Kevin Dugas

Full Stack Final Project Journal

FilmFlam

Initial backend setup

I have a Mac with an M1 chip so I had to set up everything locally, given that the provided VM was not compatible. All setup configurations went without a hitch thanks to the detailed walkthrough during lecture. I was able to get a Fastify instance, postgres instance with a single User entity, seeding, and migration working from a docker container with little to no speedbumps.

Ingesting IMDb data

This was a massive hurdle, especially compared to how seamless initial setup was because the IMDb dataset was absolutely staggeringly massive. Their API returned about 10 million (that's 10,000,000 with that many zeros) records. Each record represents some sort of video media; a film, TV show, film short, and even a record for every TV episode. On top of that, I needed two files which had 10 million records each because the basic info for each title (name, year, genre, etc.) was in one file while the average rating for each title was in another and the two datasets were joined by an ID field.

Ultimately, I needed to join the two datasets into a single postgres table so that the backend wouldn't need to do an inner join each time there is a request involving a title. I have a moderate amount of experience working with enterprise-scale dataframes using pandas/pyspark/databricks so I found an npm package that emulates pandas to do the job. Predictably, joining two 10 million record datasets was not something I had the resources to do without a cloud computing cluster so I had to filter the datasets down enough to be usable. I eventually got there after using a combination of scripting and a typescript file that's run directly from node to reduce and format a final zipped json file from which my seeder unzips, ingests, and stores into my postgres DB automatically when instantiating the container. You can find out a more detailed account of how I did it in the project README.

The end result is that there is a table containing static data of all of IMDb's movies and TV shows from 1991 (my birth year, since we're being arbitrary) and later.

Writing statements

The next major piece of data I needed to create was the statements that comprise a review. My plan was this: write numerous statements that can be inserted around a title name that would refer to some judgment based on that title's rating. For example, say a movie got a 3.4 average rating. That movie is categorized as "terrible" and when a user selects this in the frontend, the backend will choose from a randomly selected pool of statements that are for titles that are

considered “terrible” (e.g. [“Some randomly selected statement predicate”, theTitle, “the second predicate in that randomly selected statement”]). There are also addon statements for signed in users that are structured the same, only surrounding some profile field like favorite film or favorite actor.

That meant that I had to write A BUNCH of statements. And on the one hand, I could just come up with ultra-boring “Man that movie sucked!” type statements but where’s the fun in that? I enjoyed writing them, particularly because they’re monumentally stupid.

Backend routes

I wrote CRUD routes for the User entity and, as you’d expect, they looked exactly like the ones you demoed in class. There were very little hiccups and all of the endpoints seemed to work as expected from Postman. I knew that I would need some other routes and would probably need to alter current ones a bit to account for working with titles and reviews but I decided to hold off until I had a better sense of how things are going to be setup on the frontend.

Frontend setup

Setting up the frontend was even easier than setting up the backend since most things are handled by Vite. I had to spend a bit playing around with React to remember how it worked. My capstone project was a mobile app written in React Native so the syntax and concept of hooks wasn’t completely foreign though there were some definite cob webs.

Navigation and color scheme

I spent hours trying to get the navbar to look how I want it to and a color scheme that I liked, which admittedly is not great but that’s the devil of UI. I hate doing UI because I’m equal parts impatient and a perfectionist. I really wanted to have a mini Vin Diesel head as the site icon but I’m too lazy and busy to make that happen. (I’ve for some reason decided that Vin Diesel is the butt of most jokes surrounding this site. Must be all the “family” memes out there.)

First contact: the backend meets the frontend

I made my first successful API calls, which was a great feeling. I thought I was going to be debugging that for forever but everything worked pretty quickly. With that in mind, I created a new route that takes in a title ID, does a bunch of logic to select and assemble statements into a review and then return it to the frontend. I also added a route to search for a title by ID.

Homepage is finished

The homepage works like this: first it checks if the user is signed in (which they definitely are because there’s no auth at this point). Then it searches for the title they entered and displays each title with the matching name (there’s often multiple). The user selects the title that relates

to them and the backend produces a review (which is really an array of strings) that's formatted and displayed to the user. They have the option to generate a new review for the same title or enter a new title. Eventually, the user will be able to save each review if they want.

By the way, this part was a S-T-R-U-G-G-L-E, amigo. That's because I was getting this horrific CORS looking error when I was trying to make any API call from within my homepage component. I spent hours figuring it out and I knew it wasn't actually a CORS thing because I was still able to make API calls elsewhere. Turns out, the API calls were getting b0rked because the button handler that was calling them was nested from within a form tag, whose submit functionality apparently sabotages axios. FUN.

Authentication and sign up

Well, I could delay no longer. The moment had arrived and I was standing there shaking with no shoes and holding just a wimpy little stick against the behemoth homunculi that is authentication. I'm being dramatic, though I was slightly worried about it. Luckily though, it went without a hitch! I chose to use Firebase Auth with the help of this tutorial to get me started. It works as advertised on the tin with very little config setup so that was lovely. I then refactored the Doggr context with my own and we were up and running.

I did notice, however, that the User CRUD routes searched by ID, which is a number generated by mikro-orm. There's no place to store that when the user wants to login so I decided to instead search by their firebase login UID instead. This is probably dumb and I may as well search by email but I figured I would allow for users in the nonexistent future to be able to easily change and have multiple accounts under one email.

User profiles and finishing up the site

The final piece that was missing was using authentication to hide certain routes unless signed in, such as a profile page and logging out, and hide other routes when you are signed in such as login and sign up. That was straightforward and between you, me, and the fencepost, I leaned heavily on GPT like you did during lecture (see above re: perfectionist and impatient). That made churning out a bunch of UI nonsense so much easier.

The downside is that my site looks like what the CEO of Bootstrap would probably force you to make with a gun to your head.

Moment of truth: adding the frontend to docker

This was another portion I was dreading because although the backend docker experience was seamless, I just knew something was gonna blow when I added the frontend. BUT I WAS WRONG <3. Your instructions during class were spot on and copy-pastable to the max. The whole site takes a bit to build for me (like a minute and a half to two minutes) but it seems to work. Which means I'm done! WOOT!