Apperture Labs (Team 2) Final Report

Team 2: Jared Buchanan, Daniel Koohmarey, Zhongdao Ren, Ted Schelble, Mary Anne Smart, Fan Wu, & Li Yang

Project repo: https://github.com/tschelbs18/210habit

Introductions

Our team chose the name “Apperture Laboratories,” which is a play on “Aperture Laboratories” from the popular Portal video game series as well as theme that our team develops apps. Our color theme for our brand and slides was predominantly a light blue and white, captured by our logo:

Our team members’ roles were as follows. **Jared Buchanan** was in charge of our team’s CI pipeline and played the role of our database expert. **Daniel Koohmarey** wrote a significant portion of the JavaScript for our project and handled the ZingGrid habit log page integration. **Zhongdao Ren** was responsible for the application’s login and user management functionality. **Ted Schelble** handled the ZingChart calendar visualization integration, assisted with the CI pipeline, and served as team spokesperson and Scrum Master. **Mary Anne Smart** worked on the application’s frontend, and thus wrote a significant portion of the project’s HTML and CSS. **Fan Wu** handled the REST API endpoint behavior for our application. **Li Yang** was in charge of the end-to-end testing of our application using Selenium.

Project

For our project, we chose to build a habit tracker application. This application is intended to help users stay motivated and track their progress as they seek to establish new daily habits. For example, suppose that Jane has set a goal for herself to try to run a mile everyday. The app allows her to log each day that she accomplishes this goal, and it will keep track of her current “streak”--the number of days in a row that she has completed her goal; the desire to keep her streak going can help motivate Jane to continue meeting her daily goal. We felt that this application was simple enough in scope yet interesting enough to make a good class project for undergraduate students. We also felt that for future projects it would be easy to maintain the core functionality but rebrand the app as a chore tracker, fitness tracker, etc.

One of our first steps was to research existing applications in this space. Some apps with similar themes in the activity/habit tracking space include habitify.me, everyday.app,

stridesapp.com, and doneapp.com. This initial research helped us reason about what the core features of our app should be, identifying common features found among them. We also met with Professor Powell to discuss his requirements for the project. Based on our research and our conversation with Professor Powell, we collectively brainstormed an initial list of requirements and a set of user stories. To formulate our user stories, we considered approaching the app from the perspective of end users that would want to track habits - be it a fitness trainer or a generic user. These stories helped reflect the requirements of our app; users would want a way to log and track their habits, as well as visualize the progress so far.

Next, we made some initial sketches for our app using Miro. After completing several iterations and seeking feedback from Professor Powell on our proposal document, we settled on our final wireframes, a component diagram, and an entity relationship diagram to guide our development. The wireframes started as low-fidelity "fat marker" sketches, and we gradually iterated on those original designs and added detail to the original sketches. Only after completing these design documents did we begin discussions about how to divide the work to develop our application; the entity relationship diagram was particularly helpful for thinking about how the different pieces of our project would fit together and how it would make sense to divide the work.

Process

Daily Stand-ups

Given the asynchronous nature of the course and project, we opted to do meetings 4x per week over Zoom instead of a daily standup 7x per week. This included 3 shorter (~15-30 minute) meetings after every class as a touchpoint for team assignments, deliverables, pain points, progress, and new developments. We also used them as an opportunity to review PRs (pull requests) as a team. On Sunday nights (which was the only time that worked for all team members and their respective time zones), we had a longer meeting (~1-2 hours) to tackle live group work on important topics and items that necessitated immediate consensus. This longer meeting served as an informal retrospective, given the relevant discussions for that week in class. It also served as an informal touchpoint for sprint planning as well as iteration reviews for what was accomplished during the week. We tried out StatusHero with our team's Slack channel and used it more in the early days of the course. However, we ultimately found it to not be as useful as the organic communication occurring in the Slack channel or the daily meetings. We ultimately found the Zoom meeting structure to be effective for meeting all project deliverables and collaborating in a turbulent environment of COVID.

Sprint Planning and Estimation

In our design phase, we wrote out specific, complete user and developer stories for the different pieces of our app. We then assigned points to each story using planning poker and assigned who would be working on which story. The points were not tied to a unit of time and represented an arbitrary relative estimate of time/difficulty. Each team member made point

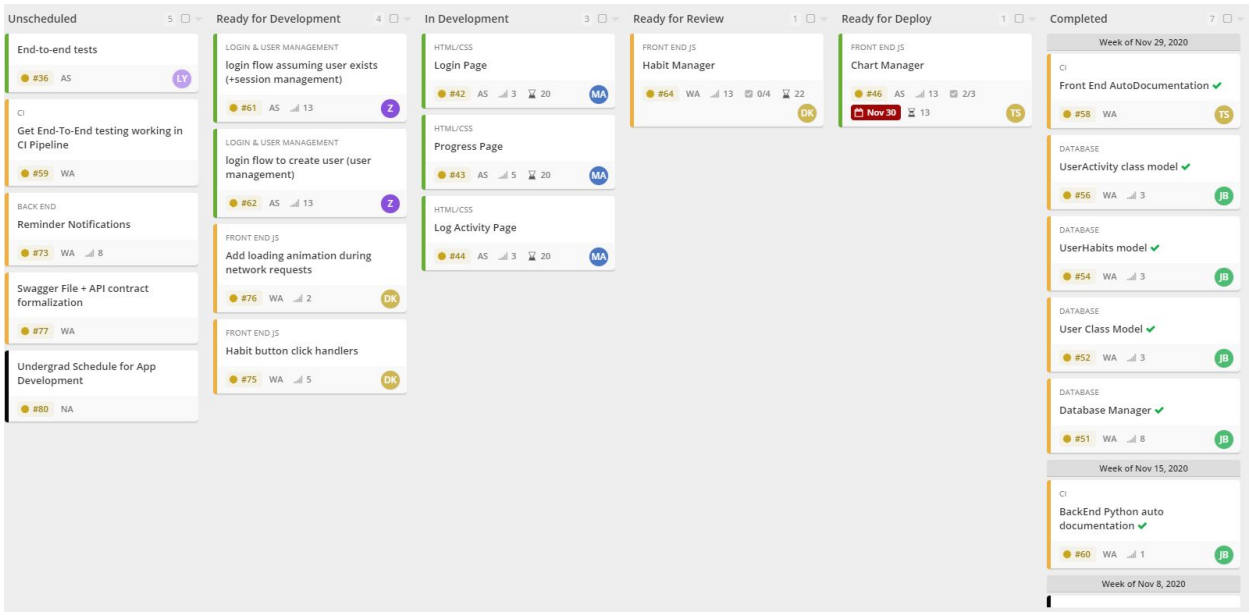
estimates of each story and estimates are then presented and discussed if there was disagreement. The highest and lowest estimates were also explained with greater care in order to evaluate outlying opinions and their rationale. This process was repeated until consensus was achieved. For us this yielded:

Sprint Structure

1. Database (21.7)
    a. Story: User class model (3.4)
    b. Story: UserHabits class model (3.1)
    c. Story: UserActivity class model (2.9)
    d. Story: DatabaseManager class (12.3)
2. End-to-End testing (19.9)
    a. Each story will test different end-to-end components
3. REST Endpoint Manager + swagger file (15)
4. Login modular
    a. Story for login flow to create user (user management)
    b. Story for login flow assuming user exists (+session management)
5. front-end js (17.6)
    a. Chart Manager (progress.js) Story (14.6)
    b. Habit Manager (habit.js) Story (13)
6. HTML/CSS (12.5)
    a. Progress page Story (5)
    b. Log Activity Story (3.9)
    c. Login page Story (3.6)
7. CI (18)
    a. Front end autodoc (5)
    b. Back end autodoc (5)
    c. End-to-end testing integration (8)

Backlog and User Story Management

At the outset, we leveraged the findings of our planning poker exercise to put stories into ClubHouse.



However, we found Clubhouse to generally be an unnecessary tool given the functionality and way we were using GitHub - comments, tags, file uploads, etc.

During weekly meetings after class and on Sunday evenings, we decided what stories we would tackle and discussed the requirements and functionality that each user story required. Once agreed upon, these requirements are added to the story. We also broke down complex user stories into smaller stories. Team members would also mention any pain points or blockers and if they needed assistance on their current assignments. Additionally at meetings we would review and discuss any existing PRs that needed to be re-worked or could be merged.
We dubbed a story “done” when the code met all requirements and passed all test cases (each story had tests of its own to be run). If there were any bugs or bugs were discovered later, a new story and requirement are added to the backlog. In those cases, we mostly handled everything through GitHub and Slack, and did not make additional documentation in Clubhouse for bugs.

Retrospectives

Retrospectives occurred informally, largely during our Sunday evening meetings when we had time to reflect upon recent changes. During Sunday meetings where we had complete attendance and more time to discuss, we would go over everyone’s progress and current state of their assignments. Discussing what had occurred in the previous week allowed us to identify areas for improvement or that needed reassessing. A few times, we pulled team members off of a certain story in order to help another team member, or connected two people together who were working on similar stories; for example, we realized that Jared and Zhongdao should collaborate to handle database calls related to user management and login functionality. Additionally, given the relevance of readings in the course and our project, we often made decisions or pivoted based on ideas discussed in lectures. For example, the security lecture

caused us to revisit how we were handling password storage, and utilize server-side validation of user inputs (we were previously planning on client side constraints for HTML forms).

Execution

Repository

The git repository contains the following (simplified) directory structure:

```
.
├── docs
├── sandbox
├── src
├── static
│   ├── img
│   └── js
├── templates
└── test
```

Our folder structure is guided by the common project layout for flask applications. We separated out the python backend server and the frontend html + vanilla javascript into distinct folders. The backend server is in *src*, html files are in *templates*, and javascript files are in *static/js*. Also, the static visual assets such as logos, images, ect are in *static/img*. We store our auto generated documentation and the associated configurations necessary to run this generation process in *docs*. *sandbox* is the location for any non-production/demo code that we can tinker with and share with the group. *test* contains the unit, integration, and end-to-end tests that are run by the CI. Lastly, the base directory contains high level application info such as package dependencies, README, and the server entry point.

We originally based our branching strategy off Gitflow with some modifications due to the smaller scope of our project. Since we never planned to deploy our code, we surmised that having both a main and dev branch would be unnecessary so we choose a branching model with a single main branch that serves both purposes. Our commit strategy was fairly simple as a result of this decision. Either a feature or hotfix branch would be forked from and merged back into main. Our rules for the code were primarily dictated by the CI pipeline checks (linting, docstrings, ect) and the commonsense placement of code into the appropriate directory. We did have an additional requirement that all new feature/bugfix branches have associated test(s) that exercise the new code, however, we ended up being fairly lax on this policy when we were struggling to get multiple large PRs merged in a timely manner. Instead, we ended up adding many tests after all of our major features had been merged successfully.

Testing

We ran back-end python test cases with the pytest framework.
We ran front-end js test cases with the Jest framework.

We ran end-to-end test cases with the selenium framework.

Unit Testing (Backend):

Unit testing on the backend was performed on multiple levels. First, lower level classes/functions such as our database manager and miscellaneous utilities were tested directly. Next, the REST API exposed by our backend application was tested via HTTP requests/responses. Since the lower level components were tested two-fold once via the direct tests and once via the REST API tests we were able to more accurately pinpoint bugs (e.g. was a bug due to some internal issue in the database manager or how our app was calling the database manager?). There were 3 major areas of backend unit tests: REST API endpoints, user session management, and database management.

The REST API tests send HTTP requests to all exposed endpoints and validated against the expected response data and status code. This included both negative and positive test cases. More specifically, the tests exercised the CRUD operations available to each endpoint (e.g. add/delete/get users, habits, and activities) and validated against the expected response.

User session management was provided through a Flask extension package, which implemented a session-id based authentication scheme under the hood. Testing here was focused on user login and logout which created and destroyed a user session respectively. Additionally, testing focused on ensuring endpoints, the associated data, as well as certain screens were only available to authenticated users. Redirects back to the login screen were expected if a user attempted to operate on an endpoint without a user session.

Database management was facilitated through an ORM package called SQL-Alchemy. This allowed for high application portability as we could switch out the particular database with ease. The ORM also provided a convenient python interface for all database operations as opposed to writing raw SQL statements. We created a DatabaseManager class that encapsulated these ORM operations and implemented unit tests to exercise the public methods. Positive validation took the following form: 1.) mutate the database state 2.) get the new state and ensure it matches expected. Conversely, negative validation took the following form: 1.) try to mutate the database in an unintended way 2.) validate that we get an error result and that we do not crash. These methods were tested both directly and indirectly via the REST API test cases. All tests involving the database utilized an in-memory SQLite database where a fresh instance would spin up for each test.

No object mocking was necessary at any stage of our backend unit tests. The app had no external dependencies or untestable internal components. Overall, this significantly increased our development velocity.

Unit Testing (Frontend):

Having verified the initial js functionality with hand tests, we attempted to integrate Jest tests to automate some type of js unit tests. Due to our js not being written with testing in mind, we found it hard to use our code with Jest or to mock our main habit manager class. As a result, we did create Jest tests, but they were more informally created by copying code from our original js files and testing them inline. We are aware this is not best practice for performing the tests, but given the constraints of our file structure and experience opted to take this approach over no tests. As we highlight in our retrospective, this is one of the key elements we would have done differently given what we know now, with improved unit tests and coverage.

Hand Testing:

Initially for the front end JS, we focused on performing hand tests to verify that the behavior worked as intended, as we did not have experience with unit testing JS. We used a combination of checking all intended use cases by loading our js on the page in a browser and verifying the page behavior matched the intention (i.e when the create habit button is clicked, a new habit is added to zinggrid. When a habit is logged, the log button is disabled and the streak increments by one). We also took advantage of console.log statements to verify data in our classes was processed as expected.

End-to-End Testing:

We run all end-to-end test cases with the Selenium framework for python. We test two main pages for possible user behaviors including click, type, and change url. We use selenium to simulate user actions(html event and JavaScript action) in headless mode and multiprocessing to run our server.

For Login Page: Users sign up/log in with wrong format email(username) or too short password should not be accepted. We test if our page shows hints or asserts when the user submits an invalid/duplicate email and password. And we check the correct username and password and successfully sign up and login and users cannot login and see asserts when users log in with invalid username/password. Also, we test that users can easily switch between signup and login subpages.

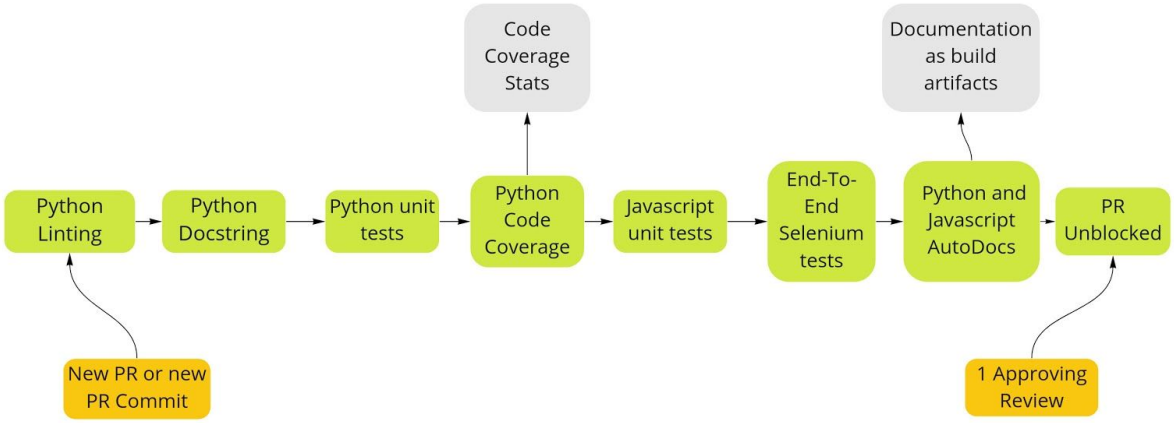
For the Habit page: Users create/log/delete habits and verify the correct result. We test the page would remain the same after creating a duplicate habit name and be scrollable after creating many habit names. Also, we test that the page shows the correct habit lists and streak after a log/delete habit is processed. We also check that the result is valid after refresh/reopen(relog in). Lastly, we test that users can log out successfully and we confirm that the page will redirect to the login page if users are not logged in.

Python Code Coverage:

Python code coverage was determined using the coverage module in conjunction with pytest. No code coverage stats were compiled for front-end javascript.

Name	Stmts	Miss	Cover
app.py	145	26	82%
src/db_manager.py	95	24	75%
src/db_models.py	24	0	100%
src/utils.py	40	2	95%
TOTAL	301	49	84%

CI / CD Pipeline Overview



The diagram above captures our current CI pipeline implemented in github actions. We started with a barebones pipeline before we actually had any code on either the front or backend. We also came in with some concrete ideas for CI steps that we would add once the project matured (e.g. auto documentation). The barebones pipeline consisted of just the docstring checker, linting, the python unit test call even though we had no unit tests, and the basic pipeline triggers/github integration. We incrementally added to the pipeline as the project progressed. Steps such as the end-to-end tests could not be added until relatively late in the project due to its reliance on MVP quality software. Lastly, we added the python code coverage

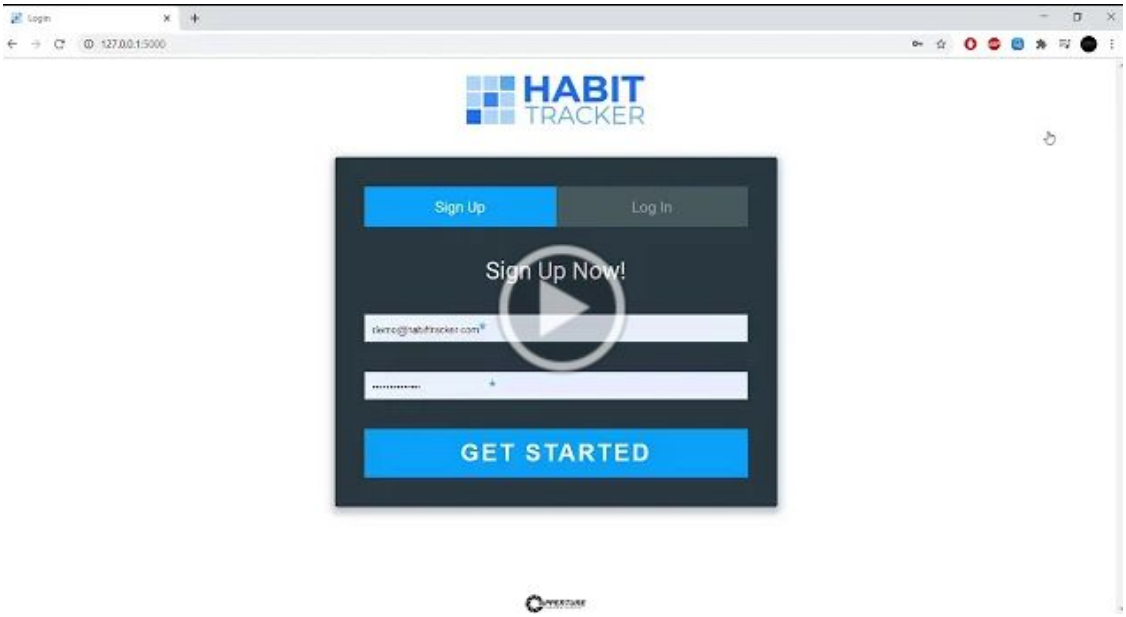
step, which had not been on our radar, at the end of the project after some feedback from the professor. Overall, this incremental approach to the CI pipeline development served us well and pipeline issues that did arise were never a major hindrance to development velocity.

Documentation

High level documentation on how to install, test, run the web application and access it in the browser is provided in the GitHub repo's README.md file. The REST API endpoints are also documented at a high level in this file. We ideally wanted to document the REST API using a swagger file, but due to time constraints were not able to complete this from our backlog. We note in the time machine section of the document how this lack of formal and clear REST API endpoint documentation created minor integration problems between front and backend. Internal documentation with regards to research, requirements, user stories, design and specification was maintained in a single google document, with separate sections dedicated to each. For code based documentation, we focused on leveraging auto generated code solutions that relied on docstrings inline in the code. For our Python Backend, we added docstrings to functions/classes so that documentation could be auto generated with Sphinx. Sphinx generates html pages for interactive/traversable or LateX pdf documentation by parsing the docstrings in the python source code. For our front end, we also added string comments that would be processed by JSDocs into documentation. We note since we are using auto generated documentation for our code, the "value" of the documentation is limited to the docstrings/comments in the code to begin with. We note that in formal "developer" documentation if we were planning on handing this off, we may have also provided a few "example usages" of code in addition to simply restating the function signatures. For our project, we tried to integrate a github action to our yaml file that generates the pdf documentation of our python and html documentation of js source code and uploads it to our documentation of the repository, but faced issues. We also tried accessing the generated action output from the artifacts section of the build action, but also saw issues with the artifact document contents appearing empty relative to generating the code locally. As a result, having finalized the codebase we manually generated the documentation and uploaded it to our repo's docs folder.

Product Demonstration

Demo video:



Why Your Project?

Readability: We use flake8 to check our python code format. Also, we have automated document generation. We add docstrings and JSDocs comments to functions/classes so that documentation could be auto generated with and JSDocs.

Modularity: We separate the backend server and the frontend into different folders and split different functions into different files. There are different files that control the database, login, JavaScript, API, testing, and user interface. Our app is also agnostic to the specific database used.

Easy to extend/reuse logic: We separate different kinds of functions into associated files so developers can easily find and use functions they need. If developers want to use our api, they can take code controlled by the app.py file and modify only the user interface to rebrand it to a fitness tracker. Also, our Pytest test cases are compatible with many other Python test frameworks.

Easy to maintain: Auto generated documentation, automated testing, and robust CI pipeline help developers to maintain their code. Having an automated build, a continuous integration system, and automated tests make it simple to make changes to the code and quickly find out if you broke anything.

Architectural Simplicity: Our architecture is not swamped with interdependent components and extraneous utility code/functions that are unused by the core features of our application. This minimal/lightweight codebase is easy for a new developer to read through and enhance with minimal ramp up.

Retrospective & Analysis

Time Machine

What would we have done differently (negative things)?

1. Explored JS libraries, pros/cons:

We realized from reading jest documentation that had we written our application with react, breaking out the modules and logic appears to have been easier to unit test. We also may have been able to leverage greater existing bodies of code given the popularity of other js libraries like react, angular, etc.

2. Design JS with testing in mind:

We realized code can be in an “untestable” state if a lot of dom/network/logic is interdependent on each other. Specifically separating inline network calls from functions that make dom changes and refactoring the code to have had a separate habit service class would have made mocking & unit testing easier.

3. Write a swagger file to concretely define REST endpoint with arguments, returned values during design phase (no loose/vague guidelines, object types concrete):

Not having clear endpoint payloads defined made disjoint front/backend development harder as we could not write the correct front end network calls and handlers until the backend was finalized.

4. Serverless:

We realized the primary operations from our server were to interact with the databases, so we could have moved all of this logic directly to the client side and interacted with a database service such as Firebase.

6. More emphasis on user design / user experience outside of core functionality:

During development we were focused entirely on “making it work” and the feature functionality, but not the actual user experience when interacting with the app. We should have allocated some more time for this user experience in planning. Luckily, we had time at the end to polish our UI.

7. Add more time to account for writing tests when considering features:

When initially doing planning poker and estimating, we bunched together a feature with its corresponding test, assuming the test implementation was trivial in comparison to the feature. This assumption did not always hold in practice as configuring good tests proved trickier than expected.

8. Use project management tools (i. Clubhouse) more often: we only pushed our progress on github, but that couldn’t let us know our percentage of completion. After defining initial tasks in Clubhouse, usage of the service stopped. We should have incorporated updating it into our weekly meetings.

9. Don’t underestimate the time required to do something “right” with tests + documentation outside of implementation:

When initially drafting requirements and things we wanted to do, we underestimated how much time following a proper process would add that led us to cut some of our original features.

What should we keep doing (positive things)?

1. Regular weekly meeting structure of one longer weekly meeting + 3x per week stand up + slack (basically, regular communication):

Everyone was active and vocal about blockers and progress which allowed us to move positively over the coding-heavy final few weeks.

2. Frequent PRs/commits:

By frequently integrating changes into our main branch, we avoided large scale merge conflicts and made integrating for the most part conflict free and easy to merge.

3. CI pipeline active early in development:

By having a working CI pipeline early, we maintained the integrity of the branch and integrity of the codebase when development ramped up and commits were frequent.

4. Bugs were quickly addressed:

If a bug was observed during testing/development from any group member, the issue was immediately resolved by the feature owner, preventing any blocking.

5. Division of work based on separate feature themes, classes/files:

We had smooth integration and assignment of pieces of independent work that allowed for the remote development of multiple features concurrently. This was enabled by assigning work separated by feature themes, classes and files and worked out well.

6. Keep using online collaboration tools: slack, miro, github, zoom.

Remote collaboration as a team was achievable through the use of the appreciate tools, all of which allowed us to collaborate synchronously or asynchronously as needed.

110 Adoption Discussion

We believe our project is a good candidate for adoption in the undergraduate software engineering course. That being said, some modifications might make the project more appropriate for the undergraduate level. One option would be to remove the user management aspect, since this was one of the significant challenges of the project; perhaps we could provide the undergraduates with skeleton code for user management and have them write the rest. It could also be helpful to provide students with our API endpoint documentation or with our pytests to give them additional guidance. We note that it would also be possible to modify our project to solely use JavaScript (rather than JavaScript and Python) or to use some existing JavaScript framework.

At the very least, we think that students should be given a high level set of requirements and desired features, as well as a roadmap with suggested timelines. Although we could give

the students our design documents, this would deprive them the opportunity of practicing the design process. We suggest allowing students to write their own design documents and then providing them with our documents afterward. They could then compare their docs with ours and make any necessary changes or updates to their designs. Finally, we think that deployment could be an interesting addition to the project. The students could be asked to set up a continuous deployment pipeline.

We think that our project is a good candidate for several reasons. First, our application presents several opportunities for learning about best security practices. For example, students will need to learn about how to safely handle user input. If the students are implementing user management, they will also need to understand best practices for handling passwords. Second, our application is easy to break up into smaller pieces that can be divided among group members. There is a clear separation between the backend and the frontend, between the various backend classes, etc. We also have a proposed undergraduate schedule of the course of the quarter that could inform structure.

Miscellaneous

Our team's top 3 of readings we did this quarter are summarized below:

1. Web Tech Foundations CSE lecture slides

   From the context of learning SE by building a web application, this was a must have "foundations" to get everyone on the same team at least familiar with the terminology and components often encountered when building an app.
2. Common security exploits, Martin Fowler's Basics of Web Application Security

   By being aware of the most common "offensive" vulnerabilities on web applications, it is easier to have a defensive mindset and due 'standard' security hardening steps such as server side input validation, sql statement preparation ect.
3. CI/CD

   Undergrads without much internship/industry experience will likely never have heard of the concepts of CI, so it's critical they read about it and its intended goals so that they can implement their own CI pipeline for their project correctly with an understanding of why they are doing it.