Hyposoft Evolution 2 Reflection

Group 8: Carter Gay, Inchan Hwang, Amber Johnson, and Brian Nieves

4 March 2020

Design retrospective

Frontend

Using Redux+React without typescript worked as we expected, but there were some difficulties, caused mostly by inexperience with developing with redux and lack of documentation.

As we anticipated, when designing the overall system with a new paradigm, it was harder to foresee what the best practice would be. We had to go through quite a few iterations before we grasped how redux should be used in our application. Things were worse since the frontend and backend teams weren't quite clear cut on which part of the data model we should return as simple rows and which to denormalize.

The lack of documentation was an issue as well - there were some non-trivial abstractions and generators that added some mental overhead to even the one who wrote it. Further, the reusability/extensibility of the components were degraded since the props/states weren't clearly outlined.

Going forward, the frontend team agreed to incrementally document the code we write and to minimize unnecessarily concise but cryptic abstractions and favor repetition instead.

Bulk Import/Export

For Evolution 1, we leveraged an existing library, django-import-export, to meet the requirements for bulk creation of Models and Assets. This package provided a stunning diff post-import and integrated with the Django admin page seamlessly. Nevertheless, this crutch proved to be problematic in Evolution 2 due to the library

being designed to serve one model per import/export resource. Therefore, numerous methods needed to be overridden to handle the bulk import/export spec presented by the committee. In hindsight, it would have been prudent to adapt the source code of the library to integrate with the frontend. The deviation of the overall aesthetic when venturing to the admin page is quite displeasing. Looking forward, it seems inevitable that we will need to migrate the import/export interface to the main page of our site.

Backend

Our choice to use Django REST Framework (DRF) continued to be a good choice of framework in Evolution 2, but bad habits using it in Evolution 1 made for extra difficulties in this evolution. Specifically, in Evolution 1 the simplicity of the requirements allowed us to use DRF's generic views without hardly any modification, accepting and returning JSON representations of the exact models in the database. In this evolution, the data representations required by the frontend were not exactly in alignment with their database representation, and providing the database representations (in JSON form) as we did before led to complications and inconsistencies in the frontend that we have still not yet recovered from. It also resulted in a deviation from our dividing point between frontend and backend development since the frontend engineer modified the backend code to fit the frontend needs better, causing further confusion. These are all problems that stem from both a misguided assumption about our framework and a lack of communication that we will address moving forward.

Design evaluation

Frontend

The current frontend architecture consists of three big pieces. Redux to store/inject the relevant global states to components and mutate it in a disciplined way, API client to interact with the backend, and React to build the view.

Having a separate API client makes it easy to adapt to backend changes and concentrate the complexity of client-side normalization/denormalization, dealing with lack of transactions on updates. As a result, the rest of program is less prone to backend changes and can be assured that they don't have to deal with complex interactions with the backend. In this evolution, however, the backend-facing logic became sufficiently complex enough that it was hard to keep track of whether a

certain business logic was dealt from the frontend or the backend, and it caused problems for the bulk import, since it was basically a second frontend. Also, it reached a point where the asset update consisted of multiple calls to ~four different APIs, depending on which fields were changed or not.

Redux worked well for us when the state tree was clearly defined, but as we ported the pagination/filtering/sorting logic to the backend, it suddenly broke the assumptions that other components were making. Hence, its effectiveness and usefulness somewhat degraded, as each component had to fetch the data that they needed anyways. There wasn't any caching logic done in the reducers, so it was basically the same as just calling the APIs directly from the components themselves. So, although it gives us some potential performance gains and a better view into what the relevant view states are, it wasn't as effective as we thought it would be when we started using it.

Bulk Import/Export

Currently, this feature gets the job done and features highly reusable code across the various import/export interfaces. The diff is helpful for identifying whether correct behavior has occurred after an import. For export, filters built-in to the Django admin page can be taken advantage of. Overall, the debugging process is enhanced by integrating this feature into the admin page.

Signal-like processes were encapsulated within bulk import/export that were dependent on column data (i.e., network_port_label_n). Due to a lack of communication, these handlings of the creation of Network Port Labels and Powered connections upon the creation of a Model and Asset respectively quarreled similar actions in the frontend. Additionally, the diff presented upon import does not display altered fields not contained within a model. For example, bulk import of Models created the needed Network Port Labels; however, the new labels themselves were not displayed in the diff. Unexpected error messages have also been frustrating to deal with since the rest of the backend uses Django REST framework error messages rather than Django error messages.

Backend

In this evolution the backend grew into one of the Monoliths we talked about in the beginning of the semester. It made sense to put Models, Assets, and Racks into a single application called "equipment" in Evolution 1 but adding Network Ports, PDUs, Connections, etc. made the logic between components hard to follow (since

all definitions were all in one huge file) and made it hard to compartmentalize different features of the application. What we learned is that while DRF provides a good starting point for code organization, it can't keep everything clean for us forever. Going forward we must clean this up and refactor our monolith as we add even more logical components to our application.

Testing

This evolution, we planned on creating well-documented use cases similar to the TA evaluation to test our program. While we spent a substantial amount of time testing our software in the final days, we did not follow any documented procedure and used the 'eye test' to determine if a requirement was met. This approach unveiled numerous bugs, but it is likely we would've discovered more if we took the time to design appropriate coverage tests.

Our current deployment design allows for automated testing, which was an intentional choice that we made in the beginning. However, we never took advantage of this feature and still need to commit more time to writing tests for our logic to avoid having too many features to debug at the end of the evolution after the code freeze.

Logistical analysis

For Evolution 2, our primary logistical strengths came from issue organization through GitLab and routine, twice-weekly meetings. Our primary weaknesses arose from code loss during merges and the lack of a systematic test plan.

Strengths

Our biggest strength this evolution as compared to the last one was to create a schedule at the very beginning of the evolution that laid out due dates for each feature along a timeline and assigned each of them to someone. We mostly stuck to the schedule did end up implementing new features into the first day or two of our intended code freeze. We caught a lot of bugs during our code freeze, but fixing them took longer than expected and didn't allow us enough time to find and fix the next wave of bugs. This was better, however, than not really doing any integration testing as in the first evolution.

We knew from Evolution 1 that we wanted to expand our use of issue boards and general issue tracking. Using GitLab issues, for Evolution 2 we created boards based

on Milestones (Evolution) and labels for Requirements, Frontend features, Backend features, and later Bugs. During our weekly meetings on Mondays and Wednesdays we would review the Requirements and close any issues we could. For Evolution 3, we've cleaned up our label systems and are working towards adding time estimates and time spent for each issue. We will also add relevant definitions to our issues in the descriptions so as to avoid unfounded assumptions about a requirement. Because of these systems, we generally have not had any problems with misunderstandings of who is doing what or not being able to work on code because other dependent components haven't been implemented yet.

Areas for improvement

One area that caused confusion and general inefficiency was code loss, likely due to conflicts during large merge requests. As expressed by other groups, there were several instances where fixes and updates were lost and needed to be re-added simply because branches were out of date. Additionally, even if merge conflicts were successfully resolved without code loss, this process still required significant time and attention. For this reason, we started to migrate away from each individual's development environment and git repository hosted on a VM (with continual file syncing) to working locally. For Evolution 3, we will all have local development environments. Further, for Evolution 3 we have deleted existing branches and each new feature will branch from dev.

Improving branch structure will also help alleviate these issues. In Evolution 2 we individually created feature branches off of our dev branch that relied on each other and merged them into each other and then dev, creating a complex web of code that sometimes generated needless conflicts. Instead we are going to create a single feature branch that will contain its own dependent code (both frontend and backend) and have multiple people working on the same feature use the same branch. This will also have the effect of keeping relevant code between multiple group members in the same place and avoiding surprises down the line.

Additionally, our group could have had a systematic test plan with specified steps. Not only would this better replicate the usability testing conducted during our group's evaluation, but it would help create a better user experience. Having a system test plan in place helps catch bugs and helps ensure all feature requirements have been met. Our testing phase for Evolution 2, although longer than Evolution 1, still left much to be desired. Specifically, we will need to double check each other's implementations to confirm system acceptance of a feature to all the relevant views.

Contributions

Carter Gay (cag65)

- Extended bulk import/export to meet Evolution 2 requirements
- Wrote several signals to perform the necessary logic when saving Assets and Models to create default Network Port Labels and Powered connections
- Constructed test files for bulk import/export
- Customized the admin page to aid the debugging process for the backend
- Contributed revisions to several model objects

Inchan Hwang (ih33)

- Fixed / improved whatever that we had from ev1, according to the feedback that we received during the ev1 evaluation.
- Extended the frontend to meet the ev2 requirements, other than the ones that Amber worked on.
- Tested, debugged and fixed several bugs on the backend APIs.
- Implemented some easy-to-consume APIs for the frontend.

Amber Johnson (ajj18)

- Created and organized GitLab issues and issue boards for requirements and feature deadlines
- Actively contributed to Bulk Format Committee decisions and created sample and additional test files for bulk import
- Implemented logging and log filtering in the frontend
- Added tooltips to all views based on user permissions
- Added minor frontend features as needed/requested

Brian Nieves (ban20)

- Created the new models in the backend
- Implemented requests to Networx PDU website on server
- Implemented logging in the backend
- Implemented Network graph serialization in the backend
- Added validation and additional processing for all models to fit requirements and maintain data consistency
- Implemented pagination and sorting in the backend