any time. The conversation threading model allows one to easily catch up on missed messages and continue a conversation hours later, "asynchronous discussion." Zulip is a relatively accessible product to anyone who would want to use it — a free version of Zulip is offered with limitations on storage and search history, while a full version is offered at $8/month per active user. Many discounts are offered to organizations like open source projects and educational institutions.

Project Governance

The primary method of communication used in the Zulip project is the Zulip chat server, along with GitHub issues and pull requests when applicable. The communication process itself is quite informal — the developers are friendly and questions or concerns at any point are welcomed. There are many pre-existing resources to help ease the onboarding process, outlining contribution guidelines and basics about the project architecture, as well as setup tutorials and troubleshooting help.

The acceptance process for any change or addition to the project is fairly straightforward: submit a pull request, after which it will be run through three different CI builds and reviewed by at least one core developer. If all goes well the patch will be accepted and merged.

The standards applied to all contributions to the Zulip project encompass commit messages, code style, and code itself. Commit messages must be informative and structured in specific ways. When making a commit, a Zulip linter is run on both the commit message and the code itself, checking several different readability and style metrics. Additionally, measures are put in place to prevent commit history from being clogged by extraneous commits. Git pulling and merging are not used, instead one must rebase the local repository before committing. Commits are encouraged to be amended and squashed if they do similar things. In terms of correctness, all CI tests must pass at every commit added to the Zulip repository, and code itself must resolve the issue in question in an acceptable way. This is verified through CI builds and code review.

Completed Tasks

Task 1

Description: Add a feature to the backend's automated testing script to report files that have reached 100% code coverage.

Implementation: A similar feature existed in the frontend's automated testing script, so we identified the relevant code and adapted it to work for Python code instead of JavaScript. It involved looping over all the previously not fully covered files and checking the line coverage for each one (using the coverage Python library), printing a message to the user if coverage is now up to 100%.

Pull Request: https://github.com/zulip/zulip/pull/9009

Accepted: Yes

Task 2

Description: Add a test case to bring zerver/lib/soft_deactivation.py up to 100% code coverage.

Implementation: We wrote a unit test for an edge case that covered the missing line of code. The test case is similar in implementation to other tests in the pre-existing Zulip test suite, but tests the relevant edge case: specifically, when a user unsubscribes from a stream before becoming long term idle. The general flow of the test case performs certain set-up actions, asserts correctness of preconditions, calls the function in question (add_missing_messages) and asserts the correctness of the postconditions.

Pull Request: https://github.com/zulip/zulip/pull/9055

Accepted: Yes

Task 3

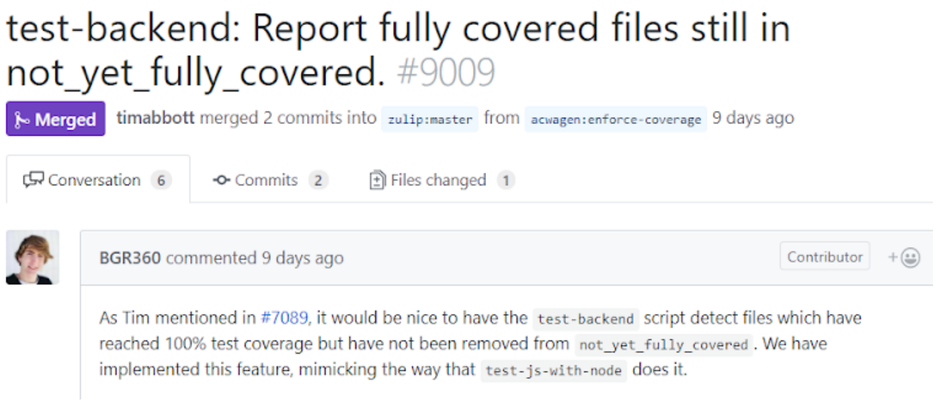
Description: Fix a bug in zerver/lib/soft_deactivation.py that sometimes results in incorrect messages being given to a user.

Implementation: Changed a < to <= in a conditional.

Pull Request: https://github.com/zulip/zulip/pull/9055

Accepted: Yes

Proof for Task 1

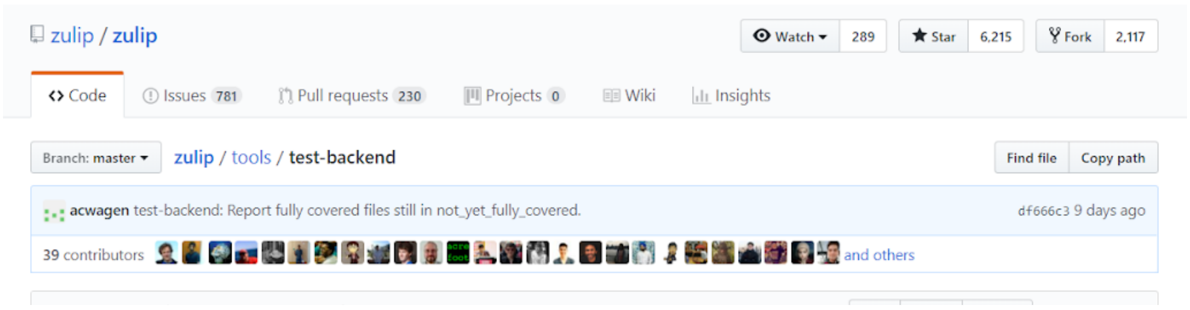

test-backend: Report fully covered files still in not_yet_fully_covered. #9009

Merged timabbott merged 2 commits into `zulip:master` from `acwagen:enforce-coverage` 9 days ago

BGR360 commented 9 days ago

As Tim mentioned in #7089, it would be nice to have the `test-backend` script detect files which have reached 100% test coverage but have not been removed from `not_yet_fully_covered`. We have implemented this feature, mimicking the way that `test-js-with-node` does it.

Our merged pull request for Task 1.


zulip / zulip

Watch 289 | Star 6,215 | Fork 2,117

Code | Issues 781 | Pull requests 230 | Projects 0 | Wiki | Insights

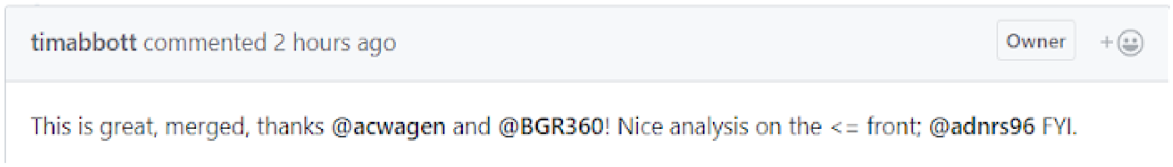
Branch: master | zulip / tools / test-backend | Find file | Copy path

acwagen test-backend: Report fully covered files still in not_yet_fully_covered. df666c3 9 days ago

39 contributors and others

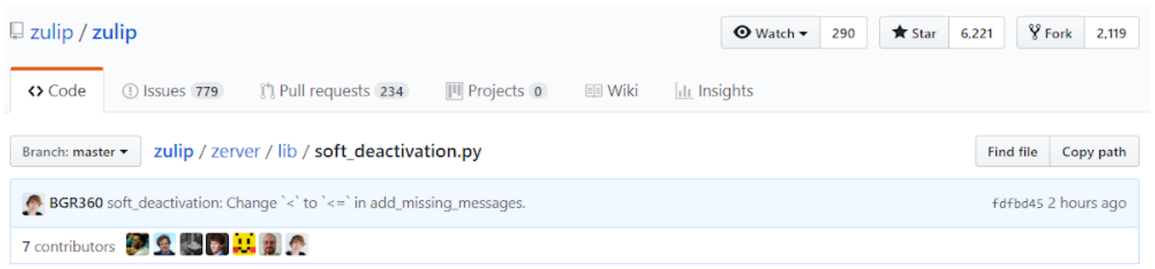
Our commit in the master branch of the Zulip repository.

Proof for Tasks 2 and 3


timabbott commented 2 hours ago | Owner

This is great, merged, thanks @acwagen and @BGR360! Nice analysis on the <= front; @adnrs96 FYI.

Evidence of merge for our pull request for Tasks 2 and 3.


zulip / zulip

Watch 290 | Star 6,221 | Fork 2,119

Code | Issues 779 | Pull requests 234 | Projects 0 | Wiki | Insights

Branch: master | zulip / zerver / lib / soft_deactivation.py | Find file | Copy path

BGR360 soft_deactivation: Change `<` to `<=` in add_missing_messages. fdfbd45 2 hours ago

7 contributors

Our commit in the master branch of the Zulip repository.

Quality Assurance

QA activities

Communicating with the developers

- When we had questions about anything relating to what we should work on or concerns about correctness, we communicated with other developers to clarify our understanding and make sure we were on the right track.

Code review

- Every pull request we made was reviewed by a core developer, who would give us suggestions on improvements or changes to make.

Reading existing code

- We read existing test cases and code, noting existing coding conventions and test case design. We modeled much of our implementation after this, to ensure that our additions would integrate into the project well.

Linters

- We used the provided Zulip linters to check the style of our commit messages and code.

CI tests

- We ran the existing test suite locally to ensure we had not caused anything to regress, as well as passing the enforced CI builds upon submission of our pull requests.

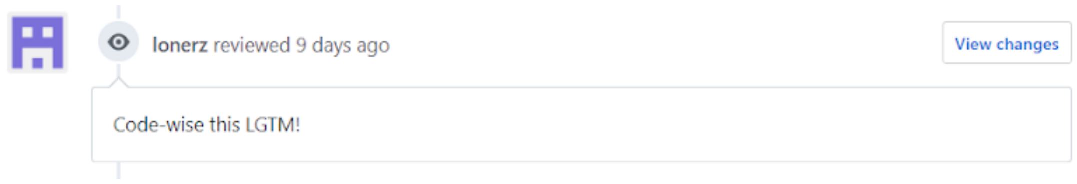
Justification

Since our task was writing tests, we couldn't write tests to test what we were working on. Instead, we used the resources available to us to ensure correctness. This mostly just included following the project's acceptance process, by getting our changes reviewed and passing the linting and CI builds. We communicated with the other developers and read existing code to fill in any gaps, especially to help ensure we were correctly understanding the context and requirements of our tasks.

QA Evidence

Task 1

Travis CI build:
https://travis-ci.org/zulip/zulip/builds/363330688


lonerz reviewed 9 days ago

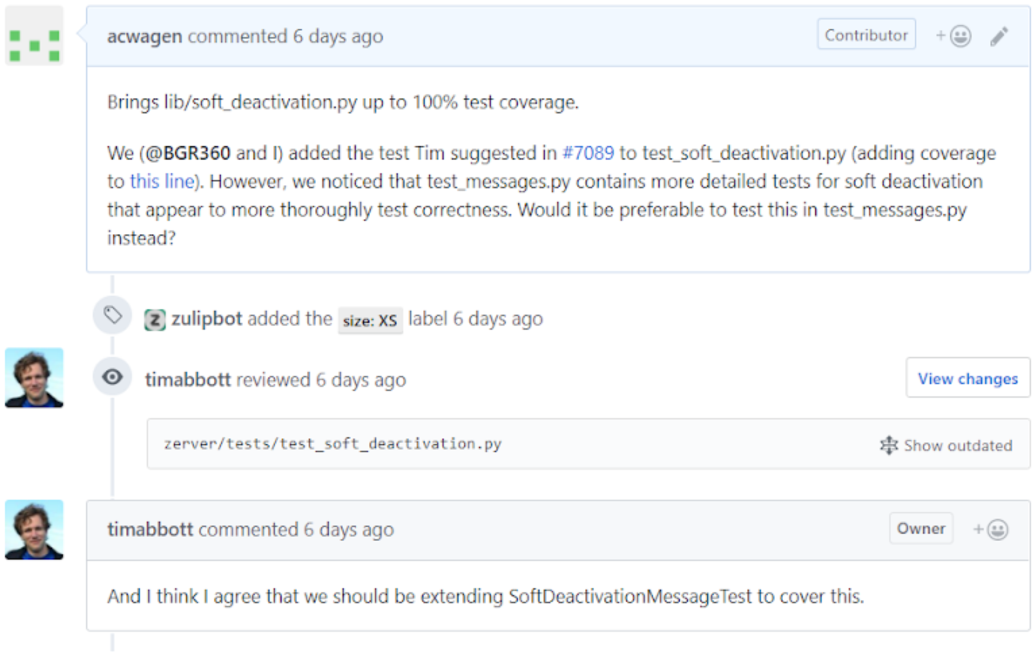
Code-wise this LGTM!

Code review on Task 1 PR.

Tasks 2 and 3

Circle CI Builds:
https://circleci.com/gh/zulip/zulip/5076
https://circleci.com/gh/zulip/zulip/5075

Travis CI Build:
https://travis-ci.org/zulip/zulip/builds/366903199



acwagen commented 6 days ago Contributor

Brings lib/soft_deactivation.py up to 100% test coverage.

We (@BGR360 and I) added the test Tim suggested in #7089 to test_soft_deactivation.py (adding coverage to this line). However, we noticed that test_messages.py contains more detailed tests for soft deactivation that appear to more thoroughly test correctness. Would it be preferable to test this in test_messages.py instead?

zulipbot added the size: XS label 6 days ago

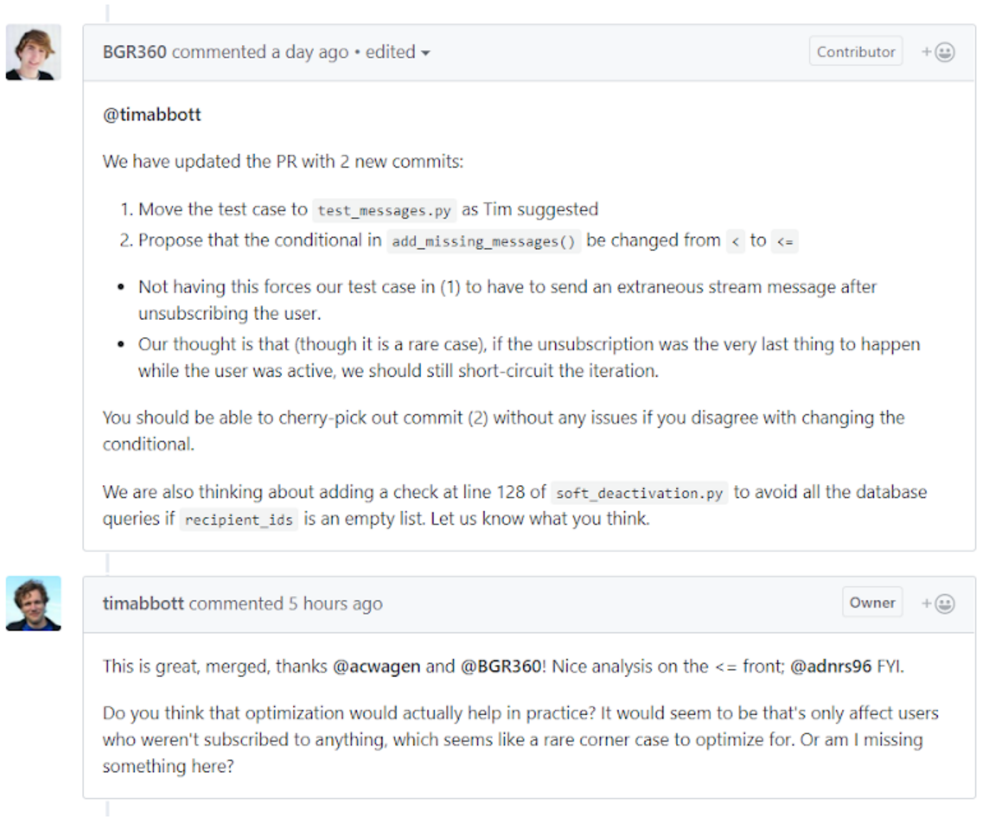
timabbott reviewed 6 days ago View changes

zerver/tests/test_soft_deactivation.py Show outdated

timabbott commented 6 days ago Owner

And I think I agree that we should be extending SoftDeactivationMessageTest to cover this.

Initial PR for Task 2 and lead developer’s code review.



BGR360 commented a day ago • edited Contributor

@timabbott

We have updated the PR with 2 new commits:

1. Move the test case to `test_messages.py` as Tim suggested
2. Propose that the conditional in `add_missing_messages()` be changed from `<` to `<=`

- Not having this forces our test case in (1) to have to send an extraneous stream message after unsubscribing the user.
- Our thought is that (though it is a rare case), if the unsubscription was the very last thing to happen while the user was active, we should still short-circuit the iteration.

You should be able to cherry-pick out commit (2) without any issues if you disagree with changing the conditional.

We are also thinking about adding a check at line 128 of `soft_deactivation.py` to avoid all the database queries if `recipient_ids` is an empty list. Let us know what you think.

timabbott commented 5 hours ago Owner

This is great, merged, thanks @acwagen and @BGR360! Nice analysis on the <= front; @adnrs96 FYI.

Do you think that optimization would actually help in practice? It would seem to be that's only affect users who weren't subscribed to anything, which seems like a rare corner case to optimize for. Or am I missing something here?

Updated PR for Task 2 after receiving feedback during code review.

Work Plan Updates

Plan Updates

The content and scope of our tasks changed somewhat significantly over the course of working on this project. Our original intention was to write test cases to bring several source files up to 100% code coverage. Instead, we implemented a new test-running script feature, wrote one test case to bring one source file up to 100% code coverage, and fixed a bug.

Change 1: Add a new feature to report files with 100% coverage to test-running script.

- This was suggested to us by one of the project owners when we were discussing what would be best for us to start working on. It seemed like a good task to start with because it was small in scope yet would help us become familiarized with the Zulip contribution process and the testing process.

Change 2: Work on soft_deactivation.py code coverage instead of actions.py and uploads.py.

- We thought it would be good to begin writing tests for a file that was already close to 100% coverage, so we started with soft_deactivation.py — after all, it only needed one more line to be covered. However, it took us a much more time than expected to understand the code enough to know how to extend the coverage. Additionally, we had lots of unexpected difficulties using the Zulip development environment, which created many roadblocks with installing the Vagrant VM. The environment itself proved to be very slow when it came to running our code. These unanticipated risks prevented us from having time to work on writing tests for more than one file.

Change 3: Fix a bug in soft_deactivation.py.

- While we were writing our test case, we came across a bug in soft_deactivation.py. We ended up fixing this along with adding the test case that revealed it.

Time Logging (in person-hours)

Identifying and becoming familiar with the project: 14 person-hours

- 2 hours searching repositories
- 4 hours pre-familiarizing with Zulip
- 1 hour communicating with project maintainers
- 7 hours setting up development environment and reading documentation for new contributors

Choosing, planning, and implementing tasks: 27 person-hours

- 2.5 hours working on Vagrantfile fix and test suite failure fix
- 5 hours working on test script feature
- 2 hours of process to get test script feature submitted

- 7 hours reading and understanding code, writing initial test case, and making an initial PR
- 5.5 hours reading existing test cases and identifying where and how to extend the test suite for our edge case
- 5 hours writing the final test case, fixing the bug, and updating the PR

Reflection and report: 12 person-hours
- 1 hour brainstorming initial thoughts for report
- 3 hours writing up initial outline for report
- 8 hours writing report

Our Open Source Experience

Learning curves and project community

Our experience contributing the Zulip project definitely confirmed our suspicions that the hardest part of software engineering is getting started. For a project like Zulip, with over 2,500 source files and 600,000 lines of code, figuring out where to start is overwhelming. Luckily, the Zulip team understands this, and they have put a huge amount of effort in making this process easier for new contributors.

As of today, there are nearly 100 pages of **documentation** specifically targeted at developers, and their GitHub ReadMe links directly to the "New Contributors" section. Additionally, all new contributors are encouraged to join the Zulip developer chat and ask questions. This chat has over 300 active users, and the lead developers respond to questions and issues every day. This chat proved to be the most influential contributor to our understanding and progression on this project because of the immediacy of feedback that it offered. Allowing new contributors to quickly engage in **knowledge transfer** with veteran developers helps prevent newcomers from giving up when they encounter roadblocks. In these ways, Zulip truly stands out as one of the most newcomer-friendly software projects in existence today.

However, despite the wealth of documentation and immediacy of support from core developers, getting comfortable with the codebase was still a substantial and effort-consuming aspect of this project. We had to learn how to use Zulip as a product, familiarizing ourselves with its functionality. We had to learn rebase-oriented Git workflow, which neither of us had used before. As we will discuss later, we also spent considerable time setting up and re-setting up our development environments. In the end, we did not end up writing a single line of code until after we had spent 14 person-hours getting started.

Design for collaboration and maintainability

Looking beyond the community and into the repository, it is clear that the Zulip team has made a concerted effort to design their code and documentation to facilitate **maintainability** and **readability**, both of which make it easier for newcomers like us to contribute. The core developers have laid out clearly defined **standards**, which helped eliminate ambiguity and uncertainty for us in terms of how to go about parts of the software engineering process. Furthermore, all code must go through an informal **code review** on GitHub by at least one core developer before it is merged to master. This ensured that all our code was clean and measured up to the readability **quality requirements** that the core developers seek to uphold. It also gave us the peace of mind necessary to write our own code, even under the uncertainty of whether or not it was quality, because we knew that a core developer would give us suggestions on how to fix it if it was bad.

Effort estimation, risk, and unexpected roadblocks

As expected, **effort estimation** in this project turned out to be difficult and inaccurate. In our initial work plan, we expected to be able to get multiple files up to 100% coverage, but in the end, we only brought one file (in fact, only one statement) up to full coverage. This drastic difference is due to high levels of **uncertainty** in the planning and development phases, unexpected changes to our work plan, and roadblocks that we encountered along the way.

Our main source of uncertainty in the planning phase was attempting to estimate how much knowledge would be required to come up with test cases for code we had not yet seen (spoiler: more than we thought). During development, our efforts were complicated by uncertainty about what the code was supposed to be doing, and if it was even doing it correctly (spoiler: it was not).

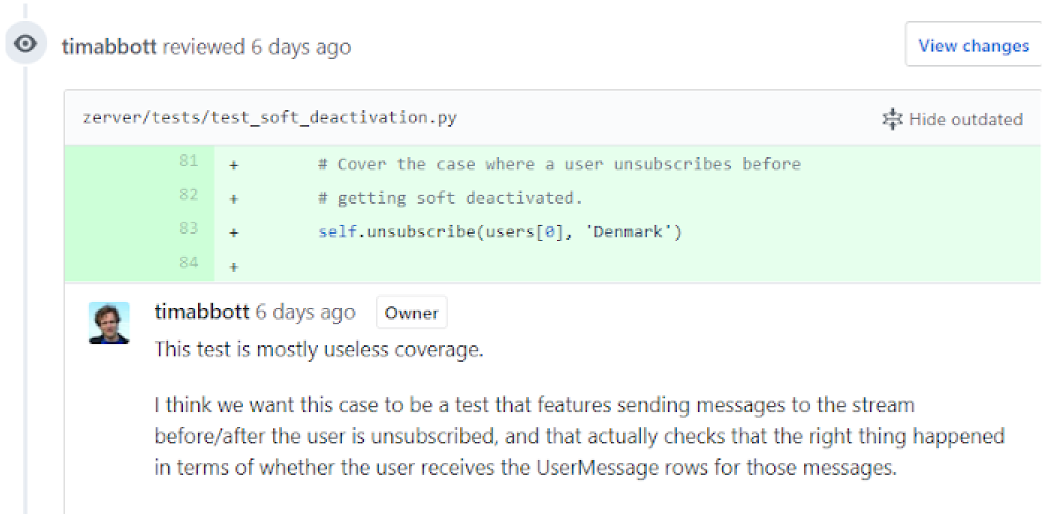
When we first began reading the test suite code and trying to figure out where to start writing tests, we decided to take a detour and tackle another issue related to our originally proposed task. This detour, detailed in Task 1 earlier in the report, took up around 30% of our allotted time. Another plan change we made was to start with an “easier” file (soft_deactivation.py, only 1 line to cover) before moving on to our originally proposed file (actions.py, ~50 uncovered lines) in order to get “warmed up” writing test cases. This “warm up” ended up taking up the remainder of our allotted time.

We also encountered a number of substantial roadblocks in regards to setting up the development environment. As is all too familiar in this class, setting up development environments can be a largely risky activity. This was definitely the case for our Windows machine., we encountered error after error and incompatibility after incompatibility when trying to set up and provision the necessary Vagrant development environment. Furthermore, the machine is old and incredibly slow, and so running the entire test suite took just under an hour, and running a single test case took a few minutes. Even on our much faster Mac OS machine, the entire test suite took about 10 minutes to run. This ended up being very inefficient for quick development and relates to our discussions on **system response time** in class.

Yet another setback we experienced was related to **understanding** the code. Initially, we figured that covering one line of code would require relatively little understanding of the backend. We were not wrong; however, it turns out that covering the line was not the real goal after all (see "Incomplete requirements…" below). What was difficult was writing a test case to verify the correctness of the code in soft_deactivation.py. This task was further complicated by the presence of a **bug** in the existing code. Because of a simple logic error in a conditional (of which we were suspicious but unsure of its invalidity), it was unexpectedly difficult for us to understand what the code was intended to do, and thus very difficult to write a test case for its correctness. Our solution was to write a test case testing the current state of the code in one commit, then change the code and the test case to reflect how we intuitively thought it should work, and submit that separately. It turns out we had indeed discovered a small bug, and the latter was accepted (see Task 3 above).

Incomplete requirements and perverse incentives

One particular story of interest in regards to Task 2 involves a lapse in **requirements elicitation** and an insight on the drawbacks of code coverage. In the original GitHub issue that we were addressing, the lead developer suggested that all that was needed to bring soft_deactivation.py to 100% coverage was "Just need a test in zerver/tests/test_soft_deactivation.py to cover the case that the user unsubscribed early." So, we went ahead and read this test file, which contained 4 very short tests. We mimicked what we saw in the file and added one line to one of the tests which successfully covered the target line in soft_deactivation.py. Upon submitting our pull request, here is what the lead developer had to say:


timabbott reviewed 6 days ago
View changes
zerver/tests/test_soft_deactivation.py
Hide outdated

```
+        # Cover the case where a user unsubscribes before
+        # getting soft deactivated.
+        self.unsubscribe(users[0], 'Denmark')
+
```

timabbott 6 days ago Owner

This test is mostly useless coverage.

I think we want this case to be a test that features sending messages to the stream before/after the user is unsubscribed, and that actually checks that the right thing happened in terms of whether the user receives the UserMessage rows for those messages.

Useless coverage.

This of course confirms what we learned in class about **code quality metrics** like code coverage: they can lead to **perverse incentives**. Initially, we were merely aiming to get this one line of code covered, and we had put no effort in to verifying the correctness of the code that was not being covered. Not only were we perversely incentivized, but we were also the victims of **incomplete requirements**. As new developers, we had no idea in which of the 71 test files we should write our tests, so we based our decision off what the lead developer had suggested in the original issue ("put it in test_soft_deactivation.py"). This however was not the proper course of action, as this file held very little information about the correct behavior of the code we were testing.

Advice for Future Students

Alyssa says: "Go to lecture, it's actually the best."

Ben says: "Don't underestimate the workload; it is a very different flavor of workload than other EECS classes at umich and may not be as easily tackled as you expect."

Cow says: "Moo."

We are willing to let future students see our materials.