AstraTutor

Team 4 Group Lessons Learned

Our Goal

Our goal was to create an all-in-one solution for an online tutoring platform. The current world pandemic has put remote learning at the forefront of the lives of students. We set out with the simple goal of making a platform that would facilitate the following functionalities:

- Students quickly and easily finding a tutor.
- Tutors being able to list subjects they want to provide lessons for.
- Students and Tutors being able to schedule a lesson at a time that suits them both.
- Students being able to pay for their lesson online.
- Tutors being able to retrieve their outstanding payouts.
- An online classroom with video sharing, screen sharing, whiteboard, chatroom and pdf sharing functionalities.
- A review system allowing students to review their tutors.

After the initial 6 weeks we achieved our goal and managed to implement all the core features we wanted, however one of the things that isnt polished or complete is our rescheduling functionality, the goal was to allow a lesson to be rescheduled to an alternative time and while the functionality is partially there, it has some serious problems where its very difficult to follow the lesson state, whether it's currently being rescheduled or not and who is asking for the reschedule, on top of that once the lesson reschedule request gets accepted the student would need to pay again as it was an oversight on our part, this all pointed towards some of our poor planning and management issues addressed elsewhere in this document.

Failure In Planning

A key issue we feel we failed on was initial planning, our initial week was very productive and we spent a lot of time planning exactly what we wanted to achieve with our project in a larger scheme, but we never spent some time planning some key aspects of our project such as the architecture and deciding how we would implement our program and what the execution flow would be like, we had made the decision on the tools and frameworks we would use but not how we would use them; this resulted in the flow of our backend api being very hard to follow and lacking consistency, this resulted in slowed development and a codebase that isn't maintainable or scalable, this issue isn't isolated to just our backend but also the react frontend, just not to the same extent, the main problem with our frontend is improper use of the framework which relates to our inexperience with React. Our development was very ad hoc and isolated where we would meet at the start of the week, decide what we wanted to get done and individually go away and do those things on our own, and this was great for later on in the project as we got a lot of progress done very quickly but early on in the project it caused a lot of the foundation of our project to lack consistency, and since these decisions were made at the start of the project, by the time they had become a problem they had already been ingrained into our codebase and were very difficult to change without major rewrites which we didn't have time for. We also hadn't chosen a project manager until near the end of our project and by then it was a little too late.

Success In Project Management

One of the planning decisions that we feel was a great success was our use of kanban, code reviews and branches, from the start of our project we set out with using what is known as the GitHub flow, this is a development pattern that allowed us to safely, quickly and easily work independently on a feature and then through the use of code reviews being able to get input from the rest of the team. This was combined with the use of kanban boards to plan out the features we wanted done each week, adding them as kanban issues and then assigning members to them, this way we could always refer back to the kanban to see what has been completed and what still needed to be completed, as well as what other members of the team where working on.

Value of Communication

During the beginning of our development of this project in the initial weeks we started off rather slow due to a lack of set meeting times. However, it was clear to see that as time progressed during the weeks after we had set out at least one or two consistent meeting times per week (apart from the weekly scrum) that development as a whole began to speed up on the project. However, at times it should be noted that some of our meetings could get delayed due to timing issues etc, and while this did not affect us too much it was clear that if this project was over a far larger time frame that this could lead to a lot of lost time overall as these small bits of missed time added up. One other important aspect of communication that should be noted is the communication during development. We began to realise the benefits of a team when it came to sharing resources to aid in other members' development and even helping solve problems in development when they arose.

Time Allocation

One of the more important things we learned during the course of this project is that time allocation is a difficult thing to get precise. Even as we became more organised after the initial weeks as we started organising our sprints via Kanban, we learned that even though we could get a rough estimate for how long something “should” take this did not usually take into account problems arising during development. Due to this, development of some features took longer than we expected and at times left us reaching the end of our sprints with multiple items left that we wanted to add as an addition onto our project. If we had considered this during the beginning of our development as a group we could have focused on more important issues that we may have left too late into our development process and finished just in time rather than finishing these parts in a more reasonable time to allow for more thorough testing of our product.

Unexpected Events

It should be noted that during development of the project that unexpected events that contributed to progress being slowed did occur. One such event was a failure of technology, this did not actually lead to the loss of code as the member had been fortunately working on a remote development server, however this did leave that member of the group unable to continue progress until another development setup was created. Even in the base level of time allocation we failed to consider that unexpected events such as these can occur during the development process. It also occurred to us that because we had not specifically planned and assumed that these events could happen that we were quite lucky that considerable progress was not lost. Due to this we learned the benefits of tools such as git and remote development servers for situations such as these. More importantly we learned the importance of actively using these tools during development and not just when progress finishes in an area.

Prior Exposure

Our choices of frameworks were made because we wanted to learn skills we were unfamiliar with:

Prior to the project:

- 3 out of 5 team members had never written Go
- 3 out of 5 team members had no experience in React
- No-one on the team had experience with WebRTC
- 3 team members had experience with WebSockets
- No-one on the team had experience with the Stripe API or implementing payments
- 1 out of 5 had only minimal experience with Docker

After the project:

- The 3 new Go learners felt that they became more proficient in the language
    - Those who already knew Go felt that their existing knowledge was reinforced
- 3 team members gained experience in using React
    - 2 other team members improved and renewed their knowledge in React
- 2 team members had the opportunity to implement WebRTC for the first time & strengthen their existing WebSocket knowledge
- 1 team member had the opportunity to implement Stripe Payments for the first time
- All team members gained significant knowledge of Docker & docker-compose

Commentary to be made on **"learn-while-you-do"**:

- It is difficult to evaluate how much work is needed for a new technology. Implementing WebRTC & Stripe (the two completely new technologies) was tasked to only 1-2 individuals and as a result, other team members did not have exposure to these technologies
- Prior experience in a technology had a **significant** effect on how the project is structured. This was immediately obvious by the quality of file structure in code committed by someone with vs someone without experience in the technology
- Code smell is an issue as lack of prior experience also translates into lack of intuition when implementing new features. This can only be developed with practice

Working With Backend Frameworks

We decided to use the programming language Go for the backend primarily because of its low barrier to entry. Since the backend was going to be a REST API, we decided on using the web framework Gorilla Mux. We also decided to use postgres interfaced with Gorm for the database. This setup allowed us to quickly prototype a basic REST API which in turn allowed the frontend to be developed early on. Since Go provides all of the necessary features to write a REST API, we had little issue getting things working however Go's limitations made some aspects more difficult. For example the type system means that primitives such as integers have a zero value of 0 which means that if we used Gorm's Update function to modify a row, if we were updating a value to 0 (in the case of an integer), gorm would omit that from the update due to it being the zero value. Go's lack of generics also made following the Do Not Repeat Yourself principle difficult when implementing some endpoints that shared a lot of functionality but operated on different data types. This setup allowed us to write a functional API in a short amount of time however if we were to start this project again we might consider using a framework such as Python, pydantic and FastAPI due to great ability to prototype quickly while not being restricted to Go's type system.

Working With Frontend Frameworks

For the UI, we decided to use React + Typescript with Ant Design components mainly due to our interest in the technology. At the start of this project, most of the team had little to no experience with React however everyone managed to start creating views and components pretty quickly due to the excellent documentation provided by the frameworks. We decided to write all of our React components as functional components since it's generally considered the way forward for modern React and wanted to learn how to write them. This proved to be a small issue however since we over complicated some of our views by having too much state and side effects in single components, making them difficult to read, this was all part of the learning process. If we were to rewrite the UI we would compartmentalise functionality into subcomponents in a more effective way.

Learning WebRTC

From the beginning we knew that video calling and screen sharing would not be an easy task. This is something usually accomplished by much larger teams of people with more time and resources. Despite the existence of open source solutions such as BigBlueButton we decided to implement the necessary functionality ourselves using

WebRTC as doing it from scratch seemed like more of a fun challenge than simply using BigBlueButton. We were able to put together a simple prototype using WebRTC within the first few weeks. After that though is where we began to truly understand the weight of doing it from scratch. The WebRTC code based on the "Perfect Negotiation Example" from the W3C is timing sensitive and can be rather temperamental. Once the WebRTC code had been integrated into the UI a number of timing related issues were discovered. We did our best to fix these with numerous checks to get the code to a more stable state. Even now there are rare times when it fails unexpectedly. All this took more time than expected and if we were to start from the beginning we would definitely reconsider using BigBlueButton. As well as the WebRTC code in order to support the greatest number of users we deployed the coturn TURN server on Netsoc Cloud. TURN provides a relay for users who cannot establish direct peer to peer connections. Coturn proved troublesome to configure which also cost us precious time.

Dockerizing Everything

Docker is a simply incredible tool which has revolutionised the software industry. Using Docker to containerize the different parts of the project such as the API and the UI meant that the containers could be run on any host without having to worry about system dependencies. Furthermore by using Docker Compose we vastly simplified the process of getting the code running. The Docker Compose config file defines all the services necessary to run the project. It automatically builds the API and UI Docker images for us. As well as that it downloads and runs a Postgres database. After cloning the code the entire project can be started with a single command on any system running Docker. This sped up the time necessary to create a local instance for development as well as deploying the code in production. Docker was definitely the best tool for the job and we would certainly use it again.

Taking Payments

Implementing payments was quite a challenge. We made use of the Stripe API and due to its public API interface, we were forced to make a number of unforseen design changes. We originally intended on charging a users card at the time the lesson began after placing a hold when they originally requested a lesson. This would happen automatically without their consent. We soon discovered that the maximum time a bank will allow a hold to be placed on a card is 7 days, so a student was forced to book a lesson within 7 days of payment. Placing a hold allows you to guarantee the funds are available. We could have avoided this by not placing holds, but then a student could book a lesson and then have their card denied at the time the lesson was due to start (leaving the tutor penniless).

We soon learned about recent EU law: Strong Customer Authentication which requires consent (with multi-factor auth) for card purchases. We realised that for card purchases that happen automatically on a saved card, there is a higher failure/deny rate by the bank. This is because the bank will deny charges on cards where consent was given a long time in the past due to anti-fraud concerns.
We chose to redesign the lesson request system to implement a 'Payment Required' stage so that users had to pay by card explicitly and could consent via 3D Secure (as required by the SCA law) for both new and saved cards..

Stripe's frontend library can handle the card number entry and 3D secure popup. By using Stripe, full card data is not stored on our servers (to comply with PCI). Stripe's card number element makes requests directly to Stripes API where they hold the card information. When the user wants to see their saved cards, we only receive the last 4 digits, name and expiry from Stripes API on our backend to pass to the user to see. We never see their full card number.

Implementing Stripe was difficult because of the number of choices given. Stripe offers multiple tiers of implementation that balance ease of implementation vs long term maintenance. Stripe Checkout was first used (a page that claimed to handle all payment processing for a particular product easily), then ditched because it was simply a fancy looking (but lacking in function) form that didn't save card numbers. We opted to use the custom payment flow instead.

Stripe Connect Express was used to give our tutors Stripe accounts that we managed. What surprised us was evolving compliance law: we need to actively check every time the user visits their billing profile that if they have met requirements to receive payouts. I.e a user can suddenly be asked to supply new identity information (e.g. photocopy of a passport) and we must redirect them to Stripe to supply this.

Stripe Connect Custom (which is a _full_ enterprise system where the developers have to implement all UI functionality and compliance) was not used because it is targeted at massive, mature companies who want full control over their payment ecosystem.

As a lesson learned, I would have spent more time reading the Stripe documentation on the differences between implementations before diving in with trying to implement payments immediately

Hosting

Hosting was essential to our project. Without being able to host the project online we would not have been able to test two users on different local networks connecting via WebRTC. UCC Netsoc offers the free service Netsoc Cloud. With it we were able to set up an instance to host our code. By hosting on Netsoc Cloud we were unable to run our own STUN server as STUN servers require two separate public IP addresses and Netsoc Cloud is running on a single IP address using NAT. This was not an issue though as there are a number of free to use public STUN servers. Netsoc Cloud was perfect for us to quickly and easily deploy a version of our code. However If we were to release the project however we would have to move away from Netsoc Cloud. As Netsoc Cloud uses NAT we were only able to forward a limited number of ports for use with the TURN server. Ideally the TURN server would have thousands of available ports for each relay connection.

Conclusion

Overall we are happy with the outcome of our project, we feel we achieved our main goal of creating an all-in-one online tutoring platform while also achieving our secondary goal of learning new frameworks and technologies, and doing all this while having fun. One of the unexpected learning outcomes was skills less pertaining to typical development and that was our team building, communication and time management skills that we feel greatly improved over the course of the project.