Julia Kollin (jskollin)
Peter K. Shultz (pshultz)

HW6B: Project Report

I. Project Description

In HW6A we selected the Karrot project. Karrot is software that aids food saving efforts. Food saving groups can use the software to organize all logistics of their food pick-ups, reducing the need for spreadsheets or Doodle polls. Individuals can use the software to start their own food saving group, or join current ones in their area. A current version of the software is hosted here. The software was developed with the intention that each foodsaving group would host their own version of the website.

II. Project Context

Karrot is meant to be the general technology for the burgeoning number of food sharing groups that face issues with recruiting new members and coordinating pickups.

Karrot was started by Foodsaving World, an organization that describes itself as a “distributed, global grassroots movement against food waste”. The project itself was set up in July 2016 as an attempt to generalize foodsharing.de, a website with the same functionality as Karrot’s but limited only to Germany. A more concerted development effort for Karrot started in March 2017, according to Foodsaving World’s history page.

The idea of using technology to reduce food waste is not a new one. Applications and websites for food saving have received press since at least 2013, when NPR reported on an application called Leftoverswap started by two roommates at the University of Michigan. The application then received press after its launch in 2014 by The Guardian. Although the project lost its commercial ambitions and declared that it would be going open source in 2016, no commits have been made to the project’s GitHub repository since its 2014 launch.

One notable and current competitor to Karrot is Olio, a mobile app for food saving that “connects neighbours with each other and with local shops so surplus food and other items can be shared, not thrown away”. Olio is closed-source and for-profit, which is markedly different from Karrot’s open-source and non-profit approach.

III. Project Governance

The acceptance process aligns with the informal, pass-around code review approach. Contributors make a pull request to the master repository, and project maintainers review the updates and either accept the changes, return the code with comments, or completely reject the request. Any additional communication among contributors and repository maintainers is done via GitHub or Slack. All code changes are required to meet style guidelines and are assumed to have been tested before being submitted in a pull request. Integration tests are performed once the pull request has been made, and are reviewed before any code changes are accepted. Details regarding style and quality assurance requirements are elaborated on below.

Communication

- Coordination amongst Karrot contributors is primarily done through Slack, the team messaging app. Communication is broken up into the following channels:
  - **#karrot-dev**, for discussing general development of the project.
  - **#karrot-rgsoc**, for preparing an application for the Rails Girls Summer of Code.
  - **#karrot-funding**, for leads on financial support for Karrot and its developers.
  - **#karrot-git**, for notifications from GitHub regarding issues and Netlify regarding deployments.
  - **#karrot-mails**, for a list of emails received by Karrot's general email: karrot@foodsaving.world
  - **#karrot-sentry**, for notifications from Sentry, the crash reporting and aggregation tool.

Style Standards and Expectations

- While developing code, the developers use editorconfig, which enables developers to specify code style standards. Plugins for this are available on many different platforms, enabling developers to maintain code standards across different editors and IDEs.
- The developers follow the PEP8 style guide, which is the same coding standard as the ones defined in the Django project. Running `flake8` in the root repository checks code for compliance to the above standard. As explained in the `README`, code will only be accepted if it passes the PEP8 test.
- Developers recommend installing an `eslint` plugin for your IDE. `eslint` finds JavaScript code that does not adhere to certain style guidelines through a static analysis approach.
- Any text added to the front end must be translated via the transifex command line client. This is done by the repository maintainers once the request had been accepted.

Quality Assurance

- Unit/component tests must be created for any new or updated functionality. All new and existing tests must pass before a pull request is accepted.
- The Codecov tool is used to compute file, line, and branch coverage for every pull request submitted. An example from one of our pull requests can be found here. This is performed when the PR is submitted.
- CircleCI is used for continuous integration testing: before a commit is successfully merged into master, the modified code is run against several test cases related to:
    - Builds for the web, Android, and Apache Cordova.
    - Installations of dependencies.
    - Linting.
    - Deployment previews.
    - Code coverage.
- Netlify is used to deploy new code from both the front end and the back end to Karrot's main site: karrot.world. Before a deployment is made, Netlify spins up a preview site for inspection. By doing this, developers can ensure the site has all the functionality expected before deploying for real. An example can be found here.

IV. Task Descriptions

Enable edit on conversations

Members of a particular conversation can post messages on the conversation wall. However, there is no way to edit a message once it has been posted. This task required updating the back end source code to enable users to edit their own messages. We assumed we had to update the front and back end, however feedback from one of the project developers explained we only had to work on the back end implementation. We originally added an API endpoint that accepted a `POST` request (`/api/messages/{messageId}/edit/`) with a JSON object of the new message content. However, when we asked one of the developers for thoughts on our approach, he believed a `PATCH` request to be a more ReSTful solution with the Django REST framework. Therefore, we changed this to a `PATCH` endpoint (`/api/messages/{messageId}/`) instead. We also added permission checks to ensure the following: no user attempted to update a message they did not create; no user attempted to edit a message in a conversation they were not a member of; and no user updated a message more than 10 days after its creation. Once we got an idea of how we were going to implement the feature and did some prototyping, we wrote test cases to ensure that our requirements defined in 6A were met.

Notice for outdated browsers

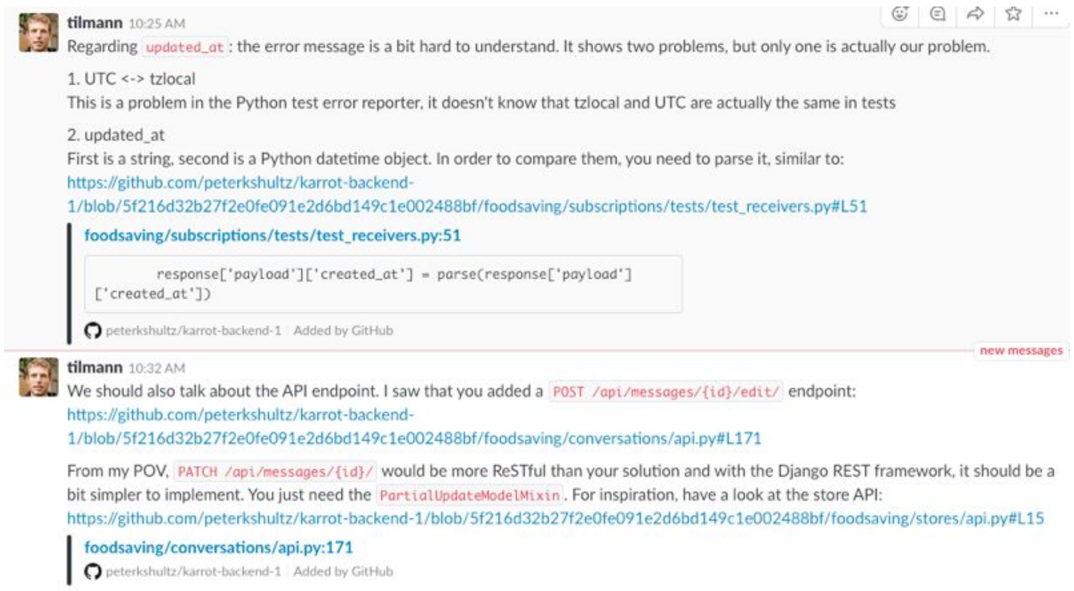
Currently, Karrot users with outdated browsers cause silent frontend errors. As such, we wanted to inform users that their browser was old and provide them a link to download a new one. This task involved determining the most efficient way to detect browsers, the best place to locate browser detection code, and the best front end components to use to display a warning message. After investigating various methods of browser detection, the best choice was the `browser-detect` NPM package. After experimenting with various places to invoke the package, the next step was determining the best way to alert the user. Although a JavaScript window alert was first proposed, it was rejected in favor of a less intrusive paragraph at the top. Additionally, further changes needed to be made to support internationalization support: because many Karrot users are unfamiliar with English, it did not make sense to include warning messages in one language alone. Our changes were made in such a way that translation volunteers can easily use a tool called Transifex and translate our warning message into other languages.

V. Submitted Artifacts

We submitted the following pull requests after fully reviewing Karrot's code of conduct and contribution guidelines:

1. Pull request for addition of conversation edit endpoints: In terms of implementing the feature, most of our changes are within `foodsaving/conversations/api.py`. Within that class, we added the permission classes: `IsAuthorConversationMessage` (line 61) and `IsWithinUpdatePeriod` (line 70). These classes check whether the user trying to update the message is the author of the message, and whether the message is being updated within 10 days after its creation. We then updated the `ConversationViewSet` class (line 82) to inherit from the `PartialUpdateModelMixin` class. `PartialUpdateModelMixin` has a method, `partial_update()`, that accepts an object, and the member variable that must be updated within that object. We used this method to update the message content on a patch request. In terms of testing, we added test cases in the `test_api.py` file within the `TestCoversationsMessageEditPatchAPI` class.

   Figure 0 below is an image of the initial feedback we received from one of the project maintainers, encouraging us to change our implementation approach to a patch request, rather than a post. Further conversation evidence with project maintainers can be found in the link to the pull request above.



tilmann 10:25 AM

Regarding `updated_at`: the error message is a bit hard to understand. It shows two problems, but only one is actually our problem.

1. UTC <-> tzlocal
This is a problem in the Python test error reporter, it doesn't know that tzlocal and UTC are actually the same in tests

2. updated_at
First is a string, second is a Python datetime object. In order to compare them, you need to parse it, similar to:
https://github.com/peterkshultz/karrot-backend-1/blob/5f216d32b27f2e0fe091e2d6bd149c1e002488bf/foodsaving/subscriptions/tests/test_receivers.py#L51

foodsaving/subscriptions/tests/test_receivers.py:51

```
        response['payload']['created_at'] = parse(response['payload']
['created_at'])
```

peterkshultz/karrot-backend-1 | Added by GitHub

new messages

tilmann 10:32 AM

We should also talk about the API endpoint. I saw that you added a `POST /api/messages/{id}/edit/` endpoint:
https://github.com/peterkshultz/karrot-backend-1/blob/5f216d32b27f2e0fe091e2d6bd149c1e002488bf/foodsaving/conversations/api.py#L171

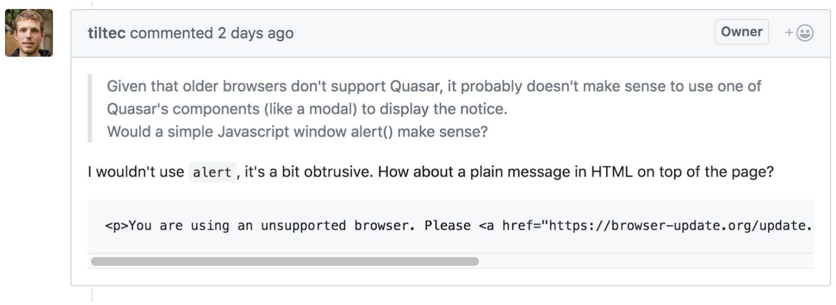
From my POV, `PATCH /api/messages/{id}/` would be more ReSTful than your solution and with the Django REST framework, it should be a bit simpler to implement. You just need the `PartialUpdateModelMixin`. For inspiration, have a look at the store API:
https://github.com/peterkshultz/karrot-backend-1/blob/5f216d32b27f2e0fe091e2d6bd149c1e002488bf/foodsaving/stores/api.py#L15

foodsaving/conversations/api.py:171

peterkshultz/karrot-backend-1 | Added by GitHub

Figure 0: Tilmann's feedback on our initial implementation, given to us through Slack.

2. Pull request for warning to users with outdated browsers: Upon first submission, this pull request was failing continuous integration tests related to builds and deployment, due to a new NPM package whose addition was not reflected in a few key build files. Additionally, the method we were using to warn users--an `alert()` method through JavaScript--was rejected upon initial submission due to user experience concerns. There were also concerns about code readability and translatability, so the original changes made to `MainLayout.vue` had to be revised.



tiltec commented 2 days ago

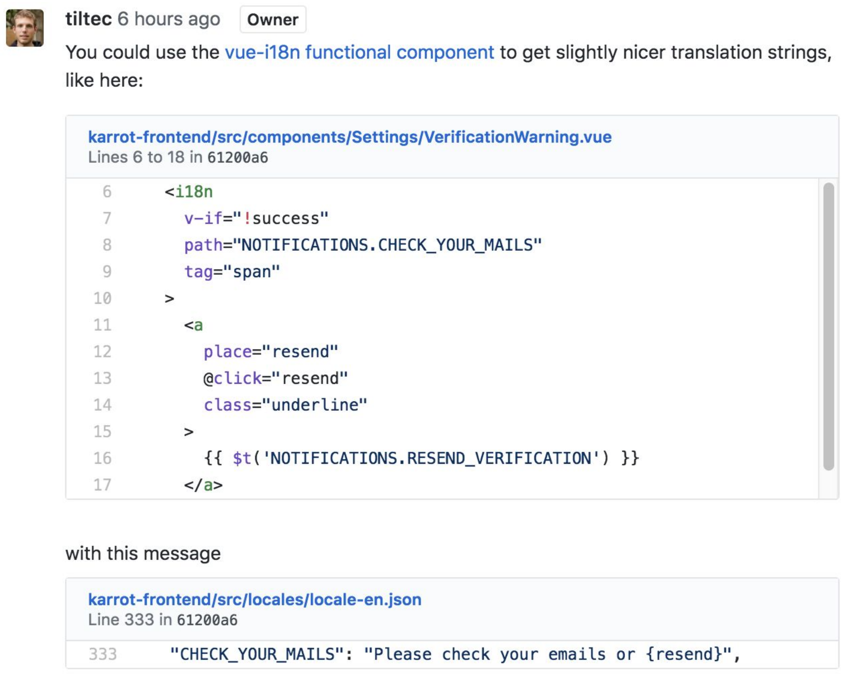
Owner

> Given that older browsers don't support Quasar, it probably doesn't make sense to use one of Quasar's components (like a modal) to display the notice.
> Would a simple Javascript window alert() make sense?

I wouldn't use `alert`, it's a bit obtrusive. How about a plain message in HTML on top of the page?

```
<p>You are using an unsupported browser. Please <a href="https://browser-update.org/update.
```

Figure 1: Tilmann suggesting a revision to a plain HTML message instead of a JavaScript `alert()`.


tiltec 6 hours ago Owner

You could use the vue-i18n functional component to get slightly nicer translation strings, like here:

karrot-frontend/src/components/Settings/VerificationWarning.vue
Lines 6 to 18 in 61200a6

```
    <i18n
      v-if="!success"
      path="NOTIFICATIONS.CHECK_YOUR_MAILS"
      tag="span"
    >
      <a
        place="resend"
        @click="resend"
        class="underline"
      >
        {{ $t('NOTIFICATIONS.RESEND_VERIFICATION') }}
      </a>
```

with this message

karrot-frontend/src/locales/locale-en.json
Line 333 in 61200a6

```
    "CHECK_YOUR_MAILS": "Please check your emails or {resend}",
```

Figure 2: Tilmann providing example code for reference to help us enhance internationalization.

VI. QA Strategy

Integration and code style tests are performed for every submitted pull request. The results of these can be found in the links to the GitHub pull requests.

Addition of conversation edit endpoints

- *Code style*: As mentioned previously, there are certain style guidelines that are required for any code changes. We ran `flake8` on our updated code to ensure everything we added adhered to the set style guidelines.
- *Unit testing*: Since we added to the `/api/messages/` endpoint, there were already several existing test cases written for the `GET` and `POST` requests within this endpoint. Therefore, we structured our tests similar to the tests already there. We tested the functionality of the feature, as well as tested our permission checks (users cannot edit messages they did not author, users cannot update a message not found in the conversation, users cannot edit a message more than 10 days after its creation). We ran our unit tests, along with all unit tests written for the project, to ensure that our code worked as expected and that we did not break any other component of the project.
- *End-to-end testing*: While not required by the developers, we “hacked” the front end source code to ensure that our back end changes would eventually integrate nicely with the front end. A `create` function is called whenever a conversation member adds a new message to the conversation. We changed this code so that rather than the `create`

endpoint being called, the `update` endpoint would be called instead (with pre-set parameters manually added by us). We were able to see that front end messages would actually update in response to a call to our endpoint.

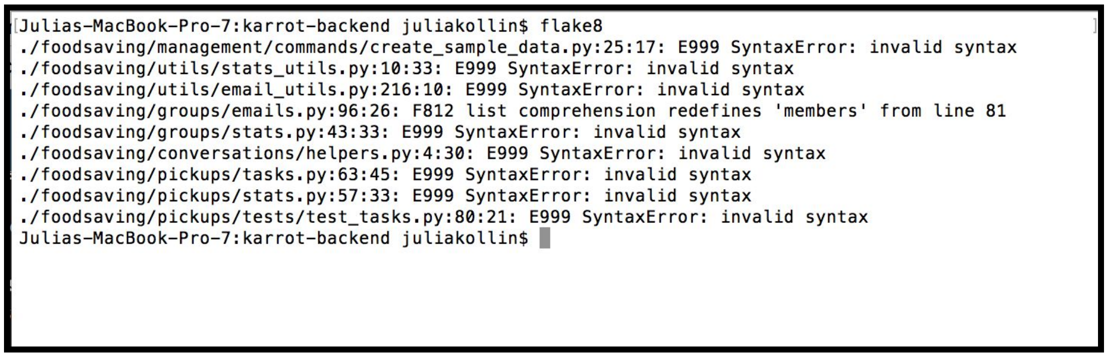
Warning to users with outdated browsers

- *Code style*: In order for Karrot's front end code to compile, it had to pass an `eslint` style test. As a result, our code had to be compliant with style guidelines in order to even begin functional testing. We preferred this over changing code to be style compliant before checking it into GitHub, as we gently came to learn `eslint` standards (as opposed to fighting the style checker just to get our code pushed so that we could work on something else). This effectively served as a coach on our coding style, which we appreciated.
- *End-to-end testing*: We confirmed our front end code worked as expected by manually walking through the steps required to invoke the functionality we were trying to add. While it is possible to test such behavior automatically with tools like Selenium, we opted out of such an approach due to the reportedly steep learning curve of the tool coupled with a shortened timeline for testing. In other words, it made more sense to manually test given the limited time horizon that automated testing could pay dividends (similar to what we learned in class about cost-benefit analyses).
- *Backwards compatibility testing*: By nature of creating a warning for outdated browsers, we needed to ensure the warning displayed properly on outdated browsers. Although we first tried this using proprietary tools like BrowserStack, we ultimately opted for a simple browser extension that changed the HTTP User-Agent string: not only was it free for unlimited use, but it was less time-consuming to configure.

VII. QA Evidence

Addition of conversation edit endpoints

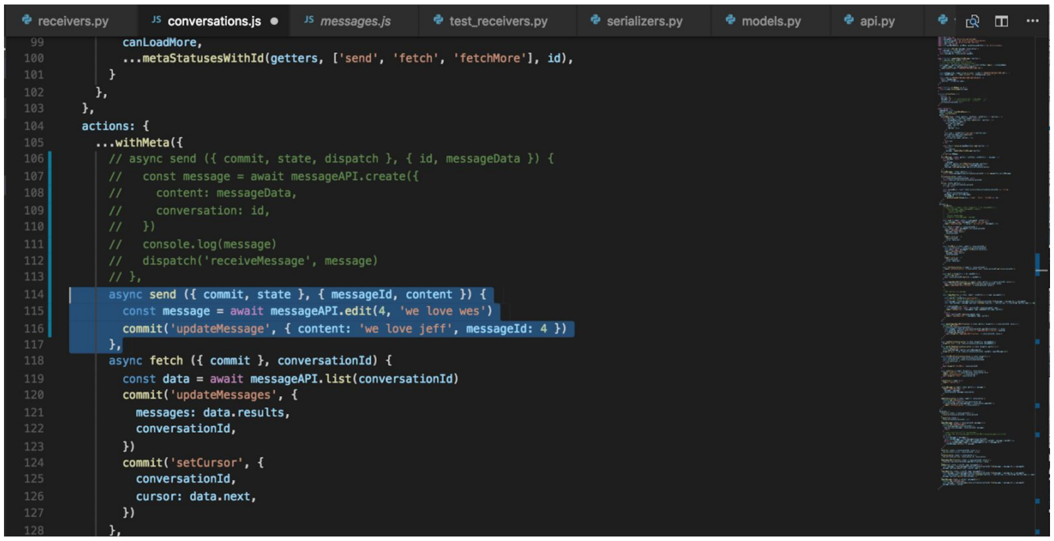
- Image of our `flake8` run: All errors originate from files that were not updated by our code changes.



```
Julias-MacBook-Pro-7:karrot-backend juliakollin$ flake8
./foodsaving/management/commands/create_sample_data.py:25:17: E999 SyntaxError: invalid syntax
./foodsaving/utils/stats_utils.py:10:33: E999 SyntaxError: invalid syntax
./foodsaving/utils/email_utils.py:216:10: E999 SyntaxError: invalid syntax
./foodsaving/groups/emails.py:96:26: F812 list comprehension redefines 'members' from line 81
./foodsaving/groups/stats.py:43:33: E999 SyntaxError: invalid syntax
./foodsaving/conversations/helpers.py:4:30: E999 SyntaxError: invalid syntax
./foodsaving/pickups/tasks.py:63:45: E999 SyntaxError: invalid syntax
./foodsaving/pickups/stats.py:57:33: E999 SyntaxError: invalid syntax
./foodsaving/pickups/tests/test_tasks.py:80:21: E999 SyntaxError: invalid syntax
Julias-MacBook-Pro-7:karrot-backend juliakollin$
```

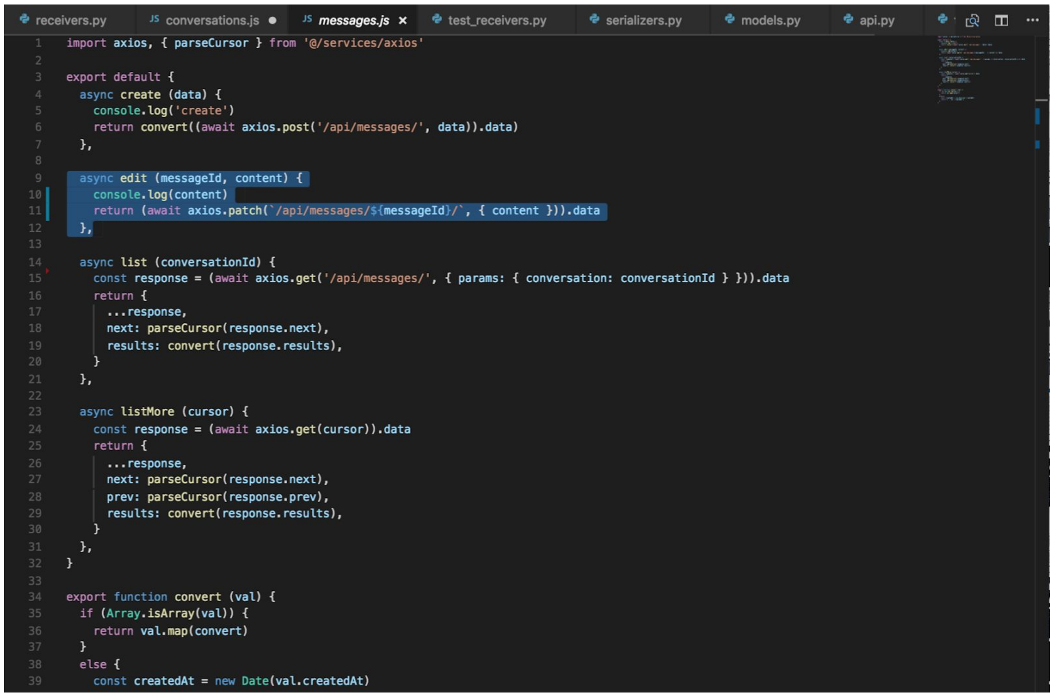
Figure 3: The results of running `flake8` in our root repository.

- Link to our added unit tests: We added test cases in the `test_api.py` file within the `TestCoversationsMessageEditPatchAPI` class.
- Image of our “hacked” front end UI code to test our endpoint: As you can see in Figure 4, we removed the existing `send` function in the `conversations.js` file and replaced it with code that would call the edit function in the `messageAPI`. The `send` method is called from the UI when the user enters a new message to add to the conversation. We updated the `messageAPI` in the `messages.js` file (evident in Figure 5) to include an `edit` function, which then made the patch request to our added endpoint.



```
          canLoadMore,
          ...metaStatusesWithId(getters, ['send', 'fetch', 'fetchMore'], id),
        }
      },
    },
    actions: {
      ...withMeta({
        // async send ({ commit, state, dispatch }, { id, messageData }) {
        //   const message = await messageAPI.create({
        //     content: messageData,
        //     conversation: id,
        //   })
        //   console.log(message)
        //   dispatch('receiveMessage', message)
        // },
        async send ({ commit, state }, { messageId, content }) {
          const message = await messageAPI.edit(4, 'we love wes')
          commit('updateMessage', { content: 'we love jeff', messageId: 4 })
        },
        async fetch ({ commit }, conversationId) {
          const data = await messageAPI.list(conversationId)
          commit('updateMessages', {
            messages: data.results,
            conversationId,
          })
          commit('setCursor', {
            conversationId,
            cursor: data.next,
          })
        },
```

Figure 4: Our “hacked” code in the `conversation.js` file. We renamed what would be our `edit` method to be called `send`, and commented out the actual `send` method.

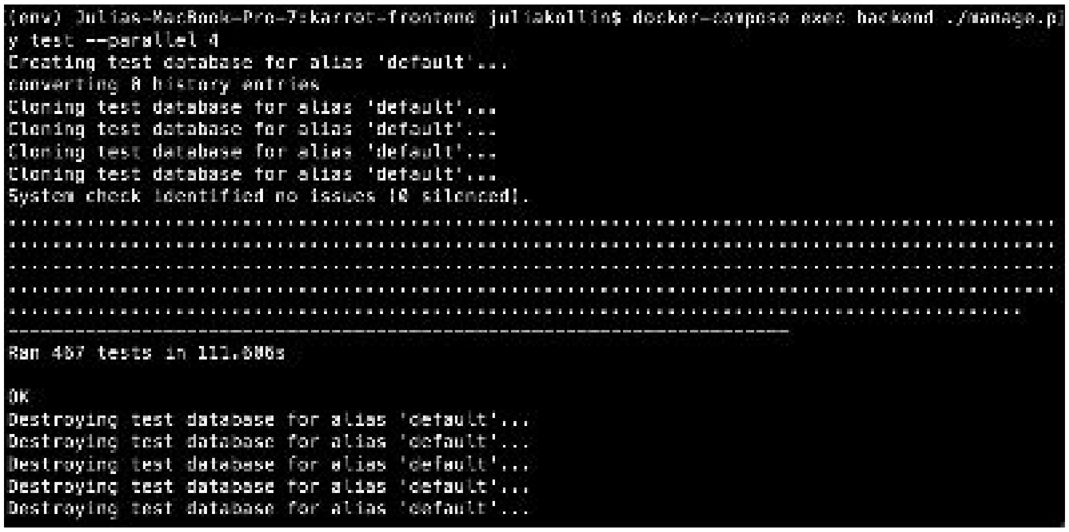


receivers.py | conversations.js | messages.js | test_receivers.py | serializers.py | models.py | api.py

```
import axios, { parseCursor } from '@/services/axios'

export default {
  async create (data) {
    console.log('create')
    return convert((await axios.post('/api/messages/', data)).data)
  },

  async edit (messageId, content) {
    console.log(content)
    return (await axios.patch(`/api/messages/${messageId}/`, { content })).data
  },

  async list (conversationId) {
    const response = (await axios.get('/api/messages/', { params: { conversation: conversationId } })).data
    return {
      ...response,
      next: parseCursor(response.next),
      results: convert(response.results),
    }
  },

  async listMore (cursor) {
    const response = (await axios.get(cursor)).data
    return {
      ...response,
      next: parseCursor(response.next),
      prev: parseCursor(response.prev),
      results: convert(response.results),
    }
  },
}

export function convert (val) {
  if (Array.isArray(val)) {
    return val.map(convert)
  }
  else {
    const createdAt = new Date(val.createdAt)
```

Figure 5: The method called from the `conversations.js` class that makes the call to the patch endpoint.

- Image of the results of running all source code tests can be seen in Figure 6.



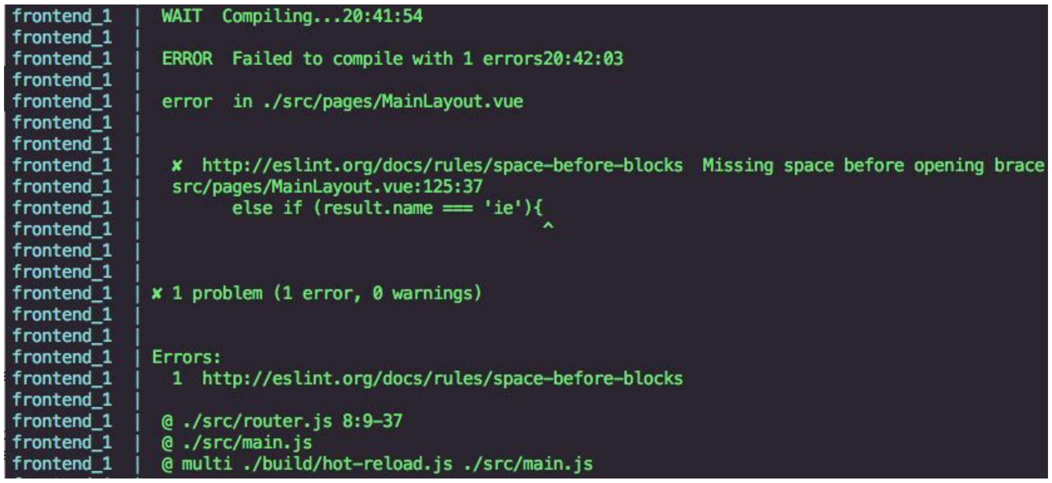
```
(env) Julias-MacBook-Pro-7:karrot-frontend juliakollin$ docker-compose exec backend ./manage.p
y test --parallel 4
Creating test database for alias 'default'...
converting 8 history entries
Cloning test database for alias 'default'...
Cloning test database for alias 'default'...
Cloning test database for alias 'default'...
Cloning test database for alias 'default'...
System check identified no issues (0 silenced).
..........................................................................................
..........................................................................................
..........................................................................................
..........................................................................................
.......................................................................................
----------------------------------------------------------------------
Ran 467 tests in 111.886s

OK
Destroying test database for alias 'default'...
Destroying test database for alias 'default'...
Destroying test database for alias 'default'...
Destroying test database for alias 'default'...
Destroying test database for alias 'default'...
```

Figure 6: The result of running all source code unit tests (including our own).

- The integration tests for the PR can be found here. All passed!

Pull request for warning to users with outdated browsers

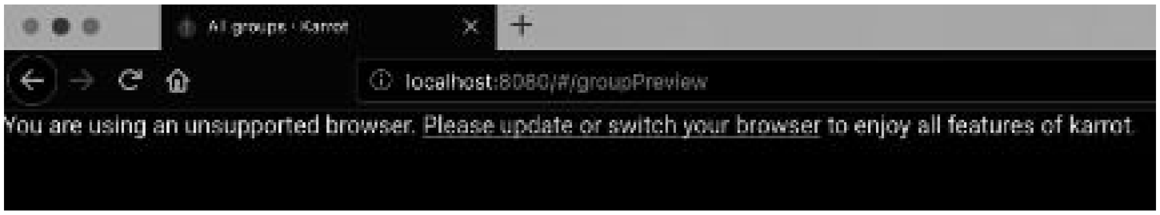
- Evidence of `eslint` usage:



```
frontend_1  |  WAIT  Compiling...20:41:54
frontend_1  |
frontend_1  |  ERROR  Failed to compile with 1 errors20:42:03
frontend_1  |
frontend_1  |  error  in ./src/pages/MainLayout.vue
frontend_1  |
frontend_1  |
frontend_1  |   ✘  http://eslint.org/docs/rules/space-before-blocks  Missing space before opening brace
frontend_1  |   src/pages/MainLayout.vue:125:37
frontend_1  |         else if (result.name === 'ie'){
frontend_1  |                                       ^
frontend_1  |
frontend_1  |
frontend_1  | ✘ 1 problem (1 error, 0 warnings)
frontend_1  |
frontend_1  |
frontend_1  | Errors:
frontend_1  |   1  http://eslint.org/docs/rules/space-before-blocks
frontend_1  |
frontend_1  |  @ ./src/router.js 8:9-37
frontend_1  |  @ ./src/main.js
frontend_1  |  @ multi ./build/hot-reload.js ./src/main.js
```

Figure 7: Unless `eslint` exits with zero errors, the entire front end fails to compile.

- Evidence of end-to-end testing:



You are using an unsupported browser. Please update or switch your browser to enjoy all features of karrot.

Figure 8: The appearance of the outdated browser warning, demonstrating the we properly tested end-to-end.

- Evidence of backwards compatibility testing with [Browserstack](#):

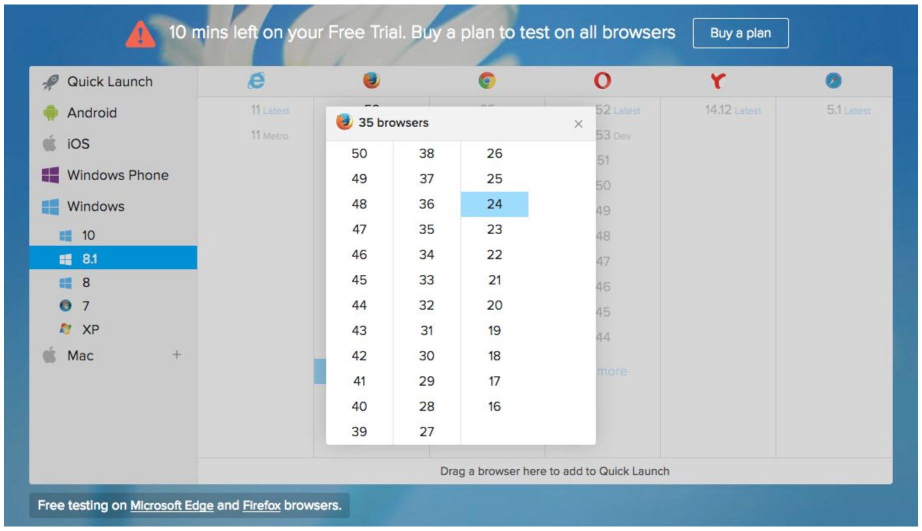


Figure 9: The interface for [BrowserStack](#), where we selected older versions of Firefox for testing the compatibility of [Quasar framework](#) components.

- Evidence of backwards compatibility testing with the User-Agent Switcher browser extension:

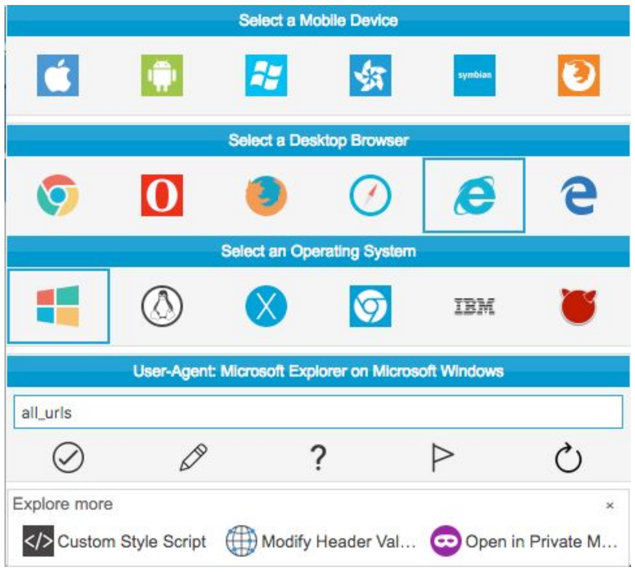
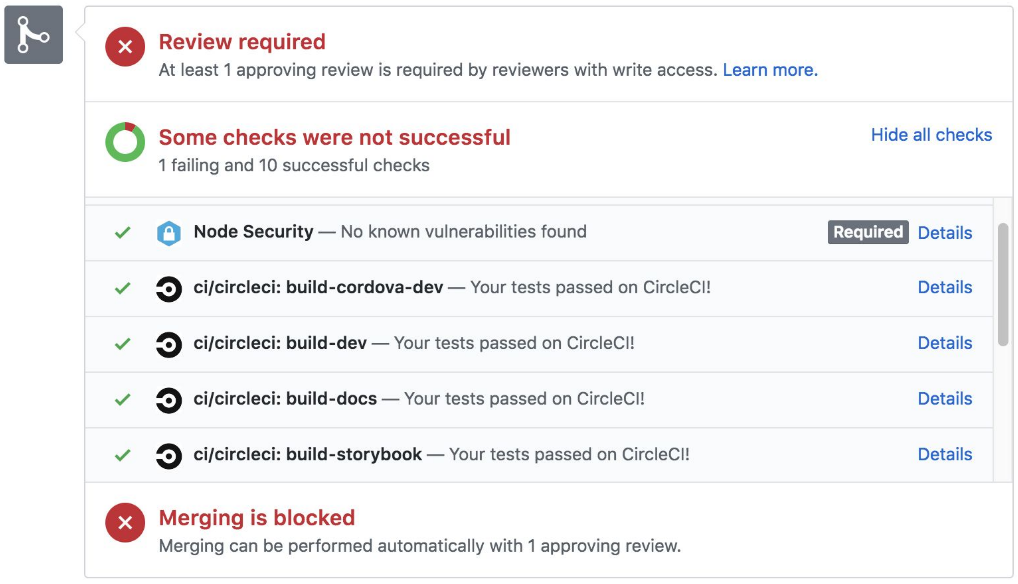


Figure 10: The interface for User-Agent Switcher, where we could easily change the browser we were testing.

- Evidence of integration tests upon submission of the pull request:
    - For edit endpoints: #1, #2, #3
    - For outdated browser warning: #1, #2, #3, #4, #5, #6, #7, #8, #9, #10, #11



Review required
At least 1 approving review is required by reviewers with write access. Learn more.

Some checks were not successful Hide all checks
1 failing and 10 successful checks

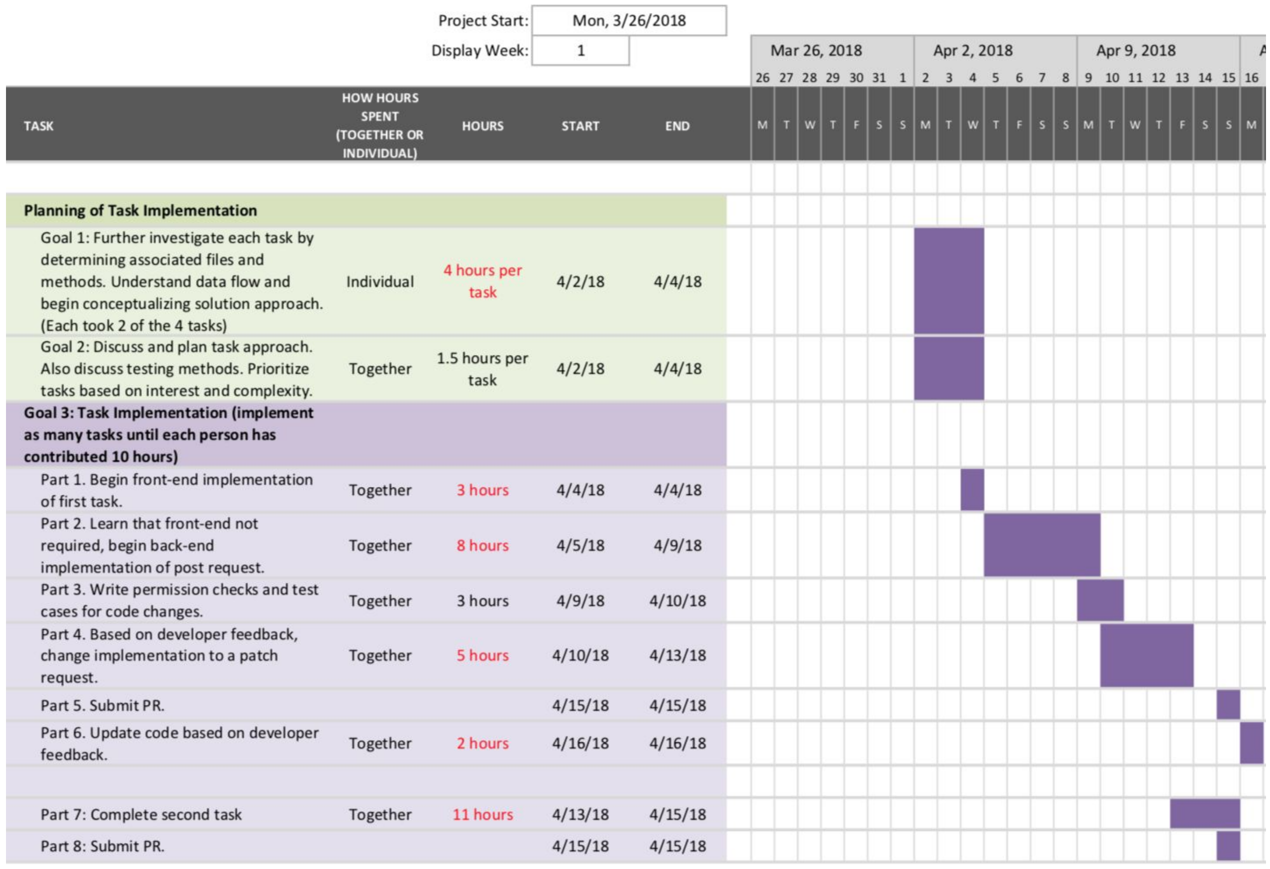
✓ **Node Security** — No known vulnerabilities found Required Details
✓ **ci/circleci: build-cordova-dev** — Your tests passed on CircleCI! Details
✓ **ci/circleci: build-dev** — Your tests passed on CircleCI! Details
✓ **ci/circleci: build-docs** — Your tests passed on CircleCI! Details
✓ **ci/circleci: build-storybook** — Your tests passed on CircleCI! Details

Merging is blocked
Merging can be performed automatically with 1 approving review.

Figure 11: A snapshot of integration tests for the outdated browser warning pull request. These tests were run after a pull request is initially submitted, as well as any time a pull request is updated with a new commit.

VIII. Plan Updates

Figure 12 is a picture of our updated timeline. The items highlighted in red took longer than expected. We elaborate on the reasoning behind this in the individual task sections below. In general, investigating each task (Goal 10) look longer than expected, simply because we were walking through an unknown code base that was much larger than some of the previous projects we have encountered. In terms of implementation, this took much longer than expected, especially counting the hours that resulted from redesigning or fixing code based on developer feedback. One of the code maintainers, Timann, did warn us that our original effort estimations were incorrect, and we would most likely only be able to get one or two tasks completed in the provided time frame. Of course, Tilman was correct and we were only able to implement our first and second prioritized tasks.



Project Start:	Mon, 3/26/2018
Display Week:	1

TASK	HOW HOURS SPENT (TOGETHER OR INDIVIDUAL)	HOURS	START	END
Planning of Task Implementation				
Goal 1: Further investigate each task by determining associated files and methods. Understand data flow and begin conceptualizing solution approach. (Each took 2 of the 4 tasks)	Individual	4 hours per task	4/2/18	4/4/18
Goal 2: Discuss and plan task approach. Also discuss testing methods. Prioritize tasks based on interest and complexity.	Together	1.5 hours per task	4/2/18	4/4/18
Goal 3: Task Implementation (implement as many tasks until each person has contributed 10 hours)				
Part 1. Begin front-end implementation of first task.	Together	3 hours	4/4/18	4/4/18
Part 2. Learn that front-end not required, begin back-end implementation of post request.	Together	8 hours	4/5/18	4/9/18
Part 3. Write permission checks and test cases for code changes.	Together	3 hours	4/9/18	4/10/18
Part 4. Based on developer feedback, change implementation to a patch request.	Together	5 hours	4/10/18	4/13/18
Part 5. Submit PR.			4/15/18	4/15/18
Part 6. Update code based on developer feedback.	Together	2 hours	4/16/18	4/16/18
Part 7: Complete second task	Together	11 hours	4/13/18	4/15/18
Part 8: Submit PR.			4/15/18	4/15/18

Figure 12: Revised schedule of our implementation efforts.

Enable edit on conversations

This task took much longer than expected, especially with a complete implementation change halfway through. We implemented the entire feature using a `POST` API endpoint, and then was asked to update this to a `PATCH` endpoint. While this seems like a minor change, it required us not only to change our endpoint, but our testing and permission checking as well. In addition, we spent several hours initially prototyping and planning our approach to front end changes. It was not until several days after that we learned that this was not expected of us. Moreover, we found initial implementation attempts to be extremely difficult because we were not using the proper development environment. It is extremely difficult debugging back end python code without an IDE like PyCharm. While we incorrectly believed we did not need an IDE like PyCharm (and thought it was too complicated and not worth it on our first attempt to use it), it became increasingly difficult to debug using print-statements. It was not until much later in the process did we fully understand why IDEs like PyCharm are so crucial.

Warning to users with outdated browsers

As detailed in the first report for this project, the original implementation plan had been to use a Vue plugin from browser-update.org to complete this task. The project authors had already thought through some of the difficult tasks associated with browser warnings, and the plugin seemed easy to use. After speaking with some of the project maintainers, however, this approach was flawed: not only did we have trouble finding the proper place to invoke the plugin, but the package was large for its limited functionality (10kB minified and gzipped). This led us to look for lighter packages, which brought us to `browser-detect` and `detect-browser`. Although `detect-browser` was lighter according to Bundlephobia, we chose `browser-detect` because of its installation compatibility with `yarn` (which the rest of the project uses), as well as its simpler interface and implementation in the context of the Vue files we were editing.

IX. Experiences and Recommendations

Overall Experience

Overall, we found this homework extremely beneficial and found that it solidified our understanding of important concepts we learned this semester. It was interesting to analyze the planning and design decisions made by project developers (for ex: in terms of quality assurance metrics and automated tool usage), and understand the rationale behind each choice. In addition, we recognized many design patterns within the code base, including a MVC structure, the observer pattern, and factory methods. Without a prior understanding of these design patterns, it would have been much more difficult to understand interactions between different components of the project. (We believe that this an example of top-down comprehension- where code cues and prior knowledge helped us gain a better understanding of the code).

Development Environment

As we have learned through previous homeworks in this course, setting up the proper development environment and getting a program to build can be extremely difficult and time consuming. Shockingly, we found the process of building the project relatively easier than expected, given the detailed steps provided in the repository. In addition, the front end and back end repositories were ran within a docker setup, enabling us to run both, simultaneously, with one line of input. There were several hiccups we encountered along the way, but nothing stackoverflow could not help us with.

On the other hand, as mentioned previously, we had a very difficult time setting up PyCharm, something that would have been extremely beneficial to our productivity. We eventually gave up, thinking that our IDEs (VSCode and Sublime) would be sufficient. Now that we understand the benefits of PyCharm, taking the time to understand and set-up PyCharm before we began our development process would have saved us a substantial amount of time. Through this experience we really understand the importance of choosing the best development environment and tools for a given project. Not all projects are similar, and each requires thought into choosing the proper languages, tools, and environment.

Communication with Project Maintainers

In terms of communication, communication with project maintainers was easy: Tilmann Becker, one of Karrot's original developers, was happy to communicate with us over Slack and GitHub. Upon first reaching out to him, we explained that our work would be part of a graded assignment. He was supportive of our efforts, despite limited open source contributions, which was refreshing given that many maintainers are not welcoming to beginners at all. His immediate and detailed responses were pleasantly surprising, creating an environment where we felt encouraged and excited to keep contributing. In fact, Tilmann went so far as to speak with other maintainers to find supervisors for our pull requests. One difficulty was speaking with Tilmann and other maintainers during working hours: since most of them are based in Germany, we were only able to speak with them for limited portions of time in between our classes before they went to bed. Because we both worked on the project at night after our classes were finished, we had to save any questions we had for the maintainers until the next day. And although the project maintainers gave very helpful feedback, we think we could have worked more effectively had we been able to speak to them more frequently. We'd recommend that future students take the geographic location of maintainers into consideration when choosing a project. Although maintainer friendliness and responsiveness is more important, being able to communicate during mutual working hours is very nice to have.

Trouble Understanding, Changing, and Contributing

We were surprised at some of the non-functional requirements that project maintainers had to consider when building Karrot. As we mentioned in previous sections, our original plan for warning users about outdated browsers was rejected due to the size of the package. We had not realized that JavaScript code bloat was an issue until one of the maintainers had mentioned Bundlephobia for checking whether the dependencies we were adding were heavy. It's important to note that these non-functional requirements were not verifiable: while project maintainers noted that code bloat was something they kept an eye out for, they didn't provide any objective metrics for measuring whether a specific package was sufficiently light given its feature set. While understandable given the difficulty of measuring an inherently immeasurable quantity (“how feature-rich is this package given its size?”), this highlights the importance of defined ways to analyze and measure non-functional requirements.

In addition, we were disappointed with the lack of comments in the source code. We found it more difficult to understand code flow between components, and this involved extra investigation on our end. On the other hand, we thought variable and method names were extremely well chosen, making it extremely intuitive to understand what each object represented and the general functionality of each method. This furthered our understanding of the importance of proper documentation and naming mechanics.

Unanticipated Challenges

After writing and testing our added code, we ran all of the projects test cases, ensuring our changes did not break any other components. While only several test cases needed to be updated due to our updates, it still took a substantial amount of time to understand the resulting errors and determine whether our code had a bug, or the test cases needed to be updated. For one continually failing test case, we had to reach out to one of the project maintainers for feedback and advice on the best solution approach.

While neither pull request was accepted immediately, we believe updates to our pull request for conversation message edits will be accepted within the next few days. Due to time constraints, we were unable to update our pull request for unsupported browser notifications.

Collaborative Culture and Team Interactions

The collaboration process and culture definitely helped our development efforts. The open community, friendly nature, and responsiveness of the developers created an environment where we felt save and encouraged to ask questions and reach out. Tillman even helped us do effort estimation, explaining that certain tasks were going to take longer than we had originally anticipated. He helped us scope these tasks down, and continually gave us feedback along the way.

Useful Feedback

There are various different factors that must be taken into account when contributing to an existing project. These include code style, the framework being used, scalability, etc. Feedback from project maintainers consistently reminded us that these are important factors that we must take into account. For example, as mentioned previously, Tilmann mentioned that a `PATCH` request would be a more ReSTful solution with the Django REST framework than a `POST` request. In addition, Tilmann also mentioned things like “code bloat”, and other non-functional requirements that must be taken into account. This feedback reminded us that contributing to open-source software or any code base is unlike building software just to test on an autograder. In other words, getting the right output is only part of the solution; the solution also includes using good coding practices (like adding comments), design patterns, and writing tests.

What Works?

It was evident that developers really thought about code design and good design patterns when implementing the framework of the system. Their use of design patterns, such as the observer pattern, are well-implemented and align with the system functionality.

What Does Not Work? What Would We Change?

We found that the developers spent more time implementing useless functionality: such as enabling users to add emoji reactions to conservation messages, rather than adding more crucial functionality, such as being able to delete or edit conversation messages. We believe that better requirements elicitation and prioritization of requirements, would have resulted in a more organized development approach. Along those lines, we also believe that there was a better way to implement the UI. Unfortunately, the UI is what the user sees and interacts with. In order to portray a modern and advanced software, the UI must embody that. While we understand the importance of functionality before form, it is important that users are not deterred from a great product by the awful UI (one plate of food can be more delicious than other, but if the delicious one looks awful, no one is going to want to eat it). The UI is not intuitive, and includes a lot of page reloading, making it lag very often.

Personal Opinions

Julia: I have never contributed to an open-source project before, and was originally quite nervous to do so. However, the immediate developer feedback, helpful `README` instructions, and detailed documentation, made the initial stages of the development process extremely manageable. Having several software engineering internships prior to this course, I found the process of contributing to this open source repository almost identical to the process of contributing to my team's code base. It would have been extremely beneficial (and probably impressive) to have had prior knowledge of how to submit a pull request prior to my internships. In terms of comparison to other homeworks, I believe this homework highlighted the importance of not only sound coding practices (such as proper use of design patterns and code comments), but also the importance of planning, documentation, and selecting the proper tools, environment, and languages for a given project. In other words, submitting code to an autograder is completely different than participating in code review. In this case, the importance of metrics, such as code readability, become more prevalent.

Peter: I've contributed to open-source projects before, but never with pull requests that add or modify actual functionality. Admittedly, I've always been a bit scared of changing functionality out of fear of doing something wrong. As a result, I normally stick to where I feel most comfortable: creating and editing documentation for clarity. Having worked software engineering internships a few years ago before moving over to program management, this project made me realize how much I've learned in the time since: understanding code quality expectations and the build process for this project were much easier than ever before. This may have been because of the project's modern stack--with technologies like Vue, Python, Docker, and ESLint--were easier to deal with than the stack that I've worked with previously. Other homework assignments for this class that forced us to read through large codebases were helpful in developing the skill of ignoring irrelevant files and functions to focus on what matters.

X. Advice for Future Students

Julia: My one sentence of advice: Use the course staff as a resource! This class provides you with the knowledge to sit through that first meeting and have an idea about what all that tech jargon means. Coincidently, last summer I kept a list of all of the vocabulary words that were used in meetings that I did not understand. The first word on my list: triage. On the first day of class, one of the first things Professor Weimer asked was: how many of you know what triaging is? The real-world can be scary, and its important to take advantage of this class and the course staff by not being afraid to reach out and ask those questions that you may think are "dumb". (I believe this class should be mandatory because it prepares you for industry more than the required professionalism course: EECS 496).

Peter: The exams in this class will be different than those that you take in most other EECS classes. Study with a group of three to four other people and talk about the big ideas behind each lecture. Try and create a latticework of concepts: understand how they all connect and interact with each other, and you'll do well. Take this class seriously: the concepts you learn will be applicable to future jobs in software engineering or product management. I found myself breathing a sigh of relief after every lecture because I walked away knowing something I was too scared to ask colleagues about during internships.

Feel free to use this as an example in future classes. Thanks for a great semester!