Group 3: Tripmate Final Report

Group Members

Ahmed Kidwai, Claire Chen, Connor Bean, Eddy He, Evan Nagasaka,

Kurt Hanel, Kyle Duque, Renan Desconsi Turra, Shepard Li

Introduction

If you're Graham Greene, perhaps you prefer to travel without a map and go simply where the road takes you. If you're a software engineer however, you know that coding without a plan often leads to folly. Planning software development projects is a hard task. Planning a great trip is also difficult. While we couldn't make software that eases the planning of software projects, we did create Tripmate. This report chronicles the building of Tripmate's MVP, from inception to implementation.

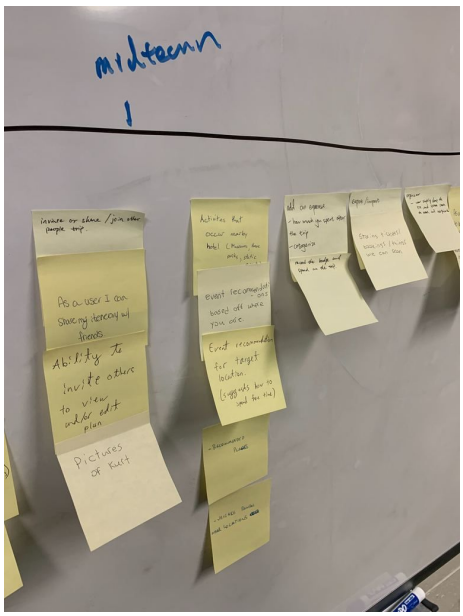
Project Planning

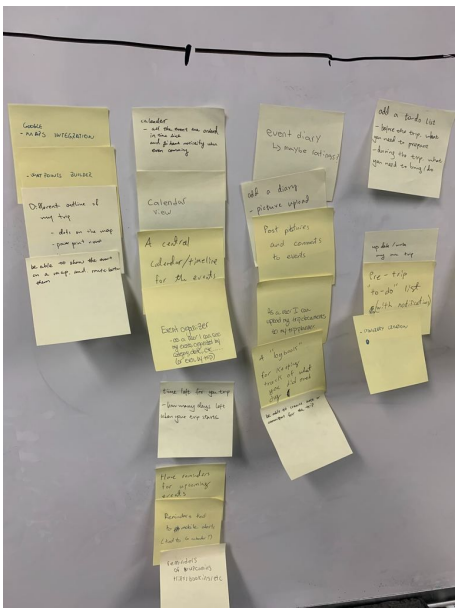
How did we come up with Tripmate and conceive the MVP?

In our first group meeting, each team member pitched multiple ideas, such as health trackers, a Patreon-like app, and a budget planner. Of course, another suggestion was

a travel planner, which had great appeal for numerous reasons. Perhaps the most influential of these was the fact that the group member who suggested this idea said that it would actually be useful to them, given that most travel planners that they had used had been insufficient. The prospect of satisfying this need was exciting. With a little brainstorming, we were also able to incorporate other app ideas, such as a budget tracker, into the app as features. To seal the deal, we put the project ideas to a vote.

In our second meeting, we took a look at competing travel planning apps and thought about what we could do better. We let our imaginations run wild. We conceived social networking features, automated trip template builders, the ability to fork other user's trips, and much more. Given that this was a four-month course, we had a small amount of time to create an MVP. This meant that the most vital and most realistic features had to be identified in order to make the MVP development possible given such time constraints. We used a technique called storyboarding, where we each wrote down ideas of what we thought the MVP would require. We took these ideas and consolidated them, taking time to discuss what the most important features were and hypothesizing what the timeline would be. This allowed us to boil the app down to its core features: itinerary management, pre-trip preparation lists, budget tracking, and the centralization of travel details, such as flight and hotel bookings, into one convenient location. While the more imaginative features certainly would have elevated the project and made it more unique, they were considered long-term goals and not crucial to the app's initial viability.





Figures 1 & 2. The above photos are from our first group meeting where we conceived of Tripmate, wrote down as many user stories as we could think of, and laid them out in a timeline from conception to MVP and beyond. These images give a glimpse into our earliest plans for the project.

What collaboration tools did we use?

In order to make sure our project was well organized, and that all team members were kept in the loop, we used a variety of communication and collaboration tools.

Slack was our central communication tool. We made a variety of channels including one for each layer of the project (e.g. a database channel) as well as a “help centre” channel, where teammates could ask for help and share educational resources. Furthermore, we configured an automated reminder to be sent out to all members

informing them of upcoming group meetings. We also integrated slack with Github and Buddy, to post notifications about updates to pull requests and deployments to our test EC2 instance. In response to the on-going pandemic, we made a Discord server with two channels for voice chat, making remote communication easier.

In terms of collaboration, Trello was crucial for making sure our project was organized. Using Trello, we were able to track which tasks were being worked on, by who, and approximately how far along in development they were. We also used Trello for a retrospective, wherein each group member anonymously filled out cards stating what was going well and what needed to be changed with regards to development and team management.

How was the work divided? Advantages/disadvantages?

Before full feature development began, we divided tasks based on the technology. Specifically, we assigned straightforward “setup” tasks for either the database, server, mobile, or web app to groups of 2-3. After these preliminary tasks were complete, we had a thin baseline to build upon. This gave everyone a chance to start familiarizing themselves with parts of the tech stack before core development began.

Moving forward, each group member chose one user story from our task list to complete. This approach had upsides and downsides. One upside was a clear separation of tasks; team members knew exactly what their assigned tasks entailed. Another was flexibility. Teammates could work at their own pace since most tasks did

not depend on each other. This was a huge boon since many team members were learning a lot of new technologies (React, Redux, MongoDB) for the first time. A major downside to this approach was siloing. Since it was uncommon for features to overlap, it became easy for the team to keep their heads down while programming. This led to the duplication of some code as features quickly progressed in isolation (e.g. Actions and Reducers in the Mobile and Web App projects).

Outside of development, the course's group activities were often handled as a team either during class or immediately after. However, some activities, such as presentations, were handled by single individuals who volunteered to do these tasks themselves.

What would you do differently, if you had the chance to start over?

If our group had the chance to start over, we'd try to establish a better way to approach design. Since we almost immediately went into distributing features among the group members to develop the MVP, we missed which features should be implemented before others. This resulted in situations where some group members were waiting on another member's feature to be implemented. If there were more brainstorming and careful consideration about what a feature is composed of, there'd be less time wasted waiting around.

Another downside to our “jump in” approach is that it caused instances where completed features needed to be refactored. This is because these completed features needed to incorporate elements from newly developed features. This further demonstrates that there wasn’t a focus on developing specific features before others. An example of this problem is how we completed many features, such as the flights, to-do lists, and hotels before trips themselves were added as a feature. It wasn’t until the last stretch of development that the “trips” feature was completed -- certainly a backwards approach to building a trip-centric app. This caused us to refactor existing, functional code which could have been avoided by adjusting our priorities earlier in development. This can be viewed as approaching the problem “bottom to top” instead of “top to bottom”.

To elaborate further on the solution to this problem, we could’ve incorporated more planned iterations across development. We believe that this would solve the problem mentioned previously because we’d have to take the time to structure the development process. By periodically evaluating what features were top priority and what should be developed before others, we could’ve avoided these issues.

Another change we would make if we were to start over is to implement a microservice architecture. After reading the best paper awards in CS on technical debt, our group understands that architecture choice has a huge impact on the overall project later in the development cycle. Therefore, we believe that a change to microservices

architecture would be the best choice when designing for scalability, creating radical new features, and implementing new technology in the future.

What went wrong in the development process?

In retrospect, our group members became narrow-sighted on implementing the specific features assigned to them without evaluating the big picture. During the development process, we should have been regularly evaluating if the state of our product was starting to resemble our MVP. By losing sight of this, near the end of development, we were left with a product that had features that weren't tied together. Even though these features were completely functional by themselves, they weren't integrated together to make a cohesive product as we had envisioned. An example of this is how we were able to complete the flight's feature where users could look up their flights and store the details in the database. However, at this point in the project, there was no way to associate the recorded flight and the actual "trip" the user was taking. So, the result was a randomly recorded flight which doesn't mean much to our users. Therefore, there was a lot of time and effort spent at the end of development pulling these features together into a cohesive package that matched our usability goals set at the beginning of the project.

Another change we would make to our development process would be to focus more on the user interface (UI). Compared to the other parts of the product implementation, creating a UI can be seen as easier in a technical aspect. However, designing a UI can

be very time consuming and striving for that perfect looking UI can be a lot of work. This touches on the idea of the ninety-ninety rule:

“The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.”

- Tom Cargill, Bell Labs

This addresses the idea of “hidden” work within a project and how it can take more time to complete something than what was expected. In addition, designing the UI makes you take a step back and evaluate how the user would go about accomplishing their tasks. This process may come to reveal “hidden” work where code needs to be changed or refactored. Therefore, we believe that If we would have chipped at this seemingly simple problem throughout development, the work near the end of the project would have been substantially reduced.

Furthermore, throughout development, we should’ve been more consistent with our planning activities. This started off as regular updates to our Trello board, but as the project progressed and deadlines approached, it was difficult to stay on top of things. This made it difficult to keep track of the progression of features and the overall state of our project. In hindsight, this may have added to why less important features were rolled out before the more important features.

Lastly, we believe that having more formal retrospectives could have helped in the workflow of our project. Throughout our project, we only had one formal retrospective, which had proven to be helpful in figuring out the strengths and weaknesses of our team. Consistently keeping up with this 10-15-minute activity during our weekly meetings could have solved some problems faster in development by allotting the time to clear up any confusion about the direction of the project.

What went right in the development process?

Throughout the development of Tripmate, our team was able to keep constant communication. The day the team was formed we set up a Slack server where we spent the remainder of the term talking and discussing the project, developer tasks, and issues. Additionally, we would meet every Wednesday afternoon and do a brief standup where each person would describe their work over the course of the past week. During our meetings, we were able to organize our user stories better and break them down into tasks for our Trello board. Everyone always seemed open to help those stuck and try to hash out solutions to blockers together, emphasizing our team's positivity and professionalism. While we did have blockers throughout the development of our MVP, for the most part, we maintained a slow and balanced momentum towards our MVP. By the end of development, we had got the primary dev tasks we wanted to be done, completed without burning out or using hacky methods to make things work.

How did the project change from your initial vision?

Surprisingly, Tripmate did not stray from our initial vision. However, we did narrow the scope of our MVP slightly; originally we intended to allow multiple users to collaborate on a trip. Having such a narrow scope helped us maintain realistic expectations of ourselves and the project as a whole. The vision most likely remained intact due to this narrow scope, and the fact that we maintain an extremely agile approach to the project throughout. That is, we did very little upfront planning, we simply moved fast and tried to react to roadblocks and design smells as quickly as they arose. In this sense, we did not place any demands on the project's identity. We let it be, and it gradually evolved as each individual did their part.

Project Implementation

What did your git workflow look like?

During our first team meeting, we settled on having one master branch that would be our production branch and one development branch that we would use as a staging environment. Each team member would branch off development for their development tasks. Upon finishing the implementation of their task, the assignee of the ticket would rebase and squash their commits before creating a merge request. We decided to rebase because we wanted to ensure that merge conflicts would be resolved before a

branch was placed back into development. We chose to squash commits because we wanted to have the commits of our primary branches (master and development) concisely represent the *key* additions from feature and hotfix branches. This way, the working trees of our primary branches are not cluttered with insignificant commits that would otherwise have appeared from merging a feature branch (e.g. commits like “removed excessive comments” or “add a semicolon”). Once a branch was rebased and squashed group members would create pull requests on Github. We set up a “code owners” file that would automatically add every group member as a reviewer once a pull request was made. This made it much quicker for reviews to be done because every group member would be emailed to review the new pull request. We also integrated Github into our Slack server. Using the Github app for Slack, whenever a new pull request was made our entire “Pipeline” channel would notify every group member that a new merge request was up and if it had passed tests. Our Github also used Github Actions that allowed us to run CI testing automatically with every merge request or changes to a merge request.

What took the most time? The least? Any surprises?

Task completion often took the most time. During our group meetings, we would often discuss features and judge them as easy implementations. We would often say a user story should take 2 or 3 days to complete, however, the reality was that we often found blockers or were waiting on others to complete a piece of code we needed. This was

most evident when pulling everything together at the end of the term, we had a bunch of UI and backend APIs that required more work than simply combining to create a trip. Setting up Github Actions for continuous integration (CI) took longer than expected. Github Actions is a relatively new feature on Github with limited official documentation that is rarely the most consistent. We had to resort to online tutorials to get it up and running.

We were able to get our app infrastructure (Mongo, Express, React, React-Native, and NodeJs) up relatively quickly. There was a lot of documentation online for setting up distributed applications that use these languages and frameworks. We were able to have the infrastructure up within a day.

The most surprising part of the development was uniquely identifying trips to specific users. It took a group meeting and some chats on slack and discord to finally decide how we were going to uniquely identify trips. It led to the creation of users. Each user when they sign up or login is assigned a JWT token that stores a userID when decrypted. We decided to play the JWT token in local storage and the app would decrypt the token and take the ID and place it in a new trip. When viewing a trip we'd only show trips that match the user's unique userID. This took many days to implement and we relied a lot on online tutorials.

Would we recommend others use a similar tech stack?

For our purposes, the tech stack we chose was perfect and we recommend other teams with similar needs use it. We wanted to develop features quickly, filling in the details as we went along. To enable this, we needed flexible technology that is well supported.

Database-wise, MongoDB and MongoDB Atlas were well-suited for this task. With MongoDB Atlas, each team member could easily make their remote database and modify entries online. What's more, since MongoDB is a non-relational database, it's incredibly easy to change the schema. This allowed us to easily change the structure of our database since the schema does not have to be explicitly defined and rigidly followed like in a relational database.

Furthermore, by choosing to use Express JS for our backend, and React JS for our front-end, we only needed to focus on one programming language. This was great for those who hadn't worked with JavaScript that much either, as it presented a variety of learning opportunities that could translate across the project. Furthermore, using JavaScript frameworks allowed us to take advantage of Node and the Node Package Manager, which allowed us to customize the test and dependencies for the server and front-end clients separately. It also granted easy access to helpful packages such as Mongoose and Redux.

Additionally, by using React JS alongside React Native, we were able to share many React components and Redux actions/reducers across projects, greatly reducing the amount of time needed to do front-end. What's more, since these two libraries are virtually the same, knowledge and skill transfer between front-end projects was nearly one-to-one.

How did we integrate completed developer tasks?

Our GitHub workflow ensured a really easy integration of peoples' work. Each pull request needed at least 3 approvals, otherwise, the merge was blocked. Having at least 3 sets of eyes on a pull request helped ensure that tasks were up to snuff, and made it easier to find smells. Making it mandatory for any three teammates to review one's pull request helped make sure that no progress with the project went unchecked. That is, a level of transparency was maintained throughout the project such that at least four people were aware of active developments. Plus, pull requests would be blocked if any tests or linting failed. Each sub-project had its own CI tests to verify that new code in one project did not break the others.

At the tail end of the project, code integration hit a bump in the road, becoming slightly more complicated for a set of final tasks. In order to reach a final working stage with the project, an overhaul was needed in order to connect the various pieces of the app under the umbrella of a single trip. This task required special attention from the entire group as it had large scale impacts on the app. These changes were made in another branch while other team members continued on with their work. The changes required to tie

everything in the app would have broken these new features, and possibly old ones if merged into our development branch. Thus, team members rebased off of the branch with the major changes while they continued working to ensure that there would be no conflicts. As discussed earlier in the report, this scenario was not ideal and could have been avoided with better planning.

The Future of Tripmate

If we kept working on the project, what would we do next?

If development on Tripmate were to continue, there is a set of immediate goals as well as some stretch goals that would help carve out a market niche. The immediate tasks are more conventional for trip planning apps but would help improve the overall usability of the app. For instance, calendar integration would help make events and travel times visible at a glance, and notifications would remind the user of incomplete to-do lists and upcoming events. In case some bookings are made or confirmed on paper, it would be great to have the option of linking pictures or pdfs with the bookings in the app. This would help round out Tripmate's ability to centralize all trip information.

We would also like to add event recommendations to the app through the use of the EventBrite external API, similar to our use of external APIs for finding available hotels. Support for finding and booking rental vehicles would also be a great addition. Since internet access is not always a guarantee when travelling, another task would be adding

a local database to the mobile app that can be synced periodically with the “master” remote one.

In terms of stretch goals, we would love to expand Tripmate to include a variety of networking features. The first of these would be the ability to add other users to your trip. This would allow families, as an example, to plan a trip together with a shared itinerary and set of to-do lists. Following this, we would add wider community features, such as the ability to share, fork, and rate trip templates made by other users -- we’d take steps to make sure users cannot broadcast the specifics of their trips. This would be a great way for users to find low budget trip ideas or inspiration for grand adventures. We could also create our own proprietary services for automatically generating new trip templates as starting points for users based on their previous travels.

Of course, as we dive into all of these additional features, we would like to refactor Tripmate to use a microservice architecture. Following the Law of Holes, going forward, all new features would be implemented as a microservice. Furthermore, old features would be refactored gradually, most likely one per each release. This way, our app can remain agile and scalable as it grows, which is vital since we clearly have a ton of ideas for the future of Tripmate.

Lessons Learned

What did you learn about team or large project development?

Throughout this project, we learned the importance of many aspects of large project development such as continuous deployment, continuous integration, interactive rebasing, formal planning activities (eg. Trello), linters, merge/pull requests and communication. Using Github as version control allowed us to create pull requests, where 3 other group members had to approve first before merging to the development branch. This was an effective strategy in only merging working code and keeping other group members in the loop about your progress.

Also, our group was able to implement continuous integration and continuous development. So, when regularly occurring pull requests were merged with the development branch, automated testing, linting and deployment occurred. This saved our group a tremendous amount of time for two reasons. First, since there was automated testing and linting, you were able to quickly determine if the code to be merged was functional and meeting coding conventions. This made reviewing code on Github much easier because you knew the code was tested and readable. Second, since there was automated deployment, we were able to continuously have an updated EC2 instance with the state of our application. This eliminated the worry of having

trouble getting a working version of our application deployed. In addition, this saved so much time as there were more than 80 automated deployments of our application.

Lastly, using Trello allowed our group to remain on task and get a high-level view of the state of our application. On Trello, we decided to have a list of tickets that reflected what needed to be completed for the MVP. Then, group members would be able to continuously pull from this list of tickets so that we were continuously making progress. With each ticket, you'd be able to see who was working on it and what stage this feature was at in development. This allowed anyone to look at our Trello board and see a snapshot of what was happening in the development of our application. Moreover, interactive rebasing was a workflow decision that further increased the readability of our development branch. This is when you rebase your branch onto development and squash all of your commits into one. The result is a fluid, clean, reversible git history that further emphasizes the readability of our progress.

Conclusion

Upon successfully completing the MVP for Tripmate, our team is left with a stronger understanding of not only numerous new technologies but also what it takes to harness these technologies effectively as a large team to build a robust baseline that has the potential to scale into a giant. From this team development process, we learned many lessons and each has our own personal takeaways, ranging from architecture considerations to team management practices, and we will employ these learnings in

future large-scale projects. Lastly, implementing and deploying a product from start to finish using real-world technologies and workflows has served as a benchmark of what to expect in future software development endeavours.