SOEN 390: Software Engineering Team Design Project
Winter 2021

Postmortem

Submitted to Dr. Yann-Gaël Guéhéneuc

Team 9

Concordia University

April 20, 2021

Table of Contents

One ERP to Rule Them All

Introduction

This postmortem is about an ERP (enterprise resource planning) software that our team of 8 people created which was aimed towards the bike industry. An ERP is a piece of software that can help you manage important parts of business operations such as manufacturing and customer sales. In our case, the ERP we created managed the orders to manufacture and produce bikes while keeping track of the inventory of all the produced bikes and of all the parts needed to produce said bikes. In addition, it also kept track of the financials by handling the accounting (e.g. revenue/expenses information) and by storing customer orders. Finally, the ERP also allowed the creation of accounts (administrators and employees) where the account type dictated the amount of control the user had over all the ERP functionalities.

We really expected this project to be very challenging because we knew that an ERP software can be very complex due to its nature of having to deal with many areas of a business. We did indeed find this project challenging, but we believe we rose to the task and created a project that we are proud of. All the functionalities that were set out by the product owner, in the beginning, were accomplished and we also managed to deliver a polished-looking interface in the process. In the end, the professional look and feel of our ERP in conjunction with the fact that we managed to host our software online for anybody to access solidifies the fact that we successfully delivered a working product according to the wanted specifications.

In terms of the architecture of the ERP, as you may have guessed when hosting was previously mentioned, we settled on creating a website since it would be a lot more accessible as opposed to a mobile application (not everyone is comfortable with mobile applications) or a desktop application (requires the download and installation of the software). The website's backend followed the REST API conventions which meant accepting/sending JSON objects, responding using standard HTTP codes, exposing various endpoints for external users, etc. The backend was also designed with the model + service + controller (MVC) philosophy in mind because one of the main advantages of it was how well it separates the data, the processing logic, and the routes. For the website's frontend, we decided to use the SPA (single-page application) design model which meant that we had one single web page that dynamically changed based on the user's inputs (versus an MPA or multiple-page application where whole new pages need to be sent by the server and loaded when the user interacts with the website). Creating a SPA forced us to make our code more compact and reusable and this also allowed us to compile our frontend website statically to host it online (which could not have been done with an MPA).

In terms of the tools used in the ERP, as previously mentioned, we hosted it for it to be accessible from anywhere. In order to achieve this, we hosted the frontend client on Firebase, the backend server on Heroku, and the database on AWS (Amazon Web Services). This was all

done for free! For the frontend, we used the React Framework and for the backend, we used the Express Framework. In both cases, the code was written in TypeScript. For the database, we used MySQL as our RDMS (relational database management system). As for our CI/CD pipeline (continuous integration / continuous delivery), we used Github Actions. We started out using TravisCI, but we decided to switch midway to Github Actions. The first reason for this was that TravisCI offered fewer free credits as opposed to Github Actions. The second reason being that people can write their own Github Actions which made it so that we had the choice of various different actions that people implemented themselves (similar to 3rd party libraries) whereas there were no such “community” actions with TravisCI. For example, if we wanted to make sure that our branches followed a certain naming convention, someone already wrote that action with Github Actions whereas we would’ve had to write it ourselves if we had continued using TravisCI since no such plugin existed. Finally, for our containerization needs, we used Docker and this is because, before we hosted our ERP, we needed a way for the product owner to run and test our software without having to download, install, and configure all the dependencies that came with the project. Docker allowed anyone to simply clone our Github repo and run the project in a container (provided they had Docker installed themselves). In the end, we decided to use all these tools because some or most of us were familiar with them and so this translated into a lower chance of things going wrong.

What went wrong

1 - Time management during the demos

During the demos, we had multiple features to present. In fact, it was required for us to not only demo every feature with their respective code and tests but also present our documentation and have every member of the team talking. As such, it was hard to fit everything under 10 minutes and we sometimes went overboard. This impacted us significantly since we lost many points in our sprint assessments and we also had fewer opportunities to answer questions. As such, it was sometimes hard to receive feedback from the teacher’s assistant. This made it hard for future sprints to improve and also explain our reasoning during the demo.

We took many actions in order to ensure that we would meet the time requirements for the subsequent demos. First, we limited the number of people that would share their screen. This made it so that we wasted less time transitioning from one part to another. Also, we further planned our demo which allowed us to have a better understanding of what needed to be prioritized and what didn’t (e.g. something less important like documentation could be presented quickly towards the end while the backlog needed to be covered first).

There isn’t a lot that could have been done to improve even more. Considering that we had only 10 minutes to present more than 10 things, it was hard to fit within the time. Something

that we could have done more was to have only a single person presenting while the others are talking. It takes a bit more preparation but it could help us fit within the time.

2 - Biting more than you can chew

During the brainstorming sessions, a lot of ideas were proposed but not all of them were implemented. The scope of some ideas was too big, so they had to be cut down due to tight deadlines to deliver on time.

One of these features that we had to cut down was the Home page that was present from the beginning but had always been empty. So towards the end of the last sprint, we decided to remove it since we were running out of time for sprint 4.

Other features like adding machines and having them complete orders in real-time were also scrapped because of the same reasons mentioned earlier.

This point could have been improved upon if we planned the amount of work required for the different tasks better. That way, we would have known exactly what to keep and what to discard from the backlog before actually developing the features.

3 - Starting the work late, had to push features to future sprints

One issue we encountered during the project is time management for actual development. In sprint 1, some members of the team started their work a few days before the end of the sprint and had issues with their code. This caused some features, that were supposed to be done, to be pushed to the next sprint and that caused a delay in the whole roadmap of the project.

To address this issue, we discussed with all team members the importance of starting the work early and the impact of being late, where features that are pushed to the next sprints may lead to having too much work in the future. We also decided to meet every week to assess the progress of each individual and to enquire about difficulties and how we could help fix them. This improved our communication and motivated late workers to start early and get help if needed. There isn't much we could have done better, we handled the situation pretty quickly and this ensured that no other sprints encountered delays.

4 - Pull requests and code review participation

After completing the development of a feature, in order to merge the new code changes to main, team members would create a pull request so that it can be reviewed. However, the pull requests were often checked rapidly without actually running the code. This happened because team members were busy with their various tasks and did not always have time to run the code changes proposed by each pull request. Also, team members would often take time to

start reviewing pull requests. Pull requests were sometimes left open for a whole week until they were reviewed and then successfully merged to main. The impact was that it contributed to extra stress as new features were sometimes merged late into the sprint. In fact, if the merged pull request resulted in a bug then we sometimes did not have enough time left or had to rush to fix the problem by the end of the sprint. To fix this issue, team members would be notified immediately when a pull request was posted on Github so that they become aware and can perform a review as soon as possible. To do better, we believe that team members could have presented a short demo of their code changes during meetings in order to make sure that the others obtain a visual representation of the proposed changes and provide immediate feedback.

Furthermore, the code reviews often lacked participation as it was always only two or three team members who would actively review and provide constructive feedback. This happened because team members felt that if the code was already approved by a few members then it must be good and so there was no need for them to also review the pull request. However, this was not always the case. The impact was that it slowed down development progress because some team members were not aware of the new changes being introduced which might cause disagreements further down the line. In addition, fewer reviewers mean more chances of having undetected bugs get through to the main branch. To address this issue, we would notify every team member to go perform a code review when a new pull request was uploaded. Also, we blocked the merging of pull requests until at least two members approved the pull request. We believe that we could have done better by blocking merging until at least 4 team members (50% of members) have approved the pull request. This would have potentially reduced the number of bugs and helped ensure that new features have fewer chances of needing changes later on.

5 - Law of Triviality

The Law of Triviality, also known as bike-shedding, describes a situation where the majority of a discussion is spent arguing on relatively minor, but easy to grasp details. This project, due to its sheer size and complexity, was a prime candidate for some bike-shedding. Many of our meetings lasted way longer than anyone would’ve wanted, without us doing any substantial progress. We spent many hours arguing about relatively small details, most notably some elements in the design of our database, instead of spending our precious time together allocating tasks, and sharing our problems.

Midway through the project, after noticing that we were often participating in bike-shedding, we started to change our attitude. We started putting time limits on our meetings, to force us to get to the point of our discussions. We started trusting each other more. We stopped trying to micromanage everything. The speed at which we were able to work increased a lot in our last sprints. We should’ve had this attitude since the start.

What went right

1 - Periodic team meetings for task separation

From the start of the project, we decided to meet every week to assess all the member's progress and see what was left to do for the current sprint. This approach allowed all the team members to stay on track with their tasks and modify certain features if needed.

Meetings would be done twice a week with each meeting varying in length depending on the amount of information that needed to be discussed. Usually, team members would meet Tuesdays and Thursdays after the TA meetings. The time of the meetings would vary depending on each team member's availability so each meeting would have as many members as possible. If a team member was not available, we would debrief them with the important information afterwards.

Discord was a powerful tool to organize the meetings and to archive important files and various pieces of information needed to build the project. Bi-weekly meetings would be done on this platform and presentations about task separation would be done by having one person share their screen while everyone participates in the conversation.

The meeting schedule could have been done better if we used certain software available where each member inputs their availability. This way, finding a moment where everyone is available could have been done more easily and smoothly.

2 - Meeting recordings

The recording of each meeting on a wiki page was a crucial part of our project. Not only was it a requirement of our project, but it was also a necessity for our development process. Each wiki page had the date of when the meeting took place as the title and was summarized to contain the main discussion points of our meetings. These meetings were for both team meetings and the meetings with our teacher's assistant (TA). The meetings' conversions included brainstorms to debates and would even sometimes derail from the main subject. So having a summary of the pertinent points helped our team stay on the same page once a meeting was over which allowed us to be more productive and sure of our development processes. Thanks to this system, our team rarely had to mention the same subject twice. Furthermore, for the meetings with our TA, we made sure to note down everything that was lacking from each of our sprints to help achieve a better sprint presentation.

It would have been possible to improve the system that we implemented by adding the different options that were considered before arriving at a common decision. It would have been helpful to add the pros and cons of each option instead of just describing the outcomes of each

meeting. This would have helped further support why we chose certain techniques or why we decide to solve an issue in a certain way and not in another.

3 - Overall Architecture and code quality

One thing we are particularly proud of is our architecture and code quality. From the start of the project, we decided on a front-end colour palette and theme that we followed on all mockups and in the actual code. This ensured a consistent design for the entirety of our app. Moreover, we are proud of our code quality and code convention. We achieved a line coverage of more than 89% over hundreds of tests. The code convention is strict and we used a linter to check that the code was formatted correctly. We also added numerous Github actions to check if commits and pull requests were respecting the strict linter and if the branch and commit naming conventions were followed. All of these made the code better, easier to read and understand, and of a generally higher quality.

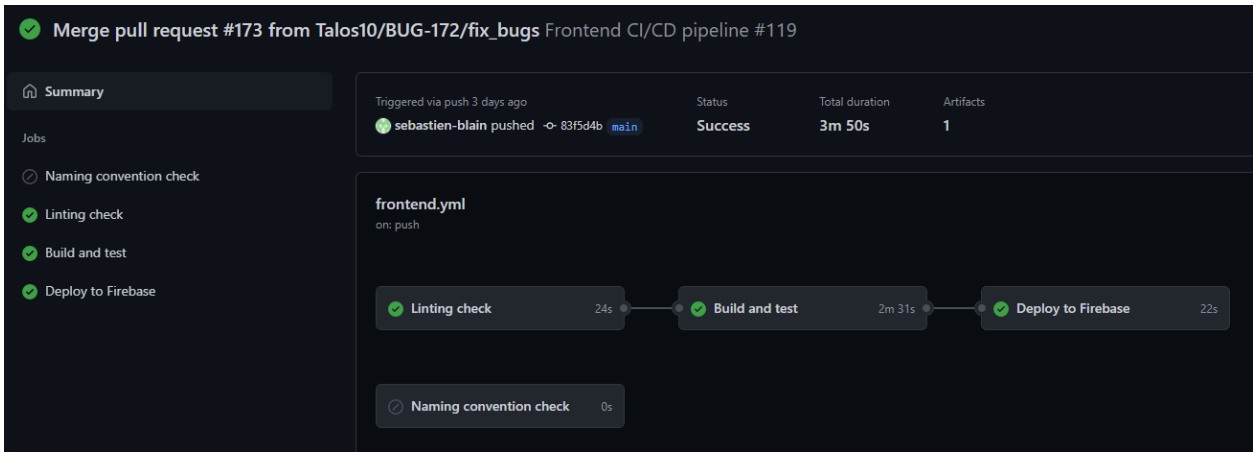

Figure 1. A successful Github action workflow that checks the linter, builds and runs tests, and deploys the app to Firebase

There isn't much we could have done better. We believe we were pretty strict with the coding rules of format, theme, colour palette and this reflects in the app. This made it so we resemble a professional coding environment that is properly structured.

4 - Mockups before coding

This project is something we all wanted to show proudly to others. We wanted our app to be fun to use and pretty to look at. As a result, we spent a lot of time studying the UX and UI of similar applications. Shopify and Firebase were two great inspirations, and their respective design guidelines, Polaris and Material, were of great use to us while building our mockups. Mockups allowed us to test designs quickly, without any concerns for the actual

implementations. After settling on the final designs of the mockups, our goal was to make our final product match the mockups in a pixel-perfect way.

Creating the mockups first and then doing the coding is great for many reasons. As previously mentioned, creating a UI with a complete disregard of how difficult it is to implement forces a developer to think outside the box when solving the issues at hand. The best example in our app that can demonstrate this is our Accounts management page.

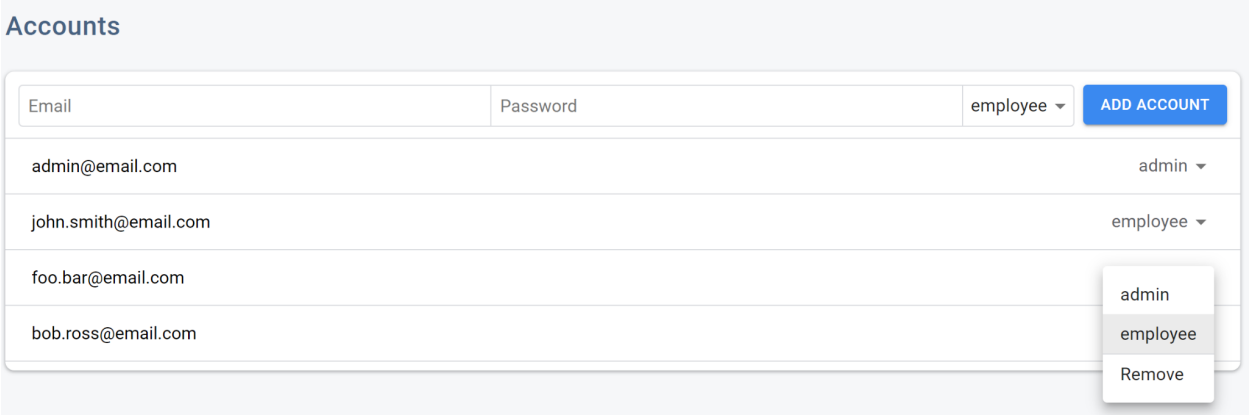

Figure 2. Accounts management page

Merging the email, password and role inputs into one big input was something that took a long time to code, but it made the process of adding a new user a lot simpler to understand, and it is much better UX than having a separate form for adding users elsewhere. Once the users have been created, instead of having a list of roles and a trash icon to delete the user, we merge those two elements into one list in order to lighten the table. Once again, this was harder to code, but we believed it improved the UX. We used a lot of similar UX improvements everywhere in our application, and we believe it paid off.

Unfortunately, as time went on, we started to spend less time on Mockups, as we focused more on coding, and some later pages seemed inconsistent. We could've done better in this regard.

5 - Hosting of the database, backend and frontend

It happened out of necessity. As we were nearing the end of sprint 3, we had many features in place and felt our application was ready to be deployed. This made it so that it was easier for users to see the current state of the app.

The impact was huge in our opinion. It was a crucial moment during the development phase that would allow end-users to play with our app without any technical knowledge. It also allowed us to find bugs that were not seen when developing locally. For example, the react framework doesn't render SCSS styling the same way locally vs when deployed.

In order to set up the hosting, we had to decide which services we would use. Not only that, but we also implemented automated deployment in our Github action pipeline to facilitate deployments. This is how we decided to address hosting for each of our services:

For the backend, we had many options but ultimately chose to host our app on Heroku. Unlike cloud services like AWS ec2 where you need to set up the infrastructure, Heroku offers the infrastructure as a service and made it so that setting up and managing our app was quick and simple. This gave us more time to focus on the features. Another advantage of going with Heroku was its free tier. Since the app wasn't used too often, the minor inconvenience of the free tier boot-up time did not impact our application.

For the frontend, we decided to go with Firebase. Since it doesn't take much to host a static web page, we had many options. We ultimately decided to go with the service that required the least amount of setting up and the most benefits. As such, we chose Firebase. First, Firebase offers a nice-looking hostname for our application which means that we didn't need to purchase a domain name. Also, it is simple to monitor the app and its various statistics on the Firebase website. Finally, Firebase also offers many features which were not used like google authentication and storage. Instead, we opted to make our own authentication system and use a MySQL database. Although we did not use these features, it is always good to have these features available if we ever decided that we needed them.

For the database, we decided to go with AWS RDS. On it, we have a MySQL server. We chose AWS because of its reliability and the fact that we fit nicely within the free tier range.

In order to improve, even more, we could have hosted everything on an infrastructure provider. It doesn't cost much but it is available at all times. Also, it would allow us to better tune our servers to only take the resources it needs. Something else we could have done to improve our performance was to use a web server like nginx to serve our application. It would be not only more scalable but give us more flexibility when it comes to serving the application. However, considering the time constraints and the amount of time required to set up using an infrastructure provider and nginx, our team did well going with quick and simple solutions.

Conclusion

We learned to communicate better as a team where we all have different opinions on how to implement some features. During our meetings, we all discussed and shared our thoughts on how we should do things. Most of the time, some members proposed different solutions to the problems we had, and we always came to one agreed conclusion. During this process, we learned to effectively weigh the pros and cons of each suggestion, and sometimes we even merged two solutions proposed by different members because both suggestions were appreciated by the team. Having these types of discussions helped all team members to understand different opinions other than their own and be open about it. Also, each team member learned some new technical skill or tried something new for the first time. It was a big

learning experience for the members because, for one feature, we may have tried more than one method. Thus, even though we wrote code that got replaced, we learned from that other method, and it's an advantage for all members.

The main takeaway the team got from working on this project is that every member can contribute meaningfully. Some people are better at writing front-end code, others are better at writing back-end code, while some members are better at writing SQL scripts. We put together all our strengths and separated the work equally between all members. This project was an opportunity for us to understand that each team member is important and that they have some skill that can be useful for the project. So, there were a few members that took more time working in the back-end while others took more time working in the front-end. Most of the members also worked on both ends. By merging our strengths, every member contributed to the project, and no one was left out of the group. Therefore, we had great synergy, and so there was no conflict.

In short, we learned to all work together as a team and to take every suggestion proposed in order to make the best out of this project. Our discussions helped us listen to others and collectively make the best decisions.