



UNIVERSITY OF
BIRMINGHAM

Using the Modern Technology Stack to Build a Scalable Web Application

by

Vladislav Podchufarov
Student ID: 2227032
MSc Computer Science

Supervisors: **Ian Kenny, Mark Lee**

School of Computer Science
University of Birmingham
September 2021

Table of Contents

Abstract	4
Project Idea and Objectives	5
Tech Stack review and Technical Challenges	6
Creating a Server-Side Rendered app with Next.js and React	6
Creating a Performant API with GraphQL and Apollo	7
Using Prisma to define the Data Models and provide access to the Database	8
Using TypeScript to enhance tooling and intellisense	8
Using AWS S3 for File Storage, and AWS RDS for Database Hosting	9
Scalable Deployment using Vercel and Digital Ocean	9
Using Figma to Design the Mockups	10
Functionality and UI/UX Design	11
Design Process	11
Browse Sketches	12
View Sketch	12
Auth System	13
Edit Sketch	13
Backend	14
Project Structure Overview	14
Setting up GraphQL Server with Apollo, Nexus, and Prisma	15
Designing the Database and creating it using Prisma	17
Search and Filtering	18
Ranking the Posts	19
Authentication System	20
Scalable Image Uploads	21
Frontend	23
Project Structure Overview	23
Setting up the project with Next.js and TypeScript	24
Apollo Set Up	24
Server-Side Rendering	26
How React Components Work (Modal Component Example)	27
Deployment	28
Frontend Server on Vercel	28
Backend API Server on DigitalOcean	29
PostgreSQL server with AWS RDS	30
Image Hosting with AWS S3	30
Evaluation	31
Performance	31
User Feedback	33
Conclusions	34
References	35
Appendix	40

Abstract

The aim of this project is to explore the process of full-stack web development using the modern toolset, to investigate ways of integrating the latest web development technologies in order to create a fast, interactive, and scalable online application that can support many users, and to solve the challenges that come with building a server-side-rendered React application with a GraphQL API.

To accomplish that, I wanted to design and build “Sketch Club” - an interactive and easy to use web application that helps art students to regularly practice their drawing skills. This application enables the students to share their artworks and browse the artworks created by other people. It features a token-based authentication system, a scalable image upload system integrated with AWS S3 using pre-signed urls, search and filtering functionality that takes advantage of the Prisma ORM features, and a performant GraphQL API built with Apollo.

The application was deployed using a scalable network architecture, published online, tested and evaluated by the potential users.

In the process of building the app I have learned how to use Next.js, Apollo Client and Server, GraphQL, Prisma, Nexus, AWS S3, AWS RDS, and Vercel.

Project Idea and Objectives

Sketch Club is a web application that helps art students to regularly practice their drawing skills. Every week the students receive a new drawing task - they need to create a sketch or design based on a particular subject. They can share their sketches, gain inspiration from each other, and see how the other artists are approaching a similar creative challenge. The application's purpose is to motivate people to draw more regularly and develop their skills faster.

The students are able to create an account using the authentication system, and upload their artworks using a secure and scalable image upload system. The artworks are organized by topic, enabling people to browse the sketches made on any particular subject. Each student has a portfolio page showcasing their artworks, where they can easily see the progress they're making.

The project is designed and built using modern web development tools and technologies. In recent years the advances in web technology have made it possible to create extremely scalable, efficient, interactive single-page applications, but understanding these technologies and combining them together into a complete software product presents multiple interesting challenges.

My goal was to explore the state of the art approaches to solving these challenges, deal with the issues that arise from combining several new technologies that are still in early development, and gain a valuable skill set that will be very useful when it comes to finding a job at a company that uses modern tech stack and needs people capable of building complex software using these technologies.

The application was built using:

- Next.js to achieve server-side-rendering of an interactive React app (making it more performant for the users and more discoverable by the search engines).
- GraphQL and Apollo to create a very performant API.
- Nexus to write more maintainable GraphQL code.
- Prisma to conveniently create, access, and modify the database.
- AWS S3 and pre-signed urls to achieve secure and fast image uploads.
- DigitalOcean, Vercel, and AWS RDS to achieve scalable network architecture and inexpensive hosting.
- Figma was used to efficiently create the initial mockups and design (and to explore the modern UI/UX design toolset).

In this report, you will find an overview of using each of these technologies in order to create a fully functional software product from the initial idea to launch, the challenges I have encountered during this process, the solutions I was able to find, and all the most interesting ideas I have learned in the process.

Tech Stack review and Technical Challenges

In this section, you will find a list of technologies I have used to build this application, the reasons for choosing these technologies, and the challenges I have encountered in the process.

Creating a Server-Side Rendered app with Next.js and React

React.js is a JavaScript library for building rich and interactive user interfaces [1.1]. It is extremely popular, powerful, and useful for creating interactive single-page applications with a very flexible and responsive UI.

React's main drawback is that the application is being rendered entirely in the browser - the server sends you an empty html page and a javascript file, and after the page has been loaded, javascript makes requests that fetch data from the server, and only after the requests are resolved React is able to render UI components based on this data. This results in two issues:

- While the app spends the time making requests and fetching the data, it appears unresponsive to the user. It uses the loading spinners as placeholders for the components, and is able to render the actual UI only after the data has been fetched. That results in a frustrating and inconvenient user experience.
- Because the app initially sends the empty html page, which is populated by data only at a later point, it is difficult for the bots to parse and process it. That creates significant disadvantages when it comes to search engine optimization (Google is the only search engine capable of rendering the React app in order to parse it), and creates difficulties with social media sharing (Facebook, Twitter, and Discord cards do not work with client-side rendered apps).

In order to solve these issues, we need Server-Side Rendering (SSR) [1.2]. SSR enables us to create universal web apps - the app is being rendered on the server, and sent to the browser only after it has been populated with data. That means that as soon as the app has loaded, it becomes immediately interactive (no need for the loading spinners), resulting in a snappier, more satisfying user experience. Search engines have no problem parsing SSR apps (to them they look like any other website), and don't penalize your app. Social media sharing works as intended as well.

In order to implement server-side rendering with React, I have used Next.js [1.3] - it is a framework that was designed for creating SSR apps [1.4], and it also provides a number of other features and benefits, such as folder-based routing, static page generation, pre-fetching data, and automatic code splitting. In addition to those features, it simplifies development and building processes (no need to set up webpack config), and, later on, enables simpler and more scalable hosting (see the sections below).

The biggest challenge with using Next.js arises from its main benefit - server-side rendered apps are more difficult to write, as they require dealing with data fetching on server as well as on client.

Also, it is relatively new, and requires quite a lot of exploration and research in order to integrate it with the other technologies I am using.

Creating a Performant API with GraphQL and Apollo

GraphQL is a query language that enables extremely efficient API requests and data fetching [1.5]. Unlike its predecessor (REST APIs), it enables front end web applications to fetch the exact data they need, without overfetching or underfetching - no bandwidth is being wasted on sending the data that is not getting used, and no extra requests are being made to fetch that data that wasn't sent by the API endpoint [1.6].

GraphQL itself is just a specification, in order to use it in my app I need Apollo - a library that provides GraphQL implementation for JavaScript applications [1.7]. Apollo has server-side and client-side components. Apollo Server enables me to build a GraphQL API endpoint that will serve the data to the client, and ApolloClient helps me to access this endpoint from my React app, as well as provide a number of other benefits and advantages (mainly the client-side data management and caching).

In order for Apollo Server to be able to create the GraphQL API, I needed to build a schema containing the GraphQL type definitions [1.8]. There are multiple ways of accomplishing that, but I found GraphQL Nexus to be the most convenient option which provides multiple advantages - mainly end-to-end type safety, ability to split the schema into multiple files, and to build the schema using JavaScript (instead of the GraphQL's schema definition language, which would require more boilerplate code and context switching) [1.9].

Apollo, GraphQL, and Nexus are relatively new technologies (unlike REST APIs which they are designed to replace), which introduces a lot of complexity to the web development process. There's a lack of learning resources and standard approaches when it comes to integrating these technologies together into a web development stack. Combining them with server-side-rendering and Next.js (see the previous section) provides additional challenges and requires additional research. I needed to explore the ways of combining these technologies with the other libraries in my stack, and solve the issues that came up in the process.

Using Prisma to define the Data Models and provide access to the Database

Prisma provides object-relational mapping (ORM) - it helps the developer to create models, generate database tables, and access the data without manually writing SQL queries [1.10]. It enabled me to define the schema in a convenient and human-readable format and use the automatic migration system to update my tables. As a side benefit, Prisma comes with Prisma Studio [1.11] - a GUI for viewing and editing the data in the database, which I found very useful and convenient.

Additionally, Prisma works well with PostgreSQL [1.12], which is preferable, in my case, to non-relational databases such as MongoDB. Technically, either option would have worked, but, due to the nature of the app I'm creating, I wanted to use a database designed to work with relational data [1.13].

The main challenge of using Prisma is that the technology is very new, and is under active development, which makes the research extra difficult, and requires a number of creative workarounds in order to solve certain problems that have already been solved with more established technologies (such as Mongoose). You will see one of such workarounds in the section about ranking the posts based on their score and submission date.

Using TypeScript to enhance tooling and intellisense

TypeScript is a superset of JavaScript, its main purpose is to provide static typing to JavaScript apps [1.14]. Its main advantage is that it helps the developer to catch many bugs during the development process, as opposed to noticing them at runtime. Once the programmer has defined the variable type, it can no longer be changed, and the compiler will alert them of all the type-related errors during the build process [1.16].

Another advantage of TypeScript is superior tooling - using it enables the editors (such as VSCode) to provide auto-completion and intellisense - the code becomes self-documenting and more convenient to write. It makes the code navigation easier, and highlights errors in the IDE immediately, speeding up and simplifying the development process [1.17].

The main challenge of using TypeScript is the learning curve of adopting a technology that is somewhat unfamiliar to me. Luckily, since TypeScript is a superset of JavaScript, it can be adopted gradually - it's possible to access many of its benefits while writing code that is mostly similar to regular JavaScript.

Because most of the advantages of TypeScript become more valuable and apparent on larger-scale projects built by large teams of people, and I was working on a relatively small one-person project, I did not find the strict typing particularly useful. So I simply let TypeScript infer the types automatically - that way I was able to write the code that is syntactically identical to JavaScript (which is more concise and familiar), and yet gain all the IDE-related benefits such as enhanced intellisense and documentation automatically. I did find both of those features extremely helpful.

Using AWS S3 for File Storage, and AWS RDS for Database Hosting

AWS [1.18] is the industry standard for cloud database and file storage. Since my application requires scalable image uploads, AWS S3 is the obvious choice to go with. I have learned to upload image files to S3 using the pre-signed urls, which achieves very secure, scalable, and inexpensive image uploads.

AWS also provides a database hosting service called RDS [1.19], which is perfect for storing my database - it is a well-established technology, it works great with PostgreSQL, and it's free. With all the other complexities and challenges involved in my project, it was convenient to have the database hosting taken care of by a standardized solution.

There are two significant challenges when it comes to using AWS. First one is using the platform itself - its UI is complicated and unfamiliar to me, and requires understanding of a number of new concepts (mostly related to users, roles, permissions, and setting up the security-related features with AWS buckets).

The second challenge was figuring out the image upload process, which involves a number of steps and can get quite complicated. I will go into this process in more detail in the image upload section of this report.

Scalable Deployment using Vercel and Digital Ocean

Once the application is built, it has to be deployed on the web in a way that scales to a large number of concurrent users, which presents a number of challenges.

The client side of my app is hosted using Vercel - a hosting platform designed specifically for Next.js [1.20], which automatically provides many scalability-related features, as well as Continuous Integration and Continuous Deployment, which make the deployment process more convenient, enable rapid iteration and testing, and prevent broken builds from being pushed to production.

Unfortunately, Vercel can only be used for serverless web apps, so in order to build a scalable GraphQL API I needed to set up my own backend server on Digital Ocean [1.21].

I am pretty familiar with the process of setting up my own server, but in order to improve the security and make my app work over HTTPS, it was necessary to set up SSL certificates. On top of that, a number of issues arose because my client and server had to be hosted separately - solving the CORS errors and enabling the client and server to communicate in production proved to be quite a challenge.

Using Figma to Design the Mockups

Last but not least, I needed to learn Figma for the UI/UX design and prototyping. It is an extremely powerful and convenient tool for creating designs and mockups [1.22]. While it looks simple and easy to use, there are a number of things I had to learn in order to use it effectively. Figma provides very powerful digital asset functionality [1.23] and enables its users to create very flexible design systems and prototypes. Since I'm relatively new to Figma (and web application design process in general), mastering this workflow was extremely valuable, but a somewhat challenging skill to learn.

And, of course, there are multiple challenges presented by the design process itself. I wanted to create a design that is beautiful, elegant, and minimalistic. I wanted my application to appear extremely simple and intuitive to the end user, but achieving this simplicity required some clever ideas and design solutions on my part - I needed to apply the principles of good UI and UX design to make the application easy and enjoyable to use.

Functionality and UI/UX Design

My first task in building this project was to specify the key functionality and design the mockups. To accomplish this task, I have learned Figma and its powerful asset system. Its flexibility has enabled me to build the mockups quickly, and gradually improve them until they have matched the vision of the app I've had in my mind.

My objective was to create a simple, beautiful, elegant, and minimalistic design. Since this application is meant to be used by art students, I wanted to hide all the technical complexity behind the simple and intuitive UI which even non-technical people would be able to understand and use.

Design Process

I have researched similar websites, and attempted to adopt the most useful UI/UX elements I could find.

- The color scheme is mainly inspired by Product Hunt.
- Sketch browsing and viewing functionality is inspired by Instagram.
- Sketch editing and image uploading functionality is inspired by ArtStation.
- The list of weekly challenges is inspired by ArtStation's categories.

I have written down a list of key features this application will need, and attempted to organize them in the most straightforward and intuitive way. I have designed four key screens - Browse Posts, View Post, Create/Edit post, and Login Modal.

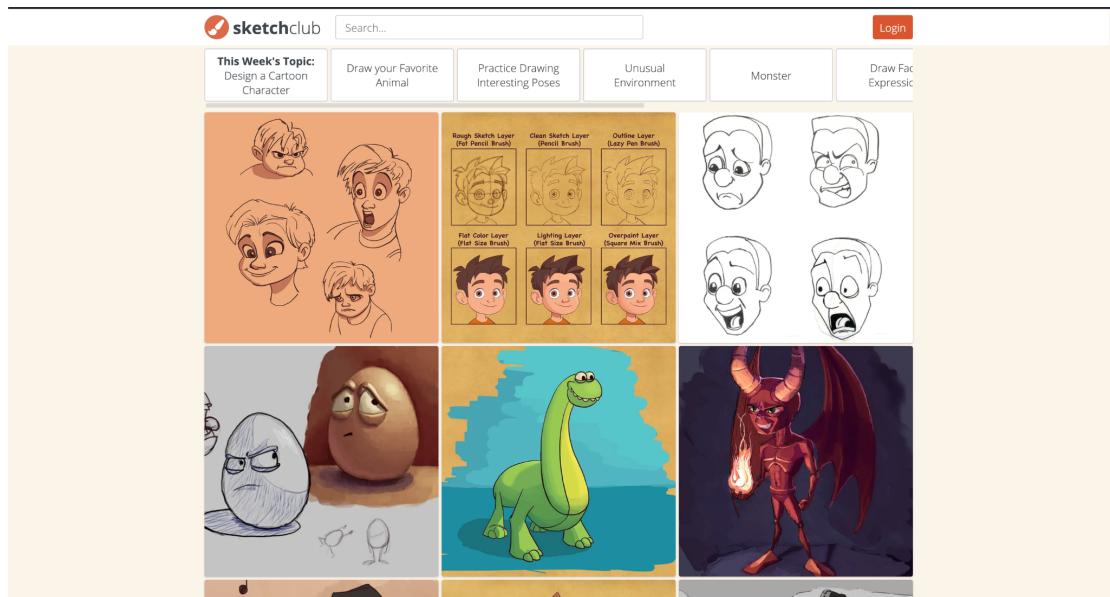
I have studied Figma's asset and component system [2.1], as well as ways of creating a design system in this tool [2.2].

For the purposes of a simple prototype I did not end up needing a full-fledged design system (this approach is very powerful, but more applicable to larger scale projects with dozens of screens), but I found it very helpful to create basic components of the page - buttons, images, links, and so on.

I was able to use these elements to assemble the designs of multiple pages quickly, and I was able to edit and change the styles with a lot of flexibility (since editing the style of the parent asset automatically updates the designs of all its descendants).

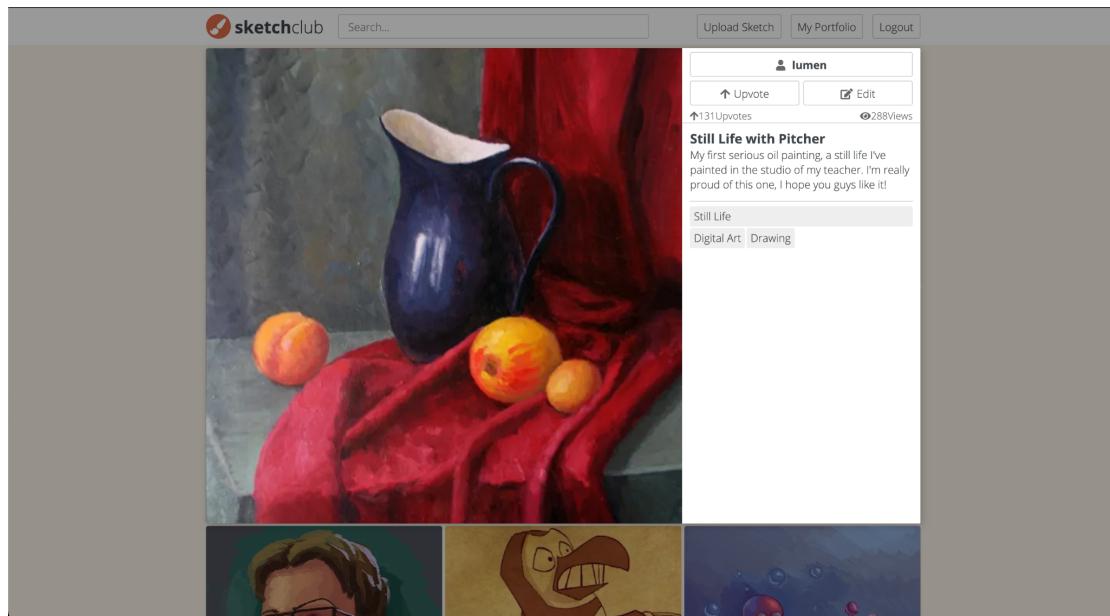
Below you will find the key pages of the website, along with the list of features I intended to build. In the pictures you can see the final look of the website - after multiple iterations and tweaking. You will find the original Figma mockups in the Appendix A. The Figma design project is available at [2.3].

Browse Sketches



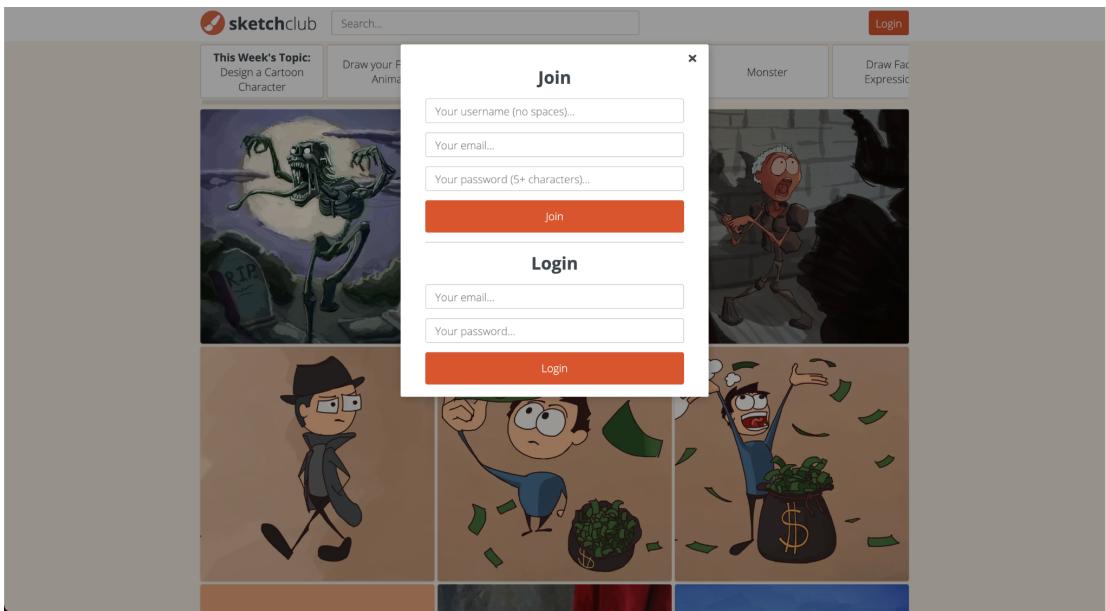
- A grid of sketches submitted to today's competition.
- A list of tags that can be used to filter the sketches by topic.
- Search bar that can be used to search by title, description, author or tag.

View Sketch



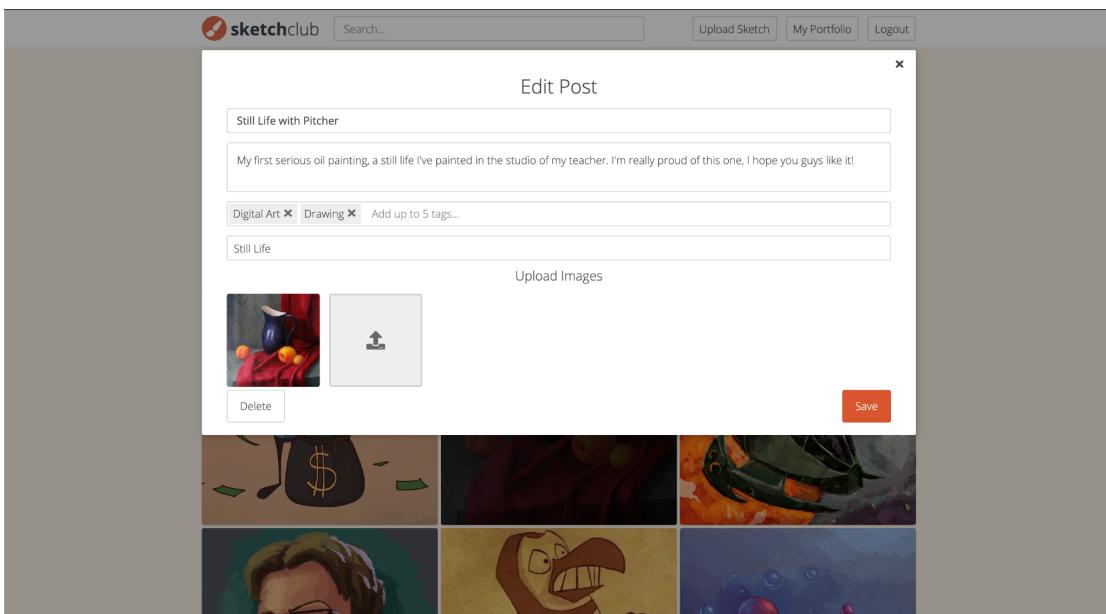
- View sketch images - image gallery, clicking on arrows switches the active image.
- Sketch information - title, author, description, topic, and tags.

Auth System



- Join. Enter the username, email, and password, create an account.
- Login. Enter the email and password, log into the existing account.

Edit Sketch



- A page where you can upload a new sketch, or edit an existing one.
- Enter sketch title, description, topic.
- Upload images, delete images.
- Save and publish the sketch, delete sketch.

Backend

In this section I will describe the backend part of my project. I will show you an overview of the files, explain how they fit together, and then give a more detailed explanation of the most interesting parts.

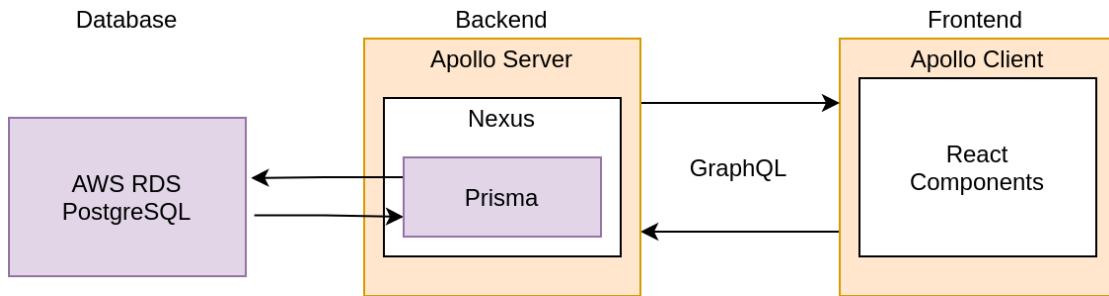
Project Structure Overview

This is an overview of the project structure and the key files.

backend	
└── apollo	
└── context.ts	/apollo/context.ts
└── config	
└── nginx.conf	/config/nginx.conf
└── nexus	
└── ImageQueries.ts	/nexus/ImageQueries.ts
└── ImageTypes.ts	
└── nexusSchema.ts	/nexus/nexusSchema.ts
└── postMutations.ts	
└── postQueries.ts	/nexus/postQueries.ts
└── postTypes.ts	
└── shield.ts	/nexus/shield.ts
└── userMutations.ts	
└── userQueries.ts	/prisma/schema.prisma
└── userTypes.ts	
└── prisma	/prisma/schema.prisma
└── add-topic.js	
└── schema.prisma	/prisma/schema.prisma
└── seed.ts	
└── seedbackup.ts	/prisma/schema.prisma
└── updateRank.ts	
└── server.ts	/server.ts
└── tsconfig.json	

To summarize - Apollo Server uses Nexus schema to create a GraphQL API. Nexus Schema is made out of Queries, Types, and Mutations which use Prisma in their resolvers to access data from the database. Queries and Mutations are protected by GraphQL Shield from unauthorized access, and the whole thing is served with Nginx.

Setting up GraphQL Server with Apollo, Nexus, and Prisma



Apollo Server is responsible for creating the GraphQL API that the client can use to fetch and modify the data.

There are multiple different implementations of the Apollo Server [3.1]. I chose 'apollo-server-express', an integration of Apollo with Express.js - a popular Node.js framework for building web applications. It was a convenient choice, because it provides certain functionality that the regular 'apollo-server' package doesn't - specifically, it can be easily integrated with the 'cookieParser' middleware, which I need to use in order to process the authentication cookies (more on that in the Authentication section).

Apollo Server builds a GraphQL API using Nexus schema [3.2]. Nexus is a JavaScript library which enables developers to create a GraphQL schema using JavaScript code, as opposed to using GraphQL's schema definition language [3.3] (more on benefits of Nexus in Tech Stack Review section).

In the '/nexus' folder you see multiple files containing Types, Queries, and Mutations for various models - Images, Posts, and Users.

Types describe the shape of the data in a way that GraphQL is able to understand - I can list the fields the client will be able to access (such as post title, body, tags, and so on) [3.4]. I can also describe ways in which the client can access related models - for example, the User type has the 'posts' field, which enables me to easily access all the posts created by the user.

Queries and Mutations are the “entry points” into GraphQL, they provide ways to access and modify the data [3.5]. Both Queries and Mutations have resolvers - the JavaScript functions that request and modify the data in the database [3.6].

Nexus queries have clearly defined input and output types, which is good for type safety - a very effective way to safeguard the application that is helpful for catching and eliminating the errors during development [3.7].

For example, the 'post' query accepts a post slug (a part of the url that uniquely defines the post) as its argument, uses Prisma to find the post in the database by this slug, and then returns the data of the type 'Post', which is defined in 'postTypes.tsx' and contains all the fields the client may need to request.

Inside the resolvers, you can see Prisma functions being executed. Prisma is the ORM that gives me the ability to actually access and modify the data in the database.

Think of it as an easier and more convenient alternative to manually writing the SQL queries [3.8].

In order to set up Prisma, I needed to define its schema in the ‘schema.prisma’ file (more on that in the Database Design section), and pass it the environment variables that describe how to connect to the PostgreSQL database. To be able to actually use Prisma in the resolvers to query and modify the database, I needed to configure apollo context (/apollo/context.ts), which provides access to Prisma’s functions inside of the resolvers.

One last step was establishing the permissions - some of the endpoints needed to be protected from unauthorized access (for example, I needed to make sure that only the post’s author is able to edit or delete the post, and only logged in users can vote on posts). To accomplish that, I have used GraphQL Shield - a library designed to simplify creating the permission layer for the application [3.9].

GraphQL Shield enables me to list all the Queries and Mutations I want to protect, and pass a function to each one - if the function returns true, the request is allowed to complete, otherwise, the user receives a permission error.

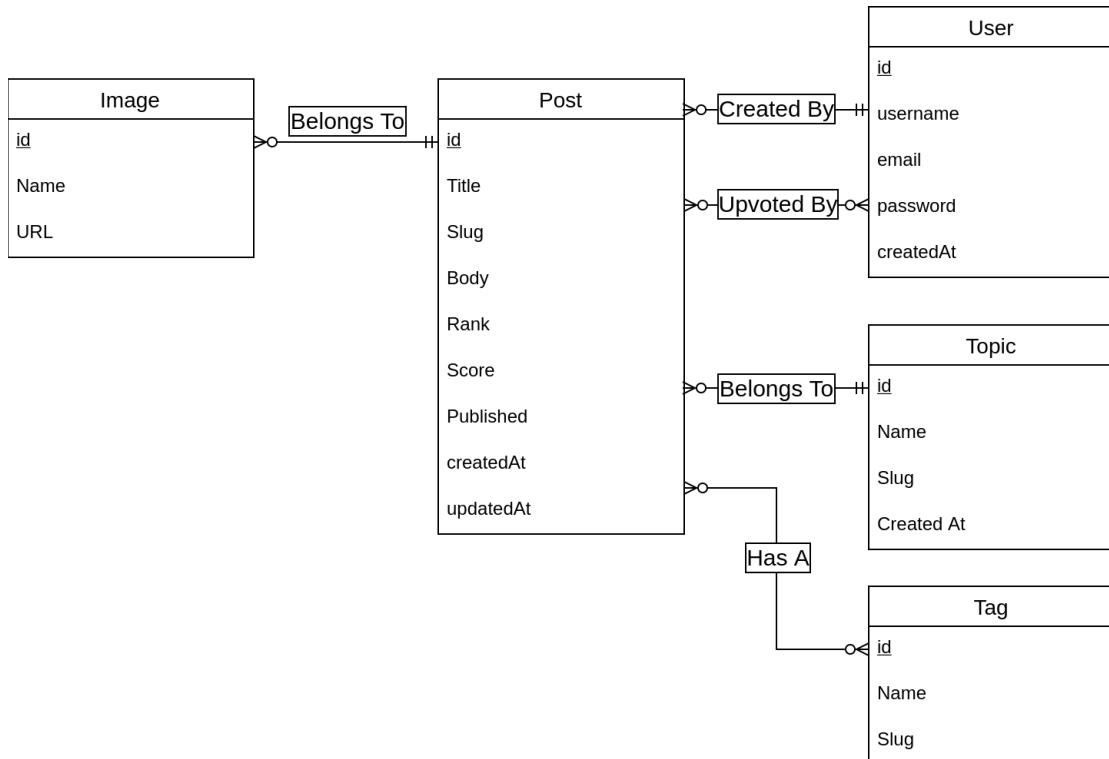
These functions are called “rules”, and they determine the user’s permissions based on the custom logic I wrote, for example - extracting the authentication token from the cookies to determine whether the user is logged in, and finding them in the database to make sure that they are the post’s author.

Everything I have described above achieved one important goal - the creation of a flexible and powerful GraphQL API my client is able to use to fetch and modify the data in the database.

With that complete, I was able to begin working on the actual functionality. In the sections below, I will highlight the most interesting parts of the backend code. They all fit within this framework of Queries and Mutations using Prisma to manipulate the database, but there are interesting details describing how it’s actually done that are worth pointing out.

Designing the Database and creating it using Prisma

Once the basic project set up was done, my tech stack was working, and I had managed to make my client and server communicate successfully, the next step was to design the database. I have created an ERD diagram to visualise the data I will need to store, and relations between the tables:



My goal was to capture all the most important information while keeping the design relatively simple.

Initially, Tags and Topics were the same entity, but after some experimentation and tweaks to the design I have decided to refactor the code and separate them, because topics needed to be sorted in the order in which they were created (so that I could display the most recent topic at the beginning of the list), but logically it did not make sense to have the creation date field on the tag model.

Also, at first the images were a part of the Post model (image urls were just stored as an array in the Post table), but after further consideration I have decided to split them off into their own entity, in case I needed to add some extra information to them later on. Right now the only extra field they have is "Name" (image's filename), but in the future it would be possible to enable users to give more detailed descriptions to each image, give credit to other artists if it's a collaborative project, or even add a model that stores videos, to enable people to share their drawing process.

Prisma's Schema file was used to establish the database models and relations between them. Prisma provides a straightforward and elegant way of doing that - I was able to list my tables, fields that I want to have on each one, and use special syntax to define one-to-many and many-to-many relationships as needed.

In the `/prisma` folder you can also see some helper files for seeding the data, manually adding the weekly challenge topic from the terminal, and regularly calculating the post ranking.

In order to seed the database with a lot of posts (as well as the initial demo users and topics), I have created a file (`seeddata.ts`) with JSON objects containing all the necessary information, and then wrote a script (`seed.ts`) which uses Prisma to establish the connection to the database, loops through all the data, and creates the necessary records in the database. This script also needed to hash the demo user's passwords (more on that in the Authentication section), and randomly assign posts to them.

Search and Filtering

Most Queries and Mutations are pretty similar to each other. To get an example of how they work, it's worth taking a look at the '`getPosts`' query (in the '`/nexus/postQueries.ts`' file).

This query accepts a number of arguments (search string, tags, author's username, etc), and its resolver uses them to filter the posts, and return only the ones that match a certain search query.

The filtering is done by using the Prisma query. First I build each individual part of the filter, depending on which arguments have been passed to the query (for example, filtering by tag will only occur if the client has requested such filtering), and then these filters are combined into one object (`allFilters`), which is passed to Prisma.

The ability to work on each part of the query as a separate javascript object approach provides a lot of power and flexibility, this kind of thing would be much more difficult and confusing to do if I was writing plain SQL queries by hand.

Filtering, sorting, and pagination are combined into one Prisma query, and then a separate query counts all the posts in the database that match the filter (knowing the total number of posts is necessary to build a pagination pagination), and both of these operations are combined into a single transaction. Finally, the object containing all of the requested data (and, unlike the REST API would have worked, nothing else) is returned from the resolver. The client will be able to use this object to render the grid of posts, with pagination.

Ranking the Posts

The posts are ranked based on their score (how many upvotes the post has received), and the submission date (the older the post, the lower it will be in the rankings). I chose to use the Hacker News ranking algorithm [3.10], because it is simple, well tested, and is a perfect fit for the sketching competition (ensuring that everybody's posts will be visible at the top of the page initially, and as the time passes, the posts ranking will get closer to upvotes-based sorting, so the most upvoted posts will be more prominent over time).

In order to sort the posts in such a way, I needed to add a “rank” field to the posts model, and update it regularly with the recalculated rank.

I wrote a script (`/prisma/updateRank.ts`) which loops through all the posts in the database, and uses their score and the submission data to calculate the updated rank.

This script is being run every 5 minutes using a cron job. It was necessary for efficiency reasons - as the database grows and users submit more and more posts, calculating the ranks in real time (whenever the user requests a page) would take a prohibitively long time. Updating the rank every 5 minutes is more than enough to show people the up to date results, and at the same time, the database can sort the posts based on the precalculated rank very quickly.

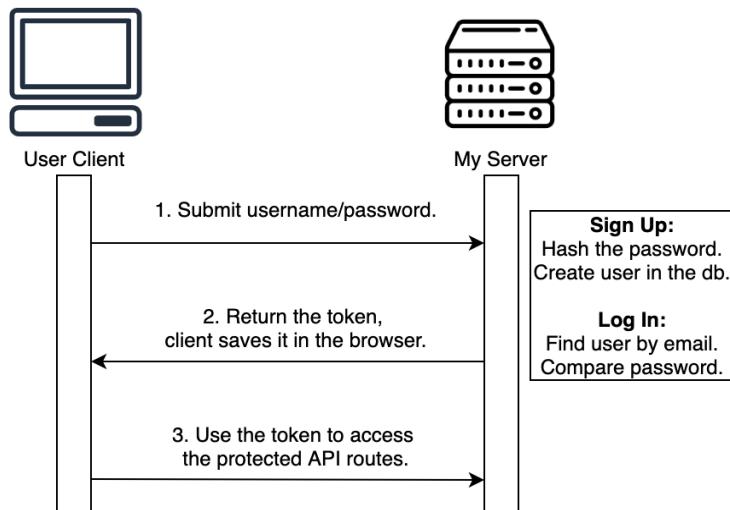
Authentication System

Building an authentication system was quite challenging, mostly because the approach was unfamiliar to me. But the final result ended up being pretty straightforward.

For authentication I have used JSON Web Tokens (JWT). JWT is a small piece of data generated on the server and stored in the browser. It contains some information about the user (in my case, `user_id`), which is *signed* by the server using a hashing algorithm [3.11]. Hashing algorithm uses a secret string known only by the server, and the information signed in such a way is guaranteed to have been created by my server.

Once JWT is obtained, my client can simply send it along with every request that requires authentication, my server will verify JWT's integrity, and once it is confirmed the client is granted access to the secure route.

Here's how the process works:



1. When the user enters their email and password in order to sign up or log in, they are sent to the server.
 - a. When the user is trying to sign up, we need to store their username and password in the database. For security reasons, the password can not be stored in a plain text form (which would expose it to the potential attacker), so it is important to hash it (run it through a one-way function that transforms it into a string of characters that can't be turned back into the password) before storing it in the database.
 - b. When the user is trying to log in, we compare the hashed version of the password they've entered with the hash saved in the database. If they match, it means that the password the user entered is correct.
2. The server generates the token and sends it back to the client. The client uses cookies to save the token in the browser.
3. The client automatically sends the token along with every request. The server verifies the integrity of the token to ensure that the token was generated by

my server, and grants the user permission to access the information behind the protected routes.

In order to verify the token, I first needed to be able to pass it from client to server on every request - Apollo Client (on the frontend) was set up in such a way as to get the “Authorization” cookie, and pass it along with the other headers [3.12].

On the server, in the file shield.ts, the token is verified using the secret string of characters, known only to the server, to confirm that the token was generated by my server (and not by the potential attacker).

In the same file you will find a set of “rules” - functions that decide whether or not the client can access a Query or a Mutation. These functions extract the user id from the verified token, which enables me to determine whether the user is logged in, as well as other information I need to determine which permissions the user should have (such as whether or not the user is the post’s author).

Scalable Image Uploads

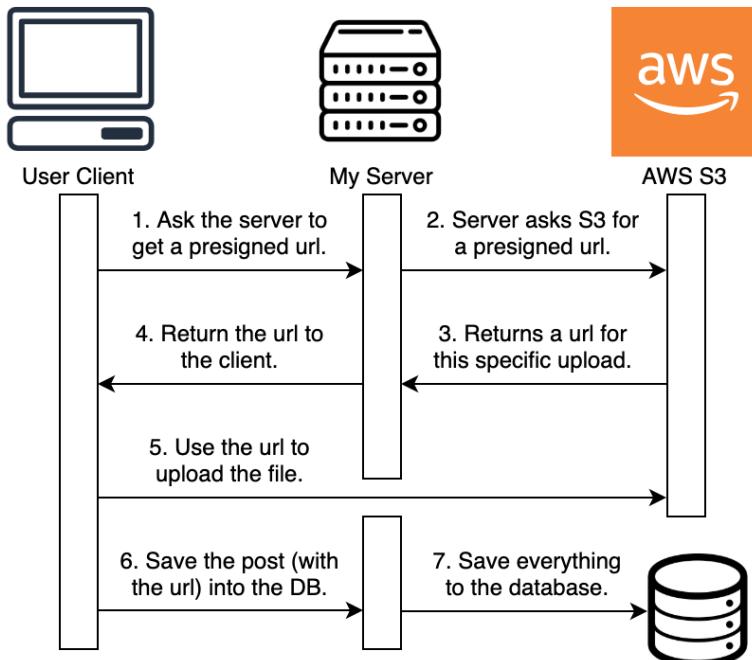
Multiple image upload options are available for a web application. Most common one involves uploading an image to your own server, and storing it in its own file system. The biggest problem with this approach is that it’s not scalable - this approach doesn’t allow you to easily spin up multiple servers (since any particular image is stored on a single one of them), and file storage/upload takes up a lot of space and bandwidth, which costs a lot.

The common solution is to use AWS - an industry standard for scalable file storage. People temporarily upload images to their own server when they submit the form, and then the server uploads the image to the database, storing only the image url in the database. This solves the problem of scalable storage, but the server’s resources are still being used during the image upload process (from the client to the server and from the server to AWS).

The best approach would be to send the image files from the client directly to AWS, without them ever touching my own server [3.15]. But in that situation, the challenge is keeping the image uploads secure - we don’t want anyone on the internet to be able to use our AWS servers for their file storage, and securing the file upload from client is more convoluted than just uploading it from the server (which we can assume is secure by default). But the benefits of cheaper and more scalable image storage outweigh the cost of complexity, so I have decided to implement this approach.

In order to upload the images securely, I am making use of so-called “Pre-Signed URLs” [3.14]. A Pre-Signed URL is generated by AWS, and can only be used once, for a file with specific name and type. If the URL isn’t being used immediately, it expires within seconds. This approach provides security - a unique url is generated for each and every file upload, which guarantees that only I know the URL that can be used to upload a file to my AWS bucket.

Here's how the process looks like [3.13]:



1. When the user wants to upload an image, the client asks the server to request a secure Pre-Signed URL from AWS.
2. The server tells AWS the name and type of the file I want to upload.
3. AWS generates the URL that will work only for uploading the file with this specific name and type, and sends it to my server.
4. My server returns this url to my client.
5. My client sends the file (using a POST request) directly to this URL. Notice that the file is being uploaded from the user's browser directly to the AWS server, not wasting the expensive storage or bandwidth of my own server.
6. Finally, I save the post (that simply contains a link to the uploaded file) to my database.

In addition to this process, I have also used a 'react-image-file-resizer' library [3.16] to automatically resize the images and convert them into a modern file format (webp) directly in the user's browser, saving space and bandwidth, and resulting in faster loading times. You can see the process of resizing the image in the 'utils/resizeImage.ts' file on the client.

Frontend

In this section I will provide an overview of my frontend project structure. I will describe the most important files, explain how the technologies I'm using fit together, and discuss the most interesting and relevant aspects of my code.

Project Structure Overview

```
└─ frontend
    └─ apollo
        └─ client.ts
        └─ fetchQuery.ts
        └─ imageQueries.ts
        └─ postsActions.ts
        └─ postsQueries.ts
        └─ userActions.ts
        └─ userQueries.ts
    └─ components
        └─ Elements
        └─ Layout
        └─ Posts
        └─ Topics
        └─ Users
    └─ context
    └─ hooks
    └─ pages
        └─ post
        └─ profile
        └─ tag
        └─ topic
    └─ _app.tsx
    └─ index.tsx
    └─ public
    └─ styles
    └─ utils
    └─ .env
    └─ next.config.js
    └─ package.json
```

/pages/

This folder is created by Next.js to set up file-based routing. Every file in it establishes a unique page on the website.

/pages/_app.tsx

This is the entry point into the app, a wrapper around all the other components (which will be rendered inside it). It wraps everything in Apollo Provider, which gives the components access to information fetched by Apollo.

/apollo/

Apollo Client is responsible for fetching the information from the server. It allows me to run GraphQL Queries and Mutations that request and modify the data.

/components/

All elements of the app are broken down into nested components, organized into folders based on their purpose.

/context/

React Context is a way to make certain data available to all the components in the component tree (such as a username of the authenticated user).

/hooks/

Hooks provide a way to use state (and many other features) in functional components. They're a modern alternative to React's Class Components, they enable me to write cleaner, more concise code that better follows the "DRY" principle.

/styles/

This folder contains all the SCSS styles responsible for the visual appearance of the app. Also fonts.

/public/

Next.js will automatically serve the contents of the folder named "public" without any changes, which is useful for serving the static files (such as my website's logo).

/utils/

Finally, I need a place to keep the miscellaneous files that don't neatly fit into the structure of my app, such as scripts that retrieve cookies and resize images.

Setting up the project with Next.js and TypeScript

The starter project was generated with Next.js [4.1]. This framework helps with the server-side rendering, routing, and provides the necessary setup for compiling the code. It supports TypeScript, to enable it I needed to install the TypeScript-related libraries, and change the file extensions from “.js” and “.jsx” to “.ts” and “.tsx”. These files will be automatically transpiled into regular JavaScript understandable by the browsers.

All the main pages on the website take advantage of the Next.js’ file-based routing [4.2]. When the user types a URL into the address bar and visits one of the pages on my website, the files contained in the “/pages/” folder determine which part of the app will be accessed depending on the path the user has entered. For example, when the user visits “<https://sketchclub.io/post/example>” page, Next.js will use the /pages/post/[postSlug].tsx file to generate the page.

The app is broken into multiple nested React components. The components determine all the functional elements of the app, and are responsible for rendering posts, images, login and submit forms, buttons, and so on. The components are made out of other components. For example the Browse page renders a grid of PostCard components, which in turn uses the SquareImage and Modal components to provide its functionality.

When the page loads, it renders one of the React components imported from the “/components/” folder. Most of the pages are wrapped in the Layout component, which contains the elements that are shared across pages (such as header with a menu and a search bar).

Apollo Set Up

In order to connect to the server and be able to query and manipulate the data with the GraphQL API I needed to set up and configure Apollo Client [4.3]. That is done in “/apollo/client.ts”, which contains the configuration related to passing the data between the server and the client.

Apollo Client is configured using Apollo Links library. Apollo Links enable the developer to customize the data flow between the client and the server . Before being passed to the server, the data is being passed through the chain of links - functions that can modify the request or perform certain actions based on the request [4.4].

That includes error logging (which enables me to see the GraphQL errors in the browser), attaching the cookies along with every request (which is used in my Authentication system), and modifying the request to avoid the CORS errors.

In the same file I am setting up Apollo Cache [4.5] - a way to manage the global state of the app, to store the server responses (so that the app could avoid making more unnecessary requests), which makes the app more responsive and saves on bandwidth (because once the request has been made, the data is saved in the browser and doesn’t need to be fetched again).

Once the client is set up, I wrap all of my app's components (in the `_app.tsx` file) in the Apollo Provider, which enables the components to access the data fetched by Apollo. With that, my components are able to run Queries and Mutations.

Queries, Mutations, and Actions

In the “`/apollo/`” folder there are multiple files containing the GraphQL queries, which are used to define the information my components will need to fetch from the server. They contain lists of variables I need to pass to the server (for example, ones responsible for searching through and filtering the posts), and the lists of fields I need the server to return in response. This is the data that will be used when rendering my components.

For every query there's an action. Actions wrap the queries into custom React Hooks (React's way of managing state inside of the functional components, more on that in the “Modal Component” section), the purpose of that is to keep my code clean and concise. Actions pass all the necessary variables and parameters to the queries, and can be used in the Components to request all the necessary data in one line of code.

To see a good example of how this works, consider the “`GET_POSTS`” query (you can see it in “`/apollo/postQueries.ts`”). This query is responsible for fetching the posts that will be displayed in the Browse, User Profile, and other pages. It accepts a long list of variables which enable me to filter posts by username, tag, search query, and to implement pagination. It returns all the data necessary for rendering the posts.

This query is wrapped into an action (“`useGetPosts()`” function inside of the “`/apollo/postsActions.ts`” file). This action is a React Hook. It extracts the necessary information from the router (which tells it which page the user is in, and all the parameters that can be passed in the address bar, such as search query), and passes it to the `GET_POSTS` query as variables.

The benefits of creating this hook are apparent once you realize that I need to use it in multiple pages - profile page, tag page, topic page, and index page. All of them use this hook to fetch the necessary posts, all filtered in ways specific to each page, using just one line of code (see the first line in the “`browse()`” function in the “`index.tsx`” file). This makes the code much more clear, concise, and convenient to write.

Apollo Cache

As described above, Apollo can cache the results of the queries I'm making, so that as the user clicks through the pages, the data never needs to be fetched more than once (if the user has already visited a certain page and then visits it again, its data will be loaded from the cache, instead of making a separate request to the server).

The downside of that is that updating the cache needs to be done manually. Luckily, Apollo makes it very easy. As you can see in the actions files (such as “`postsActions.ts`”), whenever I run a mutation (a function that updates the data on server, for example when a user creates a new post), I can pass it a “`refetchQueries`” parameter [4.6]. This parameter enables me to specify which queries need to be

refetched after the mutation is executed. For instance, once the user creates a post by running the “CREATE_POST” query, I’m telling apollo to automatically update the “GET_POSTS” query - which will fetch the most recent posts from the server (including the newly created one), and update the cache. As a result, the new post is visible in the app as soon as the user creates it (they don’t need to manually reload the page).

Server-Side Rendering

In order to achieve the server-side rendering, I need to tell Next.js how to fetch the necessary data while it’s rendering the page on the server. Once the data has been fetched, Next.js will generate the HTML and send it to the client. On the client, the HTML will be “hydrated” - the HTML event handlers will be attached to the static HTML that was returned from the server, making it dynamic and interactive.

Next.js has special functions that enable Server-Side rendering. Next.js can use “getStaticProps()” function in order to generate pages when the website is being built, this is a great fit for static websites where the content doesn’t change too often [4.7] - it speeds up the content delivery significantly (because all the server needs to do is send the pre-generated static files), but this approach is not very suitable for my website, since the content is meant to change often, and I want the posts to be visible as soon as the users submit them (without requiring me to rebuild the website every time).

On the other hand, “getServerSideProps()” function will be called on every request - when the client requests the page, this function will fetch the data from the API, and use it to render the HTML page that will be sent to the client [4.8].

Unfortunately, since this is happening on the server side, I was unable to use the same hooks that run queries inside the React components (more on how that works in the Apollo Setup section above) and fetch the data on the client.

I need to be able to run queries that provide the initial data that will populate the pre-rendered pages, and to do that I needed to create a separate function (“/apollo/fetchQuery.ts”) that sets up a simple Apollo Client when the page is being rendered on server, and fetches all the necessary data in a way that is compatible with this process.

How React Components Work (Modal Component Example)

Most of the React components are set up and work in a similar way. I will use the Modal component ('/components/Elements/Modal.tsx') as an example, because it incorporates all the interesting things I have learned about using React during this project - specifically, React Hooks and React Context.

Context is a new feature of React, which provides a convenient way to make a piece of data globally accessible by all the components within the app [4.9]. I am creating the contexts for all the pieces of data that need to be available across multiple components (such as whether the user is logged in, or which modal is currently open). I combine all the contexts into one CombinedContext (for the purpose of simplicity and clarity), which is later used in the _app.js to pass the data to all the components in my application.

React **Hooks** is another new feature, it is a way to use state (and other features that used to be available only on Class components) in React's functional components [4.10]. The 'useState()' hook is the most commonly used one, it allows me to create a variable that will be kept in the component's state, and a function that will be used to update it. Every time the state is updated, the component will rerender based on the new value of the state variable.

Using these two features together, I create Modal Context ('ModalContext.tsx'), which uses the 'useState()' hook to create a 'modal' variable, which contains a string that determines which modal is currently open (it'll be empty if no modal is open). I have also created the 'toggleModal()' function, which will update the 'modal' variable based on what model I want to open or close.

Both the 'modal' variable and the 'toggleModal()' function are passed down to all the app's components using context. All the contexts I need are combined together in the 'CombinedContext.tsx', which is wrapped around all the other components in '_app.tsx', giving them access to all the data I want to make available.

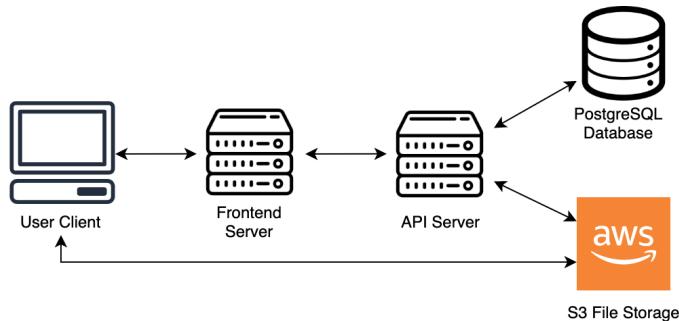
Now, the Modal component (a component used to render all the modals in the app) can plug into the context, and use the 'modal' variable I have established to decide whether or not this modal needs to be rendered. All the content I want to display inside of the modal is passed to it as its children, which abstracts away the modal functionality from the actual content I want to render inside of it.

Finally, when I want to show a specific modal (for example when the user clicks a "Login" button in the 'Header.tsx'), I simply attach the 'toggleModal()' function to the button's 'onClick()' event. The user clicks on a button, 'toggleModal()' function uses a hook to update the value of the 'modal' variable in the Modal Context, which lets the 'Modal' component know that it should be rendered.

Other components are set up in a similar way, using similar functionality to make the app responsive, interactive, and fun to use.

Deployment

Here's how the deployment architecture of this project looks like:



Frontend Server on Vercel

The frontend server is running on Vercel [5.1] - a free and convenient platform built by the developers of Next.js and optimized for hosting of Next.js apps. It was an obvious choice for hosting the frontend, since this platform makes the process very convenient and straightforward.

The main benefit of Vercel is that they're using a “serverless” approach [5.2] - the developers don't have to pay a fixed price for a constantly running server . Instead, the serverless functions are being ran only when necessary - whenever the user requests a page, the Vercel platform will run a serverless Next.js function required to generate the page on demand. That approach is cheaper, scaling of the application can be handled by Vercel automatically, and it doesn't require me to manage the server.

As a side benefit, it comes with CI/CD (continuous integration and continuous deployment) that is set up automatically. As soon as I push my code to GitHub, Vercel will automatically build it and deploy the new version, which is very convenient [5.3].

Technically, deployment to Vercel isn't very difficult, the only requirement was that the API should be served over https, which I intended to do anyway, and will discuss further below.

Backend API Server on DigitalOcean

In order to run my GraphQL API, I needed to set up my own linux server. I chose DigitalOcean to host my project because it is cheap, convenient to use, and I worked with it on my previous projects.

Setting up the ubuntu server was relatively straightforward - I needed to install the necessary packages (git, node, nginx, and so on), clone my git repository, install the npm packages, and finally build and run my backend code [5.4]. In order to run the backend code constantly, as a background process I have used pm2 - a convenient node package designed for managing daemon processes [5.5].

The challenging part was to learn how to acquire the certificates, and make my API available over HTTPS.

Serving the API over HTTPS

First, I needed to point the subdomain on which I want to serve the api (api.sketchdaily.io) to the server that's running my backend code. To accomplish that I have created an A record in the Vercel's DNS configuration, and pointed it to the IP of my DigitalOcean server.

Once it was done, I needed to acquire the SSL certificates. To do that, I have used Let's Encrypt - a certificate authority that can provide SSL certificates for free [5.6].

I have used certbot (a tool provided by Let's Encrypt) to automatically acquire the certificates, and configured Nginx to use these certificates, and connect all the requests to the port where I'm running my API. I had to set up a cron job to renew these certificates automatically, which will ensure that they will be updated every month.

While setting up the deployed project, I ran into an issue that took me a long time to resolve - despite setting everything up according to the documentation, I kept encountering the CORS (Cross-Origin Resource Sharing) errors [5.7] every time my client deployed on vercel was making requests to the API. It took quite a long time to solve, since it was impossible to test locally (everything worked on my laptop). Luckily, the solution was simple - it involved tweaking an obscure setting in the Apollo Client, as described in this issue [5.8].

I also needed to make sure that the Authorization JWT token is sent through headers, not cookies. You can see both of the tweaks in '/apollo/client.ts' on the frontend app. These solutions are not well documented and finding them required a lot of experimentation, but that's just a drawback of using technologies that are still being developed.

Finally, I needed to configure Nginx to provide the valid certificates in order to serve my GraphQL endpoint over https. Since certificates and keys were automatically generated for me by certbot, all I needed to do was configure '/config/nginx.conf' such that all the traffic is using https, is redirected to the port where my API is being served, and pass the paths to the certificates to the config.

PostgreSQL server with AWS RDS

For hosting my PostgreSQL database, I chose AWS RDS (Relational Database Service). It is a convenient way to host a PostgreSQL database in the cloud for free (on the free tier) [5.9].

The setup was pretty straightforward - I made an account, created a public database instance, and used the link provided to me to connect to the database from Prisma.

As the link must be secure, it is passed to using environment variables - I have a '.env' file which contains the secret variables necessary for the app to run, it is not committed to github to make sure that secret passwords stay secret. The local '.env' file on my laptop contains the variables I use in development (connecting me to the PostgreSQL database running on my laptop), and the '.env' file I'm using in production connects me to the database running on AWS RDS.

There isn't much else to say on this subject - AWS RDS is an established technology, they've made it as simple as possible to use, and everything worked just as expected, without the need for any extra research or creative workarounds. It was nice to use something stable and reliable for a change, and I'll definitely use it in my future projects as well.

Image Hosting with AWS S3

The image upload process is described in the "Scalable Image Uploads" section, here I will just talk about the AWS S3 set up. The biggest challenge was to set up the permissions using AWS Identity and Access Management (IAM) [5.10], which introduced me to some unfamiliar concepts.

It's very important to keep the S3 bucket secure, because if someone else gains access to it and uploads their files to my account, it can be very expensive and dangerous (because the attacker might upload something illegal or harmful).

To secure the S3 bucket, I am generating the secret access keys using IAM. IAM gives me a lot of flexibility and control over the permissions I want to grant to the person holding these credentials.

IAM has two types of records - Policies and Users [5.11]. Policies are assigned to the User and describe what the user can do. This system is extremely flexible, and gives me a very granular control over permissions. I have created a Policy that specifically enables a person to only upload images to my S3 bucket, and assigned it to a new IAM user (who's credentials my server will use to request the pre-signed urls).

Since this policy has very limited rights and works only with one S3 bucket, that ensures that even if someone does get access to the keys, the scope of the damage they can do to my AWS account is limited.

One more security measure that is available is restricting the sources from which AWS will accept the uploads to only my website. That way, a malicious person wants to allow the users of their website to upload the images, they won't just be able to request pre-signed urls from my server, and make uploads.

Each S3 bucket has a CORS configuration section [5.12], customized using JSON config written in a specific format (see '/config/bucket-cors-config.json'). I have configured it to only accept the POST requests from my domain.

Finally, S3 buckets have their own permission policies which control who can view the content inside of the bucket. By default, it is restricted to the person who has created a bucket, but in my case, I needed to relax them and make the images available publicly (because that's the purpose of my app, for everyone on the internet to be able to view the uploaded images). You can see the policy I've created in '/config/bucket-access-policy.json'.

With that, my deployment structure is complete - frontend on Vercel, backend on Digital Ocean, Postgres on AWS RDS, and file storage with AWS S3.

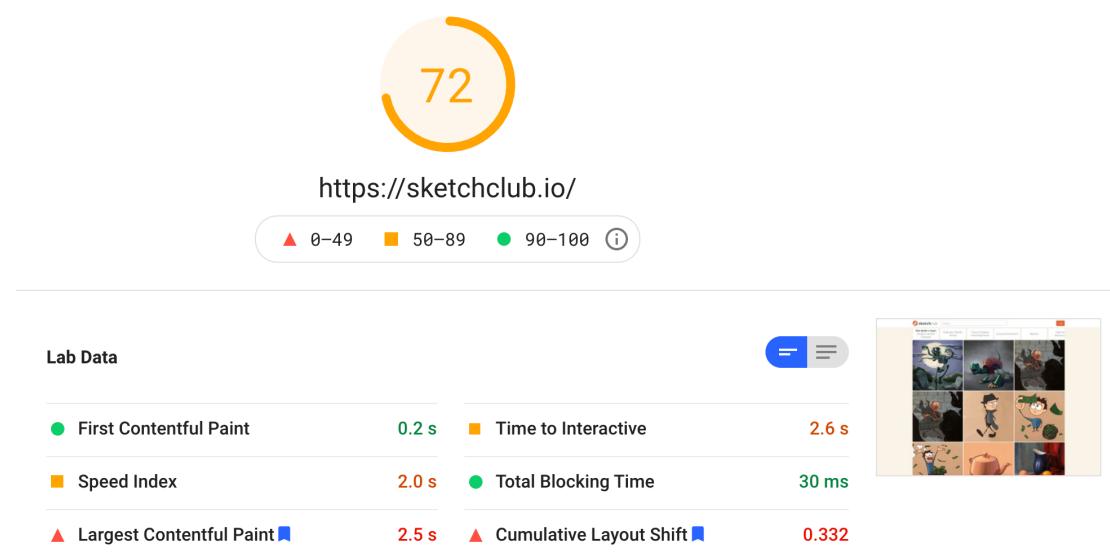
Evaluation

Finally, it was time to launch and evaluate my app. I have done that in two ways - I have evaluated the performance using Google's Page Speed Insights tool, and I have evaluated the user experience by sharing the website with some artists and asking them to fill in a short survey.

Performance

To evaluate the performance of the app, I have used Google's Page Speed Insights [6.1] - it is a standard tool that helps people to estimate how quickly their website loads and how responsive it is.

At first, the score was pretty disappointing:

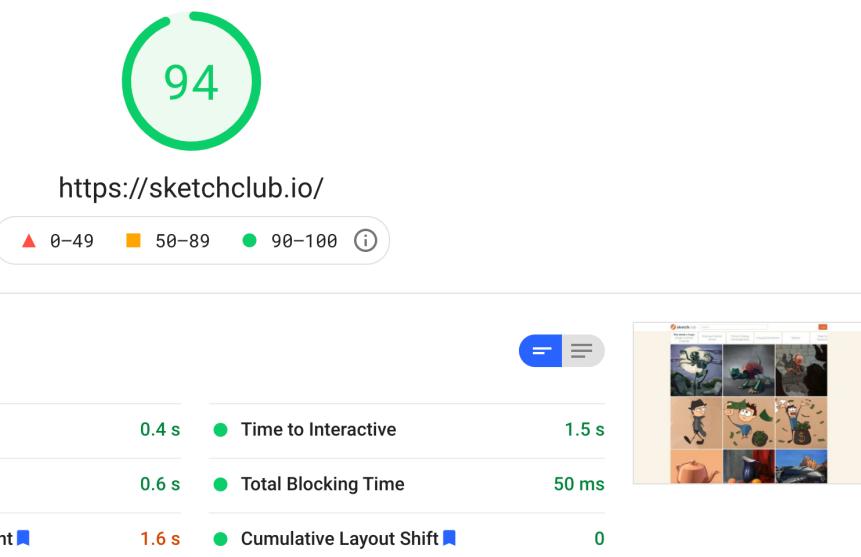


This tool provides some suggestions that help the developer to analyze and reduce the bottlenecks.

In order to improve the score (and therefore, the page speed), I have implemented google's suggestions. First, I have converted all the images into a '.webp' format - a modern lossless image compression format that results in much smaller image sizes compared to '.png'. I have also made sure to update the code in 'resizelimage.ts' to automatically generate webp files in the browser, before image upload.

In addition to that, I had to tweak the 'Cache-Control' settings in the AWS S3 bucket in order to keep the images cached for a longer period of time. Since the images on my website are never updated (when the user wants to change the image, they just delete the old one and upload a new one, which will have a different url), it was safe to set the cache age very long (1 year).

After implementing the suggestions, I have managed to achieve a score I am happy with:



A different tool (GTmetrix) gave me a score of 96%, which is also very good. The mobile test score (60%) is acceptable but still leaves much to be desired. The biggest improvement that could be made is to "Reduce initial server response time". It is my understanding that server response time depends mainly on the performance of serverless functions on Vercel, and the time it takes to contact the database.

Building my own frontend server in the same region as the backend and database servers could improve this metric, and perhaps some other experiments with network architecture and the project build settings could potentially improve its performance, but doing that is out of scope of this project. As of now, I am quite happy with the results, the most important bottlenecks have been fixed, and more minor tweaks can be done at a later date.

User Feedback

While I was working on the app, I started a Discord community for Digital Artists, to build the audience and get to know people who could leave feedback on my app once it's ready.

I have built a survey (Appendix B) with Google Forms and used it to determine people's response to the app. I have shared the survey in my discord community, asked people to try using the app and fill in the survey.

I am happy to report that the user feedback (Appendix C) was largely positive. The sample size was pretty small (10 people), but everyone who has participated in the survey seemed to have liked my app.

Most people rated my website 5/5 in terms of usability, responsiveness, and usefulness. Users did not report experiencing any bugs or issues, and seemed to have no problem using the website.

When asked to suggest changes and improvements, users have left some pretty useful recommendations. People wanted more options for filtering and organizing the weekly topics, they wanted to be able to submit their own weekly topics, they wanted to have more personalization and customization options (artists want to have more control over their portfolio profiles).

These are very useful and interesting ideas, many of them can be added to the roadmap and implemented in the future.

At the same time, most of these are "nice to have" features, not something urgent or essential for the core user experience - so it seems like I have managed to successfully implement the core functionality, and now what's left is gradual and incremental improvement of this project. Adding functionality, talking to users, and growing our community.

Conclusions

My goal with this project was to explore the modern web development technologies and become a better full-stack web developer. I wanted to build an application that is fast, beautifully designed, and easy to use.

In the process of building this website, I have learned a lot about React (especially its new features like Hooks and Context), Next.js, Apollo, GraphQL, Nexus, and Prisma. I have never used most of these technologies before (aside from React), and it was a great opportunity to learn new things and get up to speed with the cutting-edge advances in web development. Using this stack turned out to be quite complicated, much more difficult than the technologies I'm used to (Node/Express), but very rewarding.

It is clear that most of the advantages of these technologies are more apparent on large scale projects, complicated software built by teams of developers. However, I have found that once you get used to this stack and understand how to use it, it can provide a lot of power and flexibility to the personal projects as well.

React enables me to build any interactive web app I can imagine, Next provides a powerful framework for creating a full-stack web application and rendering it on the server, and Apollo/Nexus/Prisma allow me to create a powerful, flexible, well-organized (but a little too verbose) GraphQL API.

Having said that, I don't think that all the added development time and complexity is worth it when it comes to working on really small projects and prototypes. In the future, I will definitely use this stack for projects that are big, complex, long-term, and require collaboration with other people. But for small apps and prototypes, I will probably stick with REST APIs and non-server-side-rendered React apps (since SSR is one of the major causes of the increased complexity).

Using AWS S3 for file storage and AWS RDS for database storage proved to be extremely powerful and, now that I know how to do that, quite easy. They will be my tools of choice going forward. Hosting the frontend on Vercel was a very pleasant and seamless experience, I will use it in my future projects as well.

I still have a lot to learn about TypeScript, since I only took advantage of its most basic features. I will need to spend more time exploring the advantages of using strict typing to ensure type safety (which I couldn't do during this project since that would've added a lot of complexity, slowed down the development, and made the already tough learning curve much steeper). It is a promising and interesting technology, and it is worth exploring in more depth.

Designing this app in Figma was also a valuable and interesting experience, this tool turned out to be surprisingly powerful, flexible, and easy to use. The ability to quickly prototype the design and make tweaks to it without needing to rewrite the code is extremely helpful and valuable. I will definitely use it in my future projects.

The skills I have learned while building this project will definitely be very helpful in my job search, and I hope they will lead to an interesting and successful career.

References

1. Tech Stack review and Technical Challenges

[1.1] ReactJS - Getting Started

<https://reactjs.org/docs/getting-started.html>

[Accessed 14 Aug. 2021]

[1.2] What is Server-Side Rendering? Definition and FAQs.

<https://www.omnisci.com/technical-glossary/server-side-rendering>

[Accessed 14 Aug. 2021]

[1.3] Next.js by Vercel - The React Framework.

<https://nextjs.org/>

[Accessed 14 Aug. 2021]

[1.4] Client Side Rendering Vs Server Side Rendering in React js & Next js.

<https://yudhajitadhikary.medium.com/client-side-rendering-vs-server-side-rendering-in-react-js-next-js-b74b909c7c51>

[Accessed 15 Aug. 2021]

[1.5] GraphQL: A query language for APIs.

<https://graphql.org/>

[Accessed 15 Aug. 2021]

[1.6] GraphQL vs REST - A comparison.

<https://www.howtographql.com/basics/1-graphql-is-the-better-rest/>

[Accessed 16 Aug. 2021]

[1.7] The Apollo GraphQL platform.

<https://www.apollographql.com/docs/intro/platform/>

[Accessed 16 Aug. 2021]

[1.8] GraphQL Schemas and Types

<https://graphql.org/learn/schema/>

[Accessed 16 Aug. 2021]

[1.9] Code-First, Type-Safe, GraphQL Schema Construction.

<https://github.com/graphql-nexus/nexus>

[Accessed 17 Aug. 2021]

[1.10] What is Prisma? (Overview).

<https://www.prisma.io/docs/concepts/overview/what-is-prisma>

[Accessed 17 Aug. 2021]

[1.11] Prisma Studio

<https://www.prisma.io/studio>

[Accessed 17 Aug. 2021]

[1.12] PostgreSQL database connector (Reference).

<https://www.prisma.io/docs/concepts/database-connectors/postgresql>

[Accessed 17 Aug. 2021]

[1.13] Smallcombe, M. Mongodb vs. PostgresQL: Compare Database Structure.

<https://www.xplenty.com/blog/mongodb-vs-postgresql/>

[Accessed 18 Aug. 2021]

[1.14] TypeScript for JavaScript Programmers

<https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html>

[Accessed 18 Aug. 2021]

[1.15] TypeScript pros: what makes TypeScript a good fit for large projects

<https://www.altexsoft.com/blog/typescript-pros-and-cons/>

[Accessed 18 Aug. 2021]

[1.16] Feng, R. JavaScript to Typescript Migration Benefits.

<https://www.appdynamics.com/blog/engineering/the-benefits-of-migrating-from-javascript-to-typescript/>

[Accessed 18 Aug. 2021]

[1.17] Teese, B. TypeScript, Flow and the importance of toolchains over tools.

<https://shinesolutions.com/2017/01/05/typescript-flow-and-the-importance-of-toolchains-over-tools/>

[Accessed 18 Aug. 2021]

[1.18] Amazon Web Services, Cloud Object Storage.

<https://aws.amazon.com/s3/>

[Accessed 18 Aug. 2021]

[1.19] Amazon RDS for PostgreSQL.

<https://aws.amazon.com/rds/postgresql/>

[Accessed 18 Aug. 2021]

[1.20] Deployment | Next.js.

<https://nextjs.org/docs/deployment>

[Accessed 18 Aug. 2021]

[1.21] DigitalOcean Droplets.

<https://www.digitalocean.com/products/droplets/>

[Accessed 18 Aug. 2021]

[1.22] Figma for Frontend Developers & Designers.

<https://www.figma.com/figma-for-frontend-developers/>

[Accessed 18 Aug. 2021]

[1.23] Components, styles, and shared library best practices.

<https://www.figma.com/best-practices/components-styles-and-shared-libraries/>

[Accessed 18 Aug. 2021]

2. Functionality and UI/UX Design

[2.1] Component architecture in Figma

<https://www.figma.com/best-practices/component-architecture/>

[Accessed 19 Aug. 2021]

[2.2] Andrew, M. Creating a design system in Figma: a practical guide.

<https://uxdesign.cc/creating-a-design-system-in-figma-cbd01b0d2424>

[Accessed 19 Aug. 2021]

[2.3] Figma Design File

<https://www.figma.com/file/mFcOSSRfhBldiwDrf7kk7G/Sketch-Club>

[Accessed 19 Aug. 2021]

3. Backend

[3.1] Choosing an Apollo Server package

<https://www.apollographql.com/docs/apollo-server/integrations/middleware/>

[Accessed 19 Aug. 2021]

[3.2] Nexus Schema

<https://nexusjs.org/docs/guides/schema>

[Accessed 19 Aug. 2021]

[3.3] The Problems of "Schema-First" GraphQL Server Development

<https://www.prisma.io/blog/the-problems-of-schema-first-graphql-development-x1mn4cb0tyl3>

[Accessed 19 Aug. 2021]

[3.4] Nexus objectType

<https://nexusjs.org/docs/api/object-type>

[Accessed 19 Aug. 2021]

[3.5] Nexus Mutations

<https://nexusjs.org/docs/getting-started/tutorial/chapter-adding-mutations-to-your-api>

[Accessed 20 Aug. 2021]

[3.6] Writing Query Resolvers.

<https://www.apollographql.com/docs/tutorials/resolvers/>

[Accessed 20 Aug. 2021]

[3.7] Type-Safe GraphQL Servers.

<https://medium.com/open-graphql/type-safe-graphql-servers-3922b8a70e52>

[Accessed 20 Aug. 2021]

[3.8] What is Prisma?

<https://www.prisma.io/docs/concepts/overview/what-is-prisma>

[Accessed 20 Aug. 2021]

[3.9] GraphQL Shield - A GraphQL tool to ease the creation of permission layers.

<https://github.com/maticzav/graphql-shield>

[Accessed 20 Aug. 2021]

[3.10] How Hacker News Ranking Algorithm Works.
<https://medium.com/hacking-and-gonzo/how-hacker-news-ranking-algorithm-works-1d9b0cf2c08d>
[Accessed 20 Aug. 2021]

[3.11] Introduction to JSON Web Tokens
<https://jwt.io/introduction>
[Accessed 20 Aug. 2021]

[3.12] Grider S. Advanced React and Redux.
<https://www.udemy.com/course/react-redux-tutorial/>
[Accessed 20 Aug. 2021]

[3.13] Grider S. Node JS: Advanced Concepts.
<https://www.udemy.com/course/advanced-node-for-developers/>
[Accessed 20 Aug. 2021]

[3.14] Generate a presigned URL in modular AWS SDK for JavaScript.
<https://aws.amazon.com/blogs/developer/generate-presigned-url-modular-aws-sdk-javascript/>
[Accessed 20 Aug. 2021]

[3.15] Uploading to Amazon S3 directly from a web or mobile application
<https://aws.amazon.com/blogs/compute/uploading-to-amazon-s3-directly-from-a-web-or-mobile-application/>
[Accessed 20 Aug. 2021]

[3.16] React Image File Resizer
<https://www.npmjs.com/package/react-image-file-resizer>
[Accessed 20 Aug. 2021]

4. Frontend

[4.1] Create a Next.js App.
<https://nextjs.org/learn/basics/create-nextjs-app>
[Accessed 21 Aug. 2021]

[4.2] File-system Based Router.
<https://nextjs.org/docs/routing/introduction>
[Accessed 21 Aug. 2021]

[4.3] Set up Apollo Client
<https://www.apollographql.com/docs/tutorial/client/>
[Accessed 21 Aug. 2021]

[4.4] Apollo Link overview
<https://www.apollographql.com/docs/react/api/link/introduction/>
[Accessed 21 Aug. 2021]

[4.5] Configuring the Apollo Client cache

<https://www.apollographql.com/docs/react/caching/cache-configuration/>

[Accessed 21 Aug. 2021]

[4.6] When To Use Refetch Queries in Apollo Client

<https://www.apollographql.com/blog/apollo-client/caching/when-to-use-refetch-queries/>

[Accessed 21 Aug. 2021]

[4.7] Next.js Data Fetching - getStaticProps()

<https://nextjs.org/docs/basic-features/data-fetching#getstaticprops-static-generation>

[Accessed 21 Aug. 2021]

[4.8] Next.js Data Fetching - getServerSideProps()

<https://nextjs.org/docs/basic-features/data-fetching#getserversideprops-server-side-rendering>

[Accessed 21 Aug. 2021]

[4.9] React Context

<https://reactjs.org/docs/context.html>

[Accessed 21 Aug. 2021]

[4.10] React Hooks

<https://reactjs.org/docs/hooks.html>

[Accessed 21 Aug. 2021]

5. Deployment

[5.1] Vercel

<https://vercel.com/about>

[Accessed 21 Aug. 2021]

[5.2] Serverless Functions on Vercel

<https://vercel.com/docs/serverless-functions/introduction>

[Accessed 21 Aug. 2021]

[5.3] Automatic Deployments with Vercel for GitLab

<https://vercel.com/guides/getting-started-with-vercel-for-gitlab>

[Accessed 21 Aug. 2021]

[5.4] DigitalOcean - Initial Server Setup with Ubuntu 20.04.

<https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-20-04>

[Accessed 21 Aug. 2021]

[5.5] Advanced, Production Process Manager for Node.js.

<https://pm2.keymetrics.io/>

[Accessed 21 Aug. 2021]

[5.6] Let's Encrypt - a nonprofit Certificate Authority.

<https://letsencrypt.org/>

[Accessed 21 Aug. 2021]

[5.7] CORS errors

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors>

[Accessed 21 Aug. 2021]

[5.8] How to avoid "self signed certificate" error.

<https://github.com/apollographql/apollo-link/issues/229>

[Accessed 21 Aug. 2021]

[5.9] Amazon RDS for PostgreSQL

<https://aws.amazon.com/rds/>

[Accessed 21 Aug. 2021]

[5.10] AWS Identity & Access Management.

<https://aws.amazon.com/iam/>

[Accessed 21 Aug. 2021]

[5.11] Policies and Permissions - AWS Identity and Access Management.

https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html

[Accessed 21 Aug. 2021]

[5.12] CORS configuration - Amazon Simple Storage Service.

<https://docs.aws.amazon.com/AmazonS3/latest/userguide/ManageCorsUsing.html>

[Accessed 21 Aug. 2021]

6. Evaluation

[6.1] PageSpeed Insights.

<https://developers.google.com/speed/pagespeed/insights/>

[Accessed 22 Aug. 2021]

[6.2] GTmetrix - Website Speed and Performance Optimization.

<https://gtmetrix.com/>

[Accessed 22 Aug. 2021]

[6.3] Sketch Club Survey

https://docs.google.com/forms/d/e/1FAIpQLScPNwEEQIzpSI5ZYQWm_0QPO1NcUU8y-GuwSw74A2A2Q5QSAw/viewform

[Accessed 22 Aug. 2021]

Appendix

A. Original Mockups

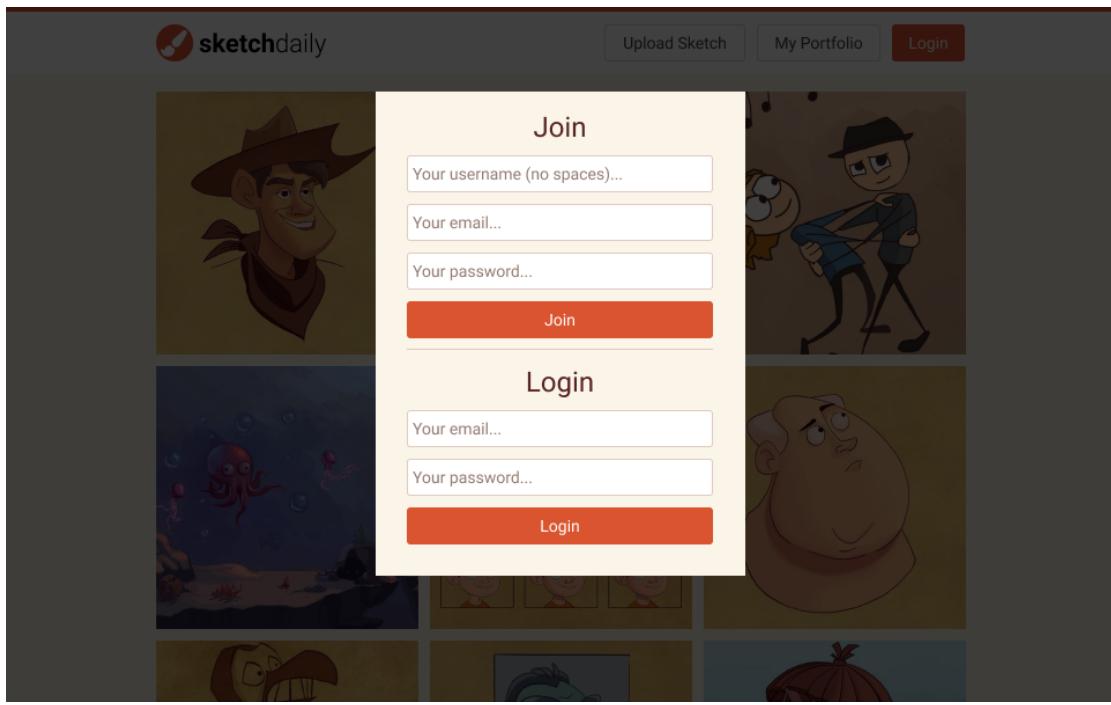
Browse Sketches

The screenshot shows a user interface for browsing sketches. At the top, there's a navigation bar with the sketchdaily logo, an 'Upload Sketch' button, a 'My Portfolio' button, and a 'Login' button. Below the navigation is a title 'Today's topic: Cartoon Character'. The main area displays a 3x3 grid of cartoon illustrations. The first row includes a cowboy, a green dinosaur, and a couple dancing with the text 'Tango Together' and 'by LilyAldrin'. The second row features an octopus in an underwater scene, three small portrait sketches labeled 'Rough Sketch Layer', 'Clean Sketch Layer', and 'Outline Layer', and a portrait of a bald man. The third row shows a continuation of the underwater scene with bubbles and small fish.

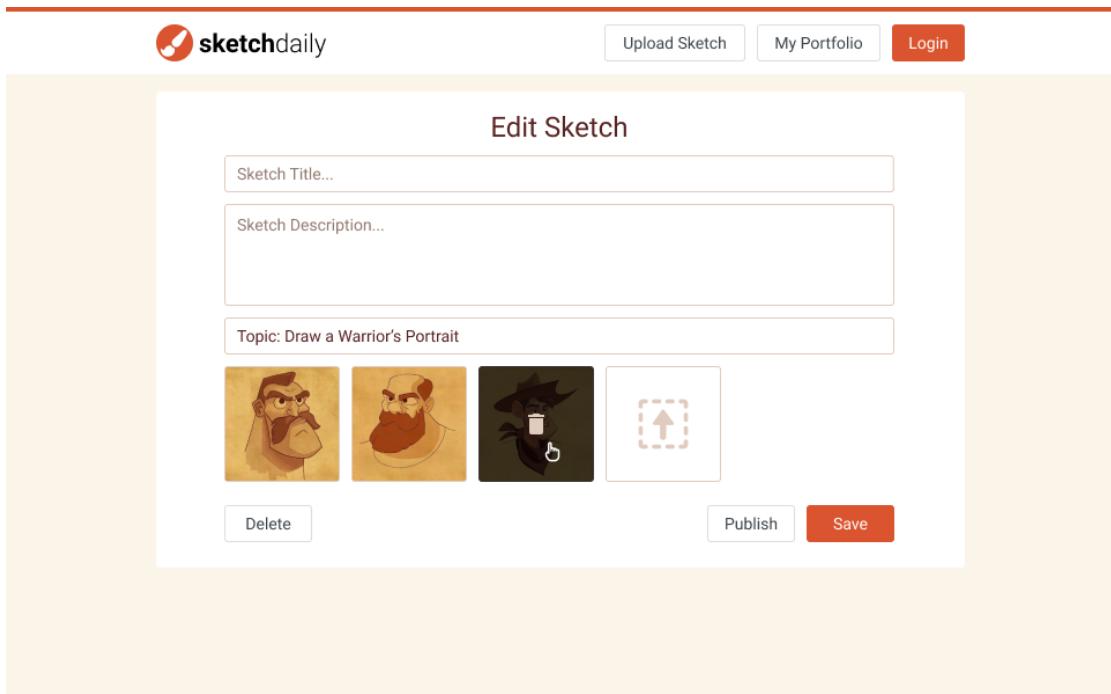
View Sketch

This screenshot shows a detailed view of a specific sketch titled 'Octopus Sketch' by LilyAldrin. The main image is a vibrant underwater scene featuring a large, pinkish-red octopus swimming gracefully. The background is filled with bubbles and smaller marine life like fish and crabs. To the right of the image is a description box: 'Octopus Sketch' by LilyAldrin, followed by the text 'An adorable octopus I drew for the "Draw your Spirit Animal" challenge. This is my first time drawing an underwater scene.' Below this is a message 'I hope you guys like it!'. At the bottom of the description box is a button labeled 'Draw Your Spirit Animal'. The top navigation bar and the 3x3 grid of sketches from the previous screenshot are visible at the bottom of this view.

Auth System



Edit Sketch



B. Survey

Survey

<https://forms.gle/KxPtaZ1WTUGYQAp27>

The screenshot shows a Google Forms survey titled "Sketch Club Survey". The survey consists of several questions with radio button and scale options:

- Question: How experienced are you at Digital Art?
Options: Novice, Intermediate, Experienced.
- Question: How useful do you think this website would be to you?
Scale: 1 (Not useful at all) to 5 (Very useful).
Current selection: 1 (Not useful at all).
- Question: Was the website easy to use?
Scale: 1 (Very Hard to Use) to 5 (Very Easy to Use).
Current selection: 1 (Very Hard to Use).
- Question: How likely are you to recommend this website to a friend?
Scale: 1 (Not likely) to 5 (Very likely).
Current selection: 1 (Not likely).
- Question: Did you encounter any issues while using the website? Was anything confusing or inconvenient?
Text area: Your answer
- Question: If you could change one thing about the website, what would it be?
Text area: Your answer
- Question: Any other thoughts/comments/suggestions?
Text area: Your answer

At the bottom right is a red "Submit" button.

20:47 =====

Hey, @everyone !

I'm a Computer Science student, and I'm working on my Master's project - a website that helps artists to regularly practice their drawing skills. Users receive a weekly topic (exactly like in this channel), and then draw and submit pictures on that topic. It's kind of like a weekly competition, you can vote on posts, and the most upvoted posts rise to the top.

Could you take a look at the website and let me know what you think?

<https://sketchclub.io/>

It would also be very helpful if you could fill in a very short survey:

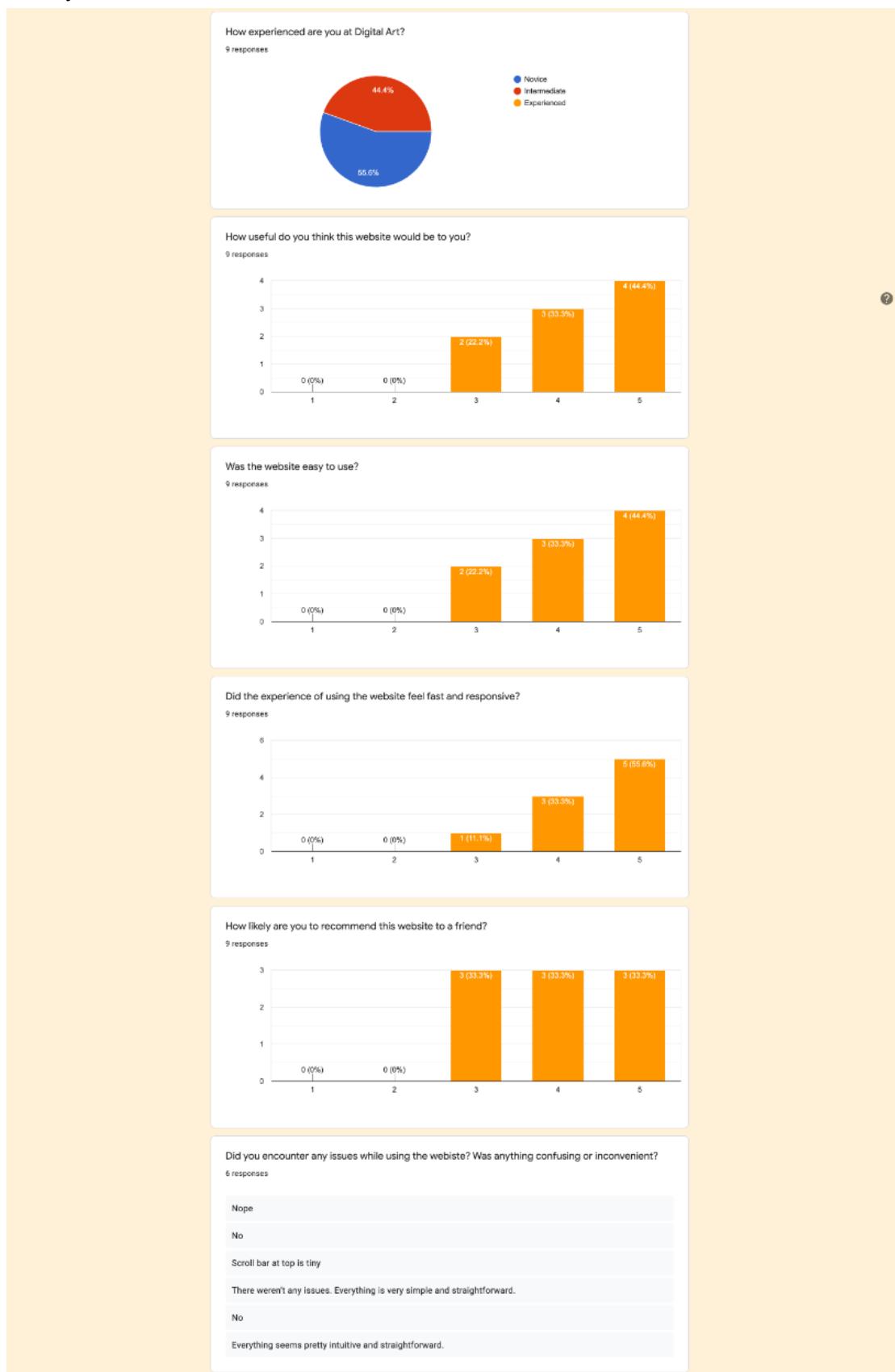
<https://forms.gle/KxPtaZ1WTUGYQAp27>

It won't take more than 5 minutes, and it would really help me out (I'm writing a project report, and I need some user feedback).

Also, feel free to message me if you have any feedback/questions. (edited)

C. Survey Results

Survey results:



D. The Final Project and the Code

The project is deployed at

<https://sketchclub.io/>

(you can try using and browsing the website)

The code is available at:

<https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2020/vxp032>

To run the project locally, clone the repo, then:

1. In “/backend” folder:

Make sure the npm and node are installed correctly.

Set the environment variables (in .env file, using env.sample as an example)

Install the packages:

```
npm i
```

Generate prisma client:

```
npm run generate
```

Generate the database based on the schema:

```
npm run prisma:migrate
```

Seed the database with the initial data:

```
npm run seed
```

Run the project in dev mode:

```
npm run dev
```

Set up the script that updates the post ranks:

```
npm run startRankPostsJob
```

2. In “/frontend” folder run:

Install the packages:

```
npm i
```

Run the project in dev mode:

```
npm run dev
```

3. Visit:

<http://localhost:3020/>