8. Evaluation

Overall, we feel that we have delivered a scalable base website for our client; it is very easy to add new Application Program Interface (API) features or modify existing ones (though in our opinion this is unnecessary) and experienced React-JS developers should have little difficulties modifying and creating webpages.

Our back-end API has been refactored several times so that the overall structure is easy to understand at a glance, and we have written a number of python scripts to automate certain processes, such as database migrations, and scalable load tests. There is also an extensive test suite (written in Pytest) that is easy to extend for future Test-Driven Development (TDD).

What we could do better is document our code, which we are currently working on.

That being said, our JavaScript front-end is a bit less structured. This was the first time any member has worked with React, so even with a couple of refactors, there is still a bit to improve on. However, it should be noted that React-JS is currently undergoing a transition period with the introduction of React-Hooks (hooks), there are some parts of the codebase which use hooks, although the majority is written with more traditional React Components (components). Within the industry, many React developers are refactoring their code in the style of react hooks, but like we mentioned it is a new development that is consequently nowhere near as well documented as components.

Nevertheless, for someone who has experience coding React, our frontend should not be difficult to understand, especially if we consult future developers maintaining our code.

8.1. Tests

API testing was fairly straightforward, Python makes it easy for us to do so, however simulating user input/output (I/O) on the website is not quite as easy. We decided that since the frontend is static and relies on the backend for data, as long as it was ‘working’ on a surface level it was fine – this is effectively the consensus in the web-dev community. It was more important to test that API calls gave correct responses, so we initially wrote extensive Python tests.

However, towards the final checkpoint our client kept on changing the MVP and increasing the list of features which we needed to add. We needed to shift our resources from the backend to react, and this meant we needed to compromise on our testing and code coverage (coverage) – which is a measure of how many lines of code are visited during our test suite’s execution, given as a percentage. Originally, we had planned to maintain coverage of at least 80%, but as of 7 January 2020 we have a coverage of 68% (note that the file manage.py should not be included in this figure since it is a script that does not affect API calls).

It is important to note that although testing is important, much of our API code is very similar, so after we had experience writing the backend, we were very confident that our code would work based on what the response was meant to be, and previous tests.

8.2. Load Testing

To test the performance of our website, we wrote a custom python script which uses a library called Locust.io. For many of the reasons covered in section 8.1, we chose to only test our API calls because that will be an accurate reflection of the real response time on the main website.

Locust relies on two parameters, the number of users (a simulated user is a locust) to simulate (n), and the “hatch rate” (h) which is how many users are spawned per second. By “spawning” a user we are simply creating a user, then logging in – so a hatch rate of 10 simulates 10 people signing up and logging in at the same time each second for a continuous period of time.

Our load test was fairly simple, each user will just get the project list every few seconds. Once the test is over, they log out and the user is deleted. This tests effectively simulates users going on the website and clicking the Project List tab every 5-9 seconds over and over again.

We understand that this kind of dummy behaviour is not an accurate reflection of a real user’s time on the site, its meant as a sort of stress test to measure the capacity of the site in an extreme situation. Moreover, it doesn’t matter which API call we have the locusts make because we are trying to test the response time of AWS and the Heroku servers.

For our load testing, varied the number of locusts (at 5, 10, 20, 50, 100) and the hatch rate (1, 2, 5)

Each test was running for a couple of seconds.

Our first test was at n=5, h=1, where we see normal behaviour – i.e. no failures.

At n=20, h=2, there are no issues during the run, but there are some problems in deleting the user, which probably arise from killing the program.

At n=20, h=5, we see some issues with creating users – it seems the hatch rate is too high for our AWS hosting solution to create users in time.

It is important to note that our website is being hosted on a student Heroku account (which is also hosting our Redis server), and file system is stored on a student AWS account – which have a very limited capacity and are certainly not meant to be used for a legitimate company (or charity in this case). As shown in our Figure 2. diagram, each line is a potential bottleneck – excluding Google Analytics (GA) which means that the current website is unable to handle a large number of concurrent users, however by changing our hosting solutions we will be able to fix this.

Now

Live Updates

To implement live updates for projects and comments, there were effectively two ways to do this – either with Web-Sockets or Server-Sent Events (SSE). We chose to use SSE because we wouldn’t need to change and define our own API (as would be required by Web-Sockets), and SSE doesn’t require a live connection to be sustained. We understand that this platform may be used in remote regions of the world, and the overhead of Web-Sockets is too great.

SSE involves the API sending packets to frontend clients whenever there has been an update in the database (such as another user uploading a project). Our backend knows when this update occurs through the use of the Pub/Sub model with our Redis server. Whenever a project/comment is updated/posted/deleted, this data gets published on our Redis server to a data channel. All other instances of the backend are subscribed to these channels, so whenever a change is published, all the other subscribers know about it. When a subscriber is notified of a change, the API sends relevant data to active users so that the information they see on their screens updates dynamically.