Appetized

# Analysis

## Problem Identification

Lots of young people aren’t cooking for themselves. Home cooking has many benefits like being able to choose your own ingredients to meet your dietary requirements or being able to easily make more healthy meal choices. A possible reason young adults aren’t cooking at home is because they don’t know many recipes that are really tasty, but also easy to make and don’t take several hours to cook. A solution to this would be to create a recipe sharing platform that is tailored to younger people.

I think that this problem would ideally be solved using computers because it fits really well into the client-server topology. Recipes can be stored easily in a database, which can then be accessed remotely over the internet. A client could be also developed to display the recipes and provide the user a simple way of viewing and creating recipes to be uploaded back to the server.

## Computational Methods

The problem is best solved using computational methods because any physical data repositories can only be accessed in person. The use of an online data repository allows anybody in the world to access the data stored in it almost instantly. There are a number of protocols governing traffic over a network that can be used to create an application that takes HTTP requests and responds to them with the requested data or action, this is called a *web API*. The solution requires a user to be able to upload and download recipes quickly and easily, to achieve this, I will develop a website alongside the server that interacts with the backend. The client will provide a user interface (UI) which will make uploading and reading recipes fast and easy.

This is a better solution than a physical repository of recipes, such as a library filled with recipe books, because of there being no physical limitations on how many recipes can be stored, and they can be searched for, filtered and sorted in milliseconds. The use of a database allows storage of millions of recipes without much overhead. An 8 TB hard drive takes up 3.5 inches of space inside a computer, but can store billions of books worth of data.

Another problem with a physical solution would be that young people tend to be less interested in going to a library to get information, seeking it online instead. Having the recipes be accessible on a website would allow them to do this and the use of a native mobile wrapper would allow users to be able to download the client from their phone’s app store.

## Target Audience

The platform would be created for young people who are seeking recipes, so they can cook for themselves. As a result, the solution would need to be engaging for young people. If the site does not have any recipes, people won’t want to use it, so encouraging users to upload their own recipes is essential. This can be achieved by adding social features to the site, which will potentially drive users to want to create the best recipes on the site, so they can get recognition from other users. This will also help to bring users higher quality recipes, because the best ones will be able to be more easily identified because they have more interactions (saves, likes, follows etc.)

When tailoring a product to young people, there are several factors to consider that may pose challenges to the adoption of the product by the target audience. Primarily, young people almost exclusively access the internet with their phones. As a result, I think adopting a mobile-first design philosophy would greatly benefit the usability and quality of the application. Furthermore, the target audience tend to use apps on their phone rather than a web browser, this makes the capability for the product to be released on mobile app stores an essential feature of a solution to this problem.

To summarise, the target audience of the application is young people learning to cook for themselves.

## Existing Solutions

There are already a number of different ways young people get their recipes, and they all seem to have unique strengths and weaknesses. To create an ideal solution to the problem, it will be important to implement each of their benefits, while avoiding whatever pitfalls of the existing solutions.

### Cookbooks

Recipe books are the traditional way to find recipes. They are often written informally which can make the actual process of reading them more inviting and engaging. Cookbooks also tend to be beginner-friendly, and the ingredients and utensils needed are fairly accessible, this allows for people on a lower budget to produce great food. They also contain a range of information about each recipe, such as nutritional information, or how long it takes to prep and cook that helps the reader pick out recipes.

Cookbooks have been around for long enough that they have a perfect balance between readability and detail. They tend to be made up of the same elements:

* Title
* Description
* Metadata such as time of writing
* Categories the recipe falls into (e.g. cuisine, starter or main)
* Cooking and prep time
* Ingredients
* Steps and tips in chronological order

Implementing these will allow the users of the application to represent any recipe. Additionally, allowing users to be able to search for recipes with these categories will make it easier to find the perfect recipe.

However, young people aren’t reading physical books as much as they used to, so even though recipe books might have the perfect format for young people, the fact that it’s a product you have to go and buy might be a deal-breaker for a lot of the target audience. Convenience is an important thing to have in a potential solution and having access to the recipes on a mobile phone is the best to implement it.

Additionally, unless you go to the library, cookbooks can be quite expensive. I think anybody should have free access to recipes, so I don’t plan to monetise access to the recipes themselves.

### Social Media

Some people get their recipes from existing social media apps, I think this is quite a flawed solution to the problem, however, each of the different social media platforms have their own disadvantages.

* Instagram is a photo/video sharing app that is sometimes used to share recipes. The video player is limited in functionality, making it a frustrating experience when cooking. The length of videos on Instagram is between 3 and 60 seconds, which means that intricate recipes are usually rushed through and missing vital details. There is also no way to pause or rewind, which means if you missed something, you have to start the video again. The discoverability of the recipes on Instagram is great, with the use of hashtags. I think hashtags aren’t the best way to categorise recipes, though, so I am going to be implementing more specific parameters to search for recipes. This will include things like cuisine, and the type of meal.
* Facebook is a more traditional social media site, where you can post text alongside some images. Some people also get their recipes from Facebook recipe pages. These pages usually post images or videos like those on Instagram linking to an external recipe website that they run for more detailed instructions. However, there isn’t a way to save or pin a recipe for later. Saving is an essential feature because it can be hard to find a post again. I am going to give users the ability to save recipes, which can be used offline too.

People may be using social media because it is convenient to use it on their phone, through an app store. This, again, shows why it is important to give the website have the capability to be deployed onto those stores.

### Recipe Blogs

Recipe blogs are another popular way to get recipes, they are usually ran by a chef and have just their recipes. These are great for the same reasons as a cookbook, having a personal touch.

A lot of blogs are built with a [WYSIWYG][wysiwyg] (*What You See Is What You Get*, pronounced: “wiz-e-wig”) website editor, these can be good for beginners because they are easy to use, and you can make a website relatively fast. There are however, a large number of drawbacks to these editors. Primarily, WYSIWYG sites often have terrible performance, SEO and user experience on mobile. The slow speed results from sharing the server with a ton of other websites and the generated code itself being poorly optimised. WYSIWYG editors also tend to not encourage mobile-first design, making the user build the desktop site, and then rearrange the elements to better fit mobile. Most of the time, people using WYSIWYG editors don’t teach their users about UX, resulting in a poor experience, especially on phones or tablets. Due to the poor performance and layout, the SEO on generated sites is already going to be low, but there also isn’t a way to implement advanced SEO. Google Search allows websites to give [structured data][structured data] which allows web developers to have their sites appear in Google’s Rich results, this is an important way to drive traffic to a recipe blog. However, on website builders, there isn’t a way to automatically have this data appear on your sites. On Wix, you have to individually add structured data for each page even though its already in the websites CMS, this is an obscure feature and most recipe bloggers won’t pay attention to it.

The blogs are also not centralised, which makes it unfeasible to implement certain features like recipe downloading or social interactions.

Building a recipe blog from scratch can be an expensive and time-consuming process and isn’t accessible to a lot of people. This could be solved with a recipe sharing website that is focused on great SEO and ease of use.

To summarise, all the current ways that are used to get recipes have their own unique disadvantages, which make them all not a great solution to the problem. Recipe books can’t be used anywhere and are costly. Existing social media are not tailored to this purpose, so they lack the specific features needed for a good solution. Recipe blogs aren’t centralised and the tools to create a great user experience aren’t accessible in the context of recipes. My application will incorporate positives from existing solutions while avoiding the disadvantages that they each have.

## Solution

The solution will be a mixture of all the great features of the existing solutions. To achieve this, it will need to be made up of two separate applications, a client and server. The client will provide a user interface to create and interact with the solution’s content. The server will consist of a database and an API which will allow the client to perform CRUD operations on the database, given they are authorised to do so.

Development of the client will focus on accessibility and user experience (UX). Accessibility is a key element of creating a good application, because it makes the solution easier to use for people, especially those who are disabled. The A11Y project provide guidelines to create a more inclusive and representative application. Making accommodations for people who need them will allow a larger group of people to be able to use the solution. It is also important to design a great user experience (UX). Creating an application that is beautiful and natural to use is an important because it will make the site more usable. To ensure that the UX is good, when designing the user interface (UI), I will employ mobile-first design, as I assume that the majority of the applications users will be visiting from a phone. This ensures that the largest group of users gets the best possible experience.

The backend will consist of a GraphQL API, as it allows me flexibility in creating the frontend application. It will give the client the ability to request any data from it, with filtering, sorting and searching capabilities. This will be where the bulk of the site’s authorization will go, therefore I will also be implementing its authentication here as well.

I will create the project using the Agile software development methodology, and therefore will be splitting the process of creating the application into individual sprints. I will decompose the different parts of the application to make more manageable, self-contained sections of the app. I plan to have 5 sprints:

1. Database Schema - Design and implement a database schema that is capable of meeting the success criteria of the project.
2. GraphQL API - Create a GraphQL API with authentication to allow creating read update and delete items in the database, given they are authorized to do so.
3. CDN - Implement a content delivery network to serve images to the client.
4. Email - Give the server the ability to send emails for account creation and recovery.
5. Client - Build out the frontend of the application

## Limitations

* The project will cost a bit of money to run, I plan to use a cloud services provider, like AWS, to host the project. Most cloud providers have a free tier, however, in the long term, there will be some minor costs associated with hosting the solution.
* The project is going to use some tools that I have limited experience using, so this may increase the amount of time that the project will take to complete. I plan to mitigate this by reading the documentation for the tools I am planning to use. This will allow me to have a good understanding of the different dependencies of the project, which allow me to create an optimal solution.
* The scope of the project is fairly large, so it may be unfeasible to finish it within the time constraints. By implementing the core features first, I can ensure that I have a minimum viable product finished before the deadline. I can then iterate on it to build out a more complete solution.
* I will not be able to implement some nice-to-haves, such as a recommendation algorithm, because alone they are similar in scope to the rest of the project. However, there are libraries available that implement these features that can be used, which can make creating a project of this size possible in the timeframe given.

## Requirements

### Software

**Development dependencies:** The solution will be developed in TypeScript. The backend will be using Express, with Apollo Server, to create the API, and Prisma, to access the database. The client will use Svelte, with the SvelteKit meta-framework. An Amazon AWS account will be required for parts of the solution to be able to work, like the CDN or email service. I am going to use an IDE called WebStorm to develop the solution because of its great refactoring tools, which will make the project easier to develop and maintain.

**Runtime dependencies:** Typescript is a superset of JavaScript, so the source code can be transpiled into JS, which will allow the client to be hosted on a web server and run in any browser. The backend can be run using Node.js, a JavaScript runtime. A database will also be needed, and I am going to use MySQL because of its full-text search capabilities.

### Hardware

I am going to be developing the solution on two different systems, my desktop and laptop. I will be using a version control system, Git to ensure that there aren’t conflicts when merging the code between the two systems. My desktop has a 3.6GHz processor and 16 GB of RAM. My laptop also has a 3.6GHz processor but has 8 GB of RAM.

The project won’t require any specific hardware to host, due to it relying on cloud providers. However, a user of the site will need a smartphone or computer with a web browser to access the client. The API on its own has the same features, but requires someone to send requests in the terminal or API client, which is a lot less user-friendly.

### Success Criteria

| Module | Reference | Criteria | Justification | Testing |
| --- | --- | --- | --- | --- |
| Database | 1.1.1 | Database has a table containing recipes. | Allows the storage of recipes so they can be accessed by other users. | Can check with SQL if the table has been created. |
|  | 1.1.2 | The database contains a users table. | Allows for authentication to be developed. | Can check with SQL if the table has been created. |
|  | 1.1.3 | There is a table containing ratings for recipes. | Makes it so the recipes can be better sorted to make users receive better recipes. | Can check with SQL if the table has been created. |
|  | 1.1.4 | A table containing users following other users exists. | Avoids many to many relationships. | Can check with SQL if the table has been created. |
|  | 1.1.5 | A table containing users saving a recipe exists. | Avoids many to many relationships. | Can check with SQL if the table has been created. |
|  | 1.1.6 | A table containing ingredients exists. | Avoids having duplicate values in the recipes table. | Can check with SQL if the table has been created. |
|  | 1.1.7 | There is a table containing images. | Allows images to be owned by a user, and more easily accessible. | Can check with SQL if the table has been created. |
|  | 1.1.8 | There is a table including steps of a recipe. | Avoids storing plain JSON in the database. | Can check with SQL if the table has been created. |
| API | 1.2.1 | Allow the user to create, read, update, and delete their own recipes on the site. | So users can create and view their own content on the site. | Can query the server for these features. |
|  | 1.2.2 | Allow anyone to read, save and unsaved recipes on the site. | So users can access recipes by other users to discover new recipes to cook. | Can query the server for these features. |
|  | 1.2.3 | Allow the user to create, read, update and delete their own account and profile. | In the case that users want to change something about their profile. | Can query the server for these features. |
|  | 1.2.4 | Allow users to access other user’s profiles. | So users can find recipes based on the author more easily. | Can query the server for these features. |
| Email | 1.4.1 | A confirmation email is sent to a user on creation of their account. | To make sure that the user has entered their email correctly and that it exists. | Create an account and check emails, then click the link, check the database to see if the email is verified. |
|  | 1.4.2 | Users can request an email containing a link that will allow them to reset their password. | If the user forgets their password or wants to reset it. | Can click the reset password button, and see if the email is sent, then change the password, attempt to login with the new password. |
| Client | 2.1.1 | Recipes can be displayed. | Essential feature of the client. | Try to load a recipe. |
|  | 2.1.2 | Recipes can be edited. | Maybe the user wants to make their recipe more detailed or tastier. | Try to edit a recipe, then load it to see if it has changed. |
|  | 2.1.3 | User account can be displayed. | To make finding recipes by a certain person easier. | Try to load a user’s profile. |
|  | 2.1.4 | User profile can be edited. | If the user wants to change their details. | Try to edit my profile and see if it changes. |
|  | 2.1.5 | Website is accessible. | So a wider range of people can easily use the site. | Use Google Chrome Lighthouse to see the accessibility score. |
|  | 2.1.6 | Website has good SEO. | To make the recipes discoverable. | Use Google Chrome Lighthouse to see the SEO score. |
|  | 2.1.7 | Website is performant. | To make the site work well on mobile, ensuring it has good performance is important. | Use Google Chrome Lighthouse to see the performance score. |
| CDN | 2.2.1 | Users can upload images to the site | Images make the recipes more easy to follow. | Try to upload an image and see if it shows up in the cloud storage providers GUI. |
|  | 2.2.2 | The location of uploaded images are stored in the site’s database. | So the client can get all of a recipes images. | Can check if the uploaded image is in the images table with SQL. |
|  | 2.2.3 | Images uploaded can be accessed by the client. | So they can be displayed conveniently to the user. | Can try to load a recipe containing an image. |

# Sprint 1

This first sprint will consist of designing and then creating the database for the application. The database is the backbone of the solution, so it will need to be thoughtfully designed. I am going to make sure that the database follows normalization rules, to ensure that create, read, update and delete (CRUD) operations can take place without breaking other parts of the database.

I am going to create the schema and perform CRUD operations on the database with an object relational mapper (ORM). This will allow me to access the database in a type-safe way, which will decrease the potential risk of bugs. The data put into the database will also be sanitised, which will reduce the risk of critical vulnerabilities like SQL injection. The ORM I am going to be using is called Prisma.

## Design

Figure 1: Entity relationship diagram of the database

This is an entity relationship diagram of what the database will look like. I am going to use join tables to make sure that the database is normalised, so CRUD operations can take place without creating redundancies. There are a lot of different types of data needed for the solution, so I used decomposition to ensure that the data is appropriately linked together and split up logically into tables.

### Initializing the project

The first step in creating the database schema is to initialise the project.

$ npm init -y  
$ pnpm install -D prisma typescript prettier ts-node @types/node  
$ tsc --init  
$ npx prisma init

This will initialise the project in the current directory, and create a package.json and a tsconfig.json, as well as install the necessary dependencies for this sprint. A directoryUser will also be created called prisma prisma which will contain a file called schema.prisma. This is where I will be writing the schema for the database.

generator client {  
 provider = "prisma-client-js"  
 previewFeatures = ["fullTextSearch", "fullTextIndex"]  
}  
  
datasource db {  
 provider = "mysql"  
 url = env("DATABASE\_URL")  
}

This is the contents of schema.prisma, generated by the $ npx prisma init command. By adding different types of data to the file, a model can be created which Prisma can use to generate a database. Prisma will also create a database client that can be used to perform type safe CRUD operations. The client generated by Prisma can be used by the GraphQL API to fetch and store data for use in the client like this:

const myProfile = prisma.users.findUnique({ where: { id: "123" } }); // Gets user with ID "123"

The code above would get the user with the ID of ‘123’ by executing the SQL statement:

SELECT \* FROM users WHERE id = "123"

### Important Variables and Data Structures

There will only be one variable created in this whole sprint, prisma, it will be the database client. This variable will be exported, so it can be accessed in other file.

The schema itself also falls under this header. There will be quite a few tables:

* users: This table will contain all the users that are registered in the application.
  + id: This is the primary key for the table. (Unique string)
  + name: This is the name of the user. (Optional string)
  + email: This is the email of the user. (Unique string)
  + password: This is a hash of the user’s password. (String)
  + profilePicture: This is the URL of the user’s profile picture. (Optional Image)
  + createdAt: This is the date the user was created. (Date)
  + emailVerified: This is a boolean that indicates whether the user has verified their email. (Boolean)
  + recipes: This is a list of recipes that the user has created. (List of Recipe)
  + saved: This is a list of recipes that the user has saved. (List of Recipe)
  + following: This is a list of users that the user is following. (List of User)
  + followers: This is a list of users that are following the user. (List of User)
  + iat: This is the date that the most recent valid authentication token was issued. (Date)
* recipes: This table will contain all the recipes that are uploaded by the users.
  + id This is the primary key for the table. (Unique string)
  + name This is the title of the recipe. (String)
  + author This is the account that created the recipe. (User)
  + authorId This is the ID of the account that created the recipe. (String)
  + description This is the description of the recipe. (Optional string)
  + image This is an image of the recipe. (Optional Image)
  + imageId This is the ID of the image. (Unique string)
  + createdAt This is the date the recipe was created. (Date)
  + steps This is a list of steps that make up the recipe. (List of Step)
  + category This is the category (For example: breakfast, starter, etc.). (Optional string)
  + cuisine This is the cuisine of the recipe. (Optional string)
  + ingredients This is a list of ingredients that are used in the recipe. (List of Ingredient)
  + cookTime This is the amount of time it takes to cook the recipe. (Optional integer)
  + prepTime This is the amount of time it takes to prepare the recipe. (Optional integer)
  + savedBy This is a list of users that have saved the recipe. (List of User)
* ingredients: This table will contain all the ingredients that are used in the recipes.
  + id This is the primary key for the table. (Unique string)
  + name This is the name of the ingredient. (String)
  + quantity This is the quantity of the ingredient. (String)
  + recipe This is the recipe that the ingredient is used in. (Recipe)
  + recipeId This is the ID of the recipe that the ingredient is used in. (String)
* images: This table will contain all the images that are used across the solution.
  + id This is the primary key for the table. (Unique string)
  + url This is the URL of the image. (String)
  + recipe This is the recipe that the image is used in. (Optional recipe)
  + step This is the step that the image is used in. (Optional step)
  + profile This is the profile that the image is used in. (Optional User)
  + userId This is the ID of the user’s profile that the image is used in. (Optional string)
* steps: This table will contain each step of the recipe.
  + id This is the primary key for the table. (Unique string)
  + name This is the name of the step. (Optional string)
  + createdAt This is the date the step was created. (Date)
  + content This is the content of the step. (String)
  + image This is an image of the step. (Optional Image)
  + imageId This is the ID of the image. (Optional string)
  + recipe This is the recipe that the step is used in. (Recipe)
  + recipeId This is the ID of the recipe that the step is used in. (String)

### Validation

The database won’t be public facing because it contains sensitive information, which is why I am developing the API in the first place. As a result validation is not strictly necessary, however, I will add some basic validation, such as unique and required constraints. This is because they are easier to implement on the data layer than on the application layer.

In Prisma, constrains are added in the schema by writing attributes. Attributes sit at the end of the field definition. For example, unique constraints are added by writing @unique after the field.

### Testing

I will test the Prisma client by performing CRUD operation on each table, to make sure that each of them are configured correctly. This can be achieved by creating unit tests, with a testing framework. I am choosing to use Mocha for this purpose because it is simple and ticks all the necessary boxes for a project of this scale. This allows me to automatically perform tests programmatically, and log the result to the console. I can write tests in the test/ directory, and run them with $ npm run test.

## Development

I ran the previously mentioned commands, and it outputted the following:

$ npm init -y

$ pnpm install -D prisma typescript prettier ts-node @types/node

$ tsc --init

$ npx prisma init  
  
✔ Your Prisma schema was created at prisma/schema.prisma  
 You can now open it in your favorite editor.  
  
Next steps:  
1. Set the DATABASE\_URL in the .env file to point to your existing database. If your database has no tables yet, read https://pris.ly/d/getting-started  
2. Set the provider of the datasource block in schema.prisma to match your database: postgresql, mysql, sqlite, sqlserver or mongodb (Preview).  
3. Run prisma db pull to turn your database schema into a Prisma schema.  
4. Run prisma generate to generate the Prisma Client. You can then start querying your database.  
  
More information in our documentation:  
https://pris.ly/d/getting-started

As you can see, npx prisma init has outputted the next steps to follow when creating the schema. It has also created a file named .env. A .env file contains environmental variables, such as API keys, or in this case, the database’s connection URL. I also created a file named .gitignore, which tells the version control system, git, to ignore the specified files. I included .env in the .gitignore because the database URL needs to stay private because publicising it could lead to the database being compromised. Accidentally committing the file to the version history could make it easier to leak sensitive information. I also included the .idea/ and node\_modules/ directories in the .gitignore file. .idea/ contains configuration files specific to the local instance of the IDE I’m using. node\_modules/ contains the dependencies of the project, the reason I am telling git to ignore them is because they will be built locally from the source code when first getting the project running locally, so including them is unnecessary.

I also needed to import the prisma client the TypeScript project:

export const prisma = new PrismaClient();

Next, I set the database provider in the schema.prisma, I set this to “mysql”. I also added the line previewFeatures = ["fullTextSearch", "fullTextIndex"] to enable the use of full text search within Prisma.

Finally, I wrote the schema in the same file, here it what I wrote:

generator client {  
 provider = "prisma-client-js"  
 previewFeatures = ["fullTextSearch", "fullTextIndex"]  
}  
  
datasource db {  
 provider = "mysql"  
 url = env("DATABASE\_URL")  
}  
  
model User {  
 id String @id @default(uuid())  
 name String?  
 username String @unique  
 email String @unique  
 password String  
 profilePicture Image?  
 createdAt DateTime @default(now())  
 emailVerified Boolean  
 recipes Recipe[] @relation("author")  
 saved Recipe[] @relation("save")  
 following User[] @relation("follow")  
 followers User[] @relation("follow")  
 iat Int? @default(0)  
  
 @@fulltext([username])  
 @@fulltext([username, name])  
}  
  
model Image {  
 id String @id @default(uuid())  
 url String  
 recipe Recipe?  
 step Step?  
 profile User? @relation(fields: [userId], references: [id])  
 userId String?  
}  
  
model Recipe {  
 id String @id @default(uuid())  
 name String  
 author User @relation("author", fields: [authorId], references: [id])  
 authorId String  
 description String?  
 image Image? @relation(fields: [imageId], references: [id])  
 imageId String? @unique  
 createdAt DateTime @default(now())  
 steps Step[]  
 category String?  
 cuisine String?  
 ingredients Ingredient[]  
 cookTime Int?  
 prepTime Int?  
 savedBy User[] @relation("save")  
  
 @@fulltext([name, description])  
 @@fulltext([name, description, cuisine, category])  
}  
  
model Step {  
 id String @id @default(uuid())  
 name String?  
 createdAt DateTime @default(now())  
 content String  
 image Image? @relation(fields: [imageId], references: [id])  
 imageId String? @unique  
 recipe Recipe? @relation(fields: [recipeId], references: [id])  
 recipeId String?  
  
 @@fulltext([name, content])  
}  
  
model Ingredient {  
 id String @id @default(uuid())  
 name String  
 quantity String  
 recipe Recipe? @relation(fields: [recipeId], references: [id])  
 recipeId String @unique  
  
 @@fulltext([name])  
}

When the command npx prisma db push is run, Prisma will generate the SQL needed to implement this schema in MySQL. Next, it will execute the SQL and generate the database client:

✔ Generated Prisma Client (3.8.1 | library) to ./node\_modules/@prisma/client in 224ms

## Testing

Initially, I wanted to check that the database had been successfully created. I connected to the database using the mysql command line tool, and I checked that the database had been created:

$ mysql  
mysql> show databases;  
+--------------------+  
| Database |  
+--------------------+  
| information\_schema |  
| mysql |  
| performance\_schema |  
| railway |  
| sys |  
+--------------------+  
5 rows in set (0.18 sec)

Now that I knew the database was successfully created, it was now a good time to write unit tests, to ensure that Prisma client behaved in the intended way. Before performing unit testing was possible, I would need to implement the systems necessary to perform unit tests on the database. The first step to developing tests is to install a testing framework for the project. I installed Mocha by executing the command:

$ pnpm install -D mocha @types/mocha chai @types/chai ts-node

Next, I had to create a file for the unit tests to be contained in, I did this by running:

$ mkdir ./test/  
$ touch ./test/test.ts

I started to write different tests for the database. Here are some of the tests for user for example:

import { prisma } from "../src/app";  
import assert from "assert";  
import { expect } from "chai";  
  
describe("prisma", () => {  
 describe("user", () => {  
 describe("CRUD", () => {  
 it("should add a user to the database.", async function() {  
 await prisma.user.create({  
 data: {  
 name: "name",  
 email: "email",  
 username: "username",  
 password: "password",  
 },  
 });  
 });  
 it("should read the created user from the database", async function() {  
 const user = await prisma.user.findUnique({  
 where: {  
 email: "email",  
 },  
 });  
 expect(user?.name).to.equal("name");  
 });  
 it("should update the created user", async function() {  
 const user = await prisma.user.update({  
 where: {  
 email: "email",  
 },  
 data: {  
 name: "name2",  
 },  
 });  
 expect(user?.name).to.equal("name2");  
 });  
 it("should delete the user", async function() {  
 await prisma.user.delete({  
 where: {  
 email: "email",  
 },  
 });  
 });  
 });  
 });  
 // Snip ✂  
});  
  
// Snip ✂

Inside the file package.json, I could create a test script (Cannot include “Snip ✂“, in this file, because JSON does not support comments, however the file contents have still been shortened):

{  
 "scripts": {  
 "test": "mocha -r ts-node/register test/test.ts"  
 }  
}

Now, because of the script in package.json, I can run:

$ yarn test  
  
# Output  
yarn run v1.22.11  
warning ../package.json: No license field  
$ mocha -r ts-node/register test/test.ts  
  
 prisma  
 user  
 CRUD  
 ✔ should add a user to the database. (120ms)  
 ✔ should read the created user from the database  
 ✔ should update the created user  
 ✔ should delete the user  
 # Snip ✂  
  
 41 passing (4214ms)

As you can see, there are no failed tests, so the database is working correctly.

## Evaluation

This sprint has been successfully completed, here are the success criteria that have been finished:

| Reference | Criteria | Testing | Completion |
| --- | --- | --- | --- |
| 1.1.1 | Database has a table containing recipes. | All tests passed | ✅ |
| 1.1.2 | The database contains a users table. | All tests passed | ✅ |
| 1.1.3 | There is a table containing ratings for recipes. | All tests passed | ✅ |
| 1.1.4 | A table containing users following other users exists. | All tests passed | ✅ |
| 1.1.5 | A table containing ingredients exists. | All tests passed | ✅ |
| 1.1.6 | A table containing users saving a recipe exists. | All tests passed | ✅ |
| 1.1.7 | There is a table containing images. | All tests passed | ✅ |
| 1.1.8 | There is a table including steps of a recipe. | All tests passed | ✅ |
| 2.2.2 | The location of uploaded images are stored in the site’s database. | N/A | ❎ |

All of this sprint’s finished success criteria are similar, so I will discuss them as one. They have been successful, as the database is now up and running along with the database client. Now the database has been created, it will be possible to start creating the GraphQL API. 2.2.2 is now able to be completed, however I want to implement this later on, because it will be more appropriate to develop alongside the other parts of the CDN.

### Maintainability

The database has been created in a way where performing CRUD operations won’t break dependent columns. Tables and rows can also be added or deleted without having to reset the database, so it is maintainable to a reasonable extent. If the database is further normalised, it would be more maintainable, however, it would make it more difficult to use, so I think that I have struck a good balance between usability and maintainability.

### Limitations

From my testing the database seems quite slow and more advanced Postgres features like indexes are not yet possible with Prisma. I may have to later create migration files to be able to speed up the database to a more acceptable level. Additionally, the database on its own is not a good solution to the problem, it has no authorization or authentication, and its validation is very limited, because I plan to implement those on the application tier, rather than the data tier.

# Sprint 2

I am going to spend the second sprint developing the API. I will need to develop authentication and authorization, as well as make important decisions surrounding the structure of the client, as it will rely on this sprints work heavily.

## Design

An application programming interface (API) is a set of methods and protocols that allow two applications to interact with each other. As the solution is using the client-server architecture, the API will be used to allow the two different sides to communicate with each other. This is an essential part of the solution, because it gives the client access to read or edit the information stored in the database. It also allows authorization to be more easily implemented, because the API will be able to issue tokens to the client, and the client will be able to use these tokens to authenticate.

There are two main ways to implement an API, one is using GraphQL, and the other is using REST. A key difference between REST and GraphQL is that REST is bade up of several distinct endpoints, each serving a different purpose. GraphQL is instead designed to be a single endpoint, which is a combination of multiple queries and mutations. These queries and mutations are then accessed by the client with the GraphQL query language. A good analogy I use to describe them is that REST is a restaurant with a strict menu and GraphQL is a buffet. Therefore, GraphQL a very powerful way to access data from the database, as it allows for more complex queries to be made, and allows for the client to be more flexible. The main advantages of REST over GraphQL are that it is easier to implement, and has better error handling. However, I am going to go with GraphQL, as it is more flexible, giving the client more control over the data it is accessing. This will ensure that unneeded data is not being sent to the client, which will reduce the bandwidth used. It also ensures that loading times on the client are reduced, and the database is not being accessed too often.

A GraphQL API is made up of a schema, which contains different types of data available to the client. The schema is made up of two parts the type definitions and the resolvers. The type definitions describe the different fields which are available to the client, and the resolvers are functions that process the requests made for those fields.

There are two different ways GraphQL is being implemented in TypeScript, one is using the schema-first approach, and the other is using a code-first approach. Schema driven development is the first way GraphQL was implemented, and is more easily readable and writable. Code-first development is more difficult to read and write, but tends to be more type safe. I am going to go with the schema-first approach, because the extra time spent on writing the more type safe  
code-first schema could instead be used on writing type declarations, which achieves the same purpose, but is more readable.

The GraphQL server I am going to use is Apollo, which is a schema-first implementation of GraphQL in JavaScript. Apollo Server has the added benefit of being able to be used with Express.js, a popular Node.js framework. This will allow me to manage the requests and responses sent through the API more easily, allowing for me to use cookies for authentication.

In Apollo, there is the option to pass an object to the context parameter, which is used to pass data to the resolvers. I will use this to give resolvers the authentication status of the user, as well as the status of the request and the response. This will allow the solution a more flexible way of handling different kinds of requests like authentication, and I will use this to great effect.

### Schema

The schema is a very important part of the solution. The whole client will rely on it to hydrate the page with data, and make edits. As a result, I am going to be very careful with the planning of it because making breaking changes to it later down the line will cause the client to need to be updated in several places. Additionally, the schema won’t match the database exactly because the database contains sensitive information, such as password hashes. Therefore, I am going to have to design a schema and consider each field carefully.

#### Type Definitions

The type definitions are the different types of data that are available to the client. I will need to create a type for each table in the database. Additionally, I will need to add two extra types, Query and Mutation. The query and mutation types define the actual requests and responses that the API will be able to make. A query is a request for data, which doesn’t change any data stored on the server. A mutation, on the other hand, is a request to change data on the server.

The first type definition I am going to write is the User type. This type will contain publicly available information pertaining to a user of the site. As I mentioned, certain database columns will not be included in the API, such as the user’s password hash, their email address, and the expiration date of their tokens. This is private information, and therefore, I am not going to include it in the API. The columns that I am going to include are:

* id: The ID of the user. This will be useful for requesting data about the user in other queries.
* username: The username of the user. This will be used to log in to the site, and can be used in place of the ID to request data about the user.
* name: The name of the user. This is an optional field and is used to display the user’s name in a more personal way.
* profilePicture: This will contain the image used as their profile picture. This is also an optional field.
* createdAt: The date and time that the user was created. This can be used to sort users by their creation date.
* recipes: This will contain the recipes that the user has created.
* savedRecipes: This will contain the recipes that the user has saved. This will be used on the client to download recipes in local storage. Having this information in the database will make it easier to sync this across devices.
* following: This will contain the users that the user is following.
* followers: This will contain the users that are following the user.

The fields that return an array of data, recipes, savedRecipes, following and followers will take parameters when they are called. These parameters will be used to perform cursor based pagination on the data that is returned. The parameters will be take, from, and sort. The take parameter will be used to specify how many items to return. The from parameter will take the ID that the data returned should start from. The sort parameter will take an object that tells the server how to sort the data. Cursor based pagination is a very common way of paginating data, and I am going to use it because it is easy to implement, and for the number of records being used in this solution, it is the most efficient way of paginating. The recipes and savedRecipes will return the type RecipeResponse, which is a union of Recipe and Error. The following and followers will return the type UserResponse, which is a union of User and Error.

The next type definition is going to be Image. This will contain the information needed to display an image on the client. The fields that I am going to include are:

* id: The ID of the image. This will be used to request data about the image in other queries.
* url: The URL of the image. This will be used to display the image on the client.
* uploader: The user that uploaded the image. This is only used when the image is a profile picture.
* recipe: The recipe that the image is associated with. This is only used when the image is used in a recipe.
* step: The step that the image is associated with. This is only used when the image is used in a specific step in a recipe.

The resolvers for Image will not be implemented in this sprint. This is because image handling is being implemented later on in the project. However, I think that creating the type definition for Image should be done now, so that I don’t have to implement breaking changes in the schema later on.

The next type definition is going to be Recipe. This will contain the information about a recipe on the site. The fields that I am going to include are:

* id: The ID of the recipe. This will be used to request data about the recipe in other queries.
* name: This is going to be the title of the recipe.
* author: The user that created the recipe.
* description: This is going to be the description of the recipe.
* image: This will contain the image that is display with the recipe.
* createdAt: The date and time that the recipe was created. This can be used to sort recipes by their creation date.
* steps: This will contain the steps that the recipe has.
* category: This will contain the category that the recipe belongs to (e.g. “Dinner”, “Starter”).
* cuisine: This will contain the cuisine that the recipe belongs to (e.g. “Italian”, “American”).
* ingredients: This will contain the ingredients that the recipe uses.
* cookTime: This will contain the amount of time it takes to cook the recipe.
* prepTime: This will contain the amount of time it takes to prepare the recipe.
* savedBy: This will contain the users that have saved the recipe.

The recipe type also has some fields that return an array, savedBy, ingredients and steps. These fields will take parameters when they are called. These parameters will be take, from, and sort. These parameters act in the same way as in the User type. The savedBy field will return a UserResponse type, which is a union of User and Error . The ingridients will return a IngredientResponse type, which is a union of Ingredient and Error. The steps will return a StepResponse type, which is a union of Step and Error.

The next type definition will be Step. This will contain the data used in each step of a recipe. The fields that I am going to include are:

* id: The ID of the step. This will be used to request data about the step in other queries.
* name: This will be the name of the step.
* createdAt: The date and time that the step was created. This can be used to sort steps by their creation date.
* content: This will contain the content of the step.
* image: This will contain the image that is associated with the step.
* recipe: This will contain the recipe that the step is associated with.

The next type definition will be Ingredient. This will contain the data used in each ingredient of a recipe. The fields that I am going to create are:

* id: The ID of the ingredient. This will be used to request data about the ingredient in other queries.
* name: This will be the name of the ingredient.
* quantity: This will be the quantity of the ingredient.
* recipe: This will contain the recipe that the ingredient is associated with.

Next, I will create the Error type. This will be used to return errors to the client. The fields that I am going to include are:

* code: This will contain the error code.
* message: This will contain the error message.

Next, I will create the Query type. This will contain all the queries that the client can make. The fields that I am going to include are:

* user: given an ID or username, this will return the user with that ID or username. If neither of these parameters are given, this will return the currently logged-in user. This will return a UserResponse type, which is a union of User and Error.
* image: given an ID, this will return the image with that ID. This will return a ImageResponse type, which is a union of Image and Error.
* recipe: given an ID, this will return the recipe with that ID. This will return a RecipeResponse type, which is a union of Recipe and Error.
* step: given an ID, this will return the step with that ID. This will return a StepResponse type, which is a union of Step and Error.
* ingredient: given an ID, this will return the ingredient with that ID. This will return a IngredientResponse type, which is a union of Ingredient and Error.
* users: this will return all the users in the database if no parameters are given. The parameters will be query, take, from, and sort. These allow the client to search for users by their name or username, and perform pagination.
* recipes: this will return all the recipes in the database if no parameters are given. The parameters will be query, take, from, and sort. These allow the client to search for recipes, and perform pagination.
* steps: this will return all the steps in the database if no parameters are given. The parameters will be query, take, from, and sort. These allow the client to search for steps, and perform pagination.
* ingredients: this will return all the ingredients in the database if no parameters are given. The parameters will be query, take, from, and sort. These allow the client to search for ingredients, and perform pagination.

Next, I will create the Mutation type. This will contain all the mutations that the client can make. The fields that I am going to include are:

* createUser: this will create a new user. The parameters will be user, which will be the type CreateUserInput, and image which will be of the type ImageInput. This will return a UserResponse type, which is a union of User and Error.
* loginUser: this will log in a user. The parameters will be usernameOrEmail andpassword. This will return a UserResponse type, which is a union of User and Error.
* logoutUser: this will log out the currently logged-in user. This will return a bool.
* editUser: this will edit the currently logged-in user. The parameters will be user, which will be the type EditUserInput, and image which will be of the type ImageInput. This will return a UserResponse type, which is a union of User and Error.
* deleteUser: this will delete the currently logged-in user. This will return a bool.
* followUser: this will follow the user with the given ID. The parameters will be id. This will return a UserResponse type, which is a union of User and Error.
* unfollowUser: this will unfollow the user with the given ID. The parameters will be id. This will return a UserResponse type, which is a union of User and Error.
* createRecipe: this will create a new recipe. The parameters will be recipe, which will be the type CreateRecipeInput, and image which will be of the type ImageInput. This will return a RecipeResponse type, which is a union of Recipe and Error.
* editRecipe: this will edit the recipe with the given ID. The parameters will be id, recipe, which will be the type EditRecipeInput, and image which will be of the type ImageInput. This will return a RecipeResponse type, which is a union of Recipe and Error.
* deleteRecipe: this will delete the recipe with the given ID. The parameters will be id. This will return a bool.
* saveRecipe: this will save the recipe with the given ID. The parameters will be id. This will return a bool.
* unsaveRecipe: this will un save the recipe with the given ID. The parameters will be id. This will return a bool.
* createStep: this will create a new step. The parameters will be step, which will be the type StepInput, and image which will be of the type ImageInput. This will return a StepResponse type, which is a union of Step and Error.
* editStep: this will edit the step with the given ID. The parameters will be id, step, which will be the type StepInput, and image which will be of the type ImageInput. This will return a StepResponse type, which is a union of Step and Error.
* deleteStep: this will delete the step with the given ID. The parameters will be id. This will return a bool.
* createIngredient: this will create a new ingredient. The parameters will be ingredient, which will be the type IngredientInput. This will return a IngredientResponse type, which is a union of Ingredient and Error.
* editIngredient: this will edit the ingredient with the given ID. The parameters will be id, ingredient, which will be the type IngredientInput. This will return a IngredientResponse type, which is a union of Ingredient and Error.
* deleteIngredient: this will delete the ingredient with the given ID. The parameters will be id. This will return a bool.

In some GraphQL APIs, there is an additional Subscription type. These allow the client to make websocket connections to the server. I will not be implementing this in this project, because the real-time functionality is not needed for this project.

Next in the type definitions are the different input types, these are used to define the input parameters for the other type definitions.

* CreateUserInput: this is the input type for the createUser mutation. It has the following fields:
  + name: this is the name of the user.
  + email: this is the email of the user.
  + username: this is the username of the user.
  + password: this is the password of the user.
* EditUserInput: The username and password will not be editable by the user at this stage of the project. I am going to add the ability to do this when adding the email functionality.
  + name: this is the name of the user.
  + username: this is the username of the user.
* CreateRecipeInput: this is the input type for the createRecipe mutation. It has the following fields:
  + name: this is the title of the recipe.
  + description: this is the description of the recipe.
  + category: this is the category of the recipe.
  + cuisine: this is the cuisine of the recipe.
  + cookTime: this is the cook time of the recipe.
  + prepTime: this is the prep time of the recipe.
* EditRecipeInput: this is the input type for the editRecipe mutation. It has the following fields:
  + name: this is the title of the recipe.
  + description: this is the description of the recipe.
  + category: this is the category of the recipe.
  + cuisine: this is the cuisine of the recipe.
  + cookTime: this is the cook time of the recipe.
  + prepTime: this is the prep time of the recipe.
* StepInput: this is the input type for the createStep mutation. It has the following fields:
  + name: this is the name of the step.
  + content: this is the actual message of the step.
* IngredientInput: this is the input type for the createIngredient mutation. It has the following fields:
  + name: this is the name of the ingredient.
  + quantity: this is the quantity of the ingredient.
* UserSort: this is used to sort an array of users. It will contain all the fields in the User type with an additional \_count field. Each of the fields will either be of the type Direction or a different kind of sorting input.
* RecipeSort: this is used to sort an array of recipes. It will contain all the fields in the Recipe type with an additional \_count field. Each of the fields will either be of the type Direction or a different kind of sorting input.
* StepSort: this is used to sort an array of steps. It will contain all the fields in the Step type with an additional \_count field. Each of the fields will either be of the type Direction or a different kind of sorting input.
* IngredientSort: this is used to sort an array of ingredients. It will contain all the fields in the Ingredient type with an additional \_count field. Each of the fields will either be of the type Direction or a different kind of sorting input.

Next are in the type definitions the different union types created:

* UserResponse: this is the union type for the User type. It will contain either the User type or the Error type.
* RecipeResponse: this is the union type for the Recipe type. It will contain either the Recipe type or the Error type.
* StepResponse: this is the union type for the Step type. It will contain either the Step type or the Error type.
* IngredientResponse: this is the union type for the Ingredient type. It will contain either the Ingredient type or the Error type.

Finally, the Direction enum will be created, it will have a value of ASC and DESC.

#### The resolvers

Resolvers are functions that are run when a request is made to the API. They are passed the arguments in the request as parameters, as well as the user’s authentication information. This information can then be used to resolve any requests that the user sends.

The resolvers for authentication are important to get right, as they are the ones that the whole project’s security rely on. Here are the mutations used on the site:

loginUser (takes a username or email, and password as parameters) will firstly check weather the user is trying to log in with an email or a username. It will do this by checking their input against a simple regular expression that checks for characters that are not allowed in usernames, but are allowed in emails. The result of this check will be used to determine what will be passed into the database query. This query will then find a user with that email or username. If no user is found, or their email isn’t verified, an error will be returned. If the user is found, the password attempt will be verified against the hash in the database. If the password is correct, the user will be issued two tokens, an access token and a refresh token, these can be used to authenticate future requests by them. If the password is incorrect, an error will be returned.

logoutUser will first, calculate the current number of seconds since the epoch, a UNIX timestamp. This will then be added to the database as the new value for iat. This will ensure that the token is no longer valid. Next, the user’s access token and refresh token will be cleared, and then the function will return true.

createUser will run a number of checks on the user’s input. Returning an error if the data provided is invalid. Next, the database will be checked to see if the username or email already exists. If it does, an error will be returned to the client. The user’s password will then be hashed using argon2, and then the user is stored in the database. This will be changed a lot in future sprints to add email verification and image upload.

editUser will take the user’s input and check that its valid and won’t break any unique constants. This data will then be added to the database.

deleteUser will delete the logged-in user from the database. When adding email verification, I may require the user to verify their email before they can delete their account.

followUser will take the user’s input and check that the user is logged in. If they are, and they are not trying to follow themselves, the user will be added to the database as a follower of the user they are trying to follow.

unfollowUser will take the user’s input and check that the user is logged in. If they are, they will be removed from the database as a follower of the user they are trying to unfollow.

createRecipe will take a user’s input of a recipe, check that they are logged-in, and if they are, it will create the recipe in the database.

editRecipe will take a recipe’s ID and what changes they want to make to the recipe. If they are logged in and own the recipe, it will edit the recipe in the database.

deleteRecipe will take a recipe’s ID and check that the user is logged in. If they are, and they own the recipe, it will be deleted from the database.

saveRecipe will take a recipe’s ID and check that the user is logged in. If they are, they will save the recipe. This will allow user’s to save their own recipes, because saving a recipe has the utility of downloading it, so allowing them to add to the save counter is a good idea.

unsaveRecipe will take a recipe’s ID and check that the user is logged in. If they are, they will un save the recipe from the user in the database.

createStep will take a recipe’s ID and a step’s content. If the user is logged in and owns the recipe, it will add that step to the database.

editStep will take a step’s ID and what changes they want to make to the step. If the user is logged in and owns the step, it will edit the step in the database.

deleteStep will take a step’s ID and check that the user is logged in. If they are, and they own the step, it will be deleted from the database.

createIngredient will take a recipe’s ID and an ingredient’s content. If the user is logged in and owns the recipe, it will add that ingredient to the database.

editIngredient will take an ingredient’s ID and what changes they want to make to the ingredient. If the user is logged in and owns the ingredient, it will edit the ingredient in the database.

deleteIngredient will take an ingredient’s ID and check that the user is logged in. If they are, and they own the ingredient, it will be deleted from the database.

The different queries discussed in the type definitions are self-explanatory. As a result, I have not included a description of their function here.

There are also some miscellaneous resolvers that don’t resolve a field:

* UserResponse will determine weather a response is of the type User or Error.
* RecipeResponse will determine weather a response is of the type Recipe or Error.
* StepResponse will determine weather a response is of the type Step or Error.
* IngredientResponse will determine weather a response is of the type Ingredient or Error.
* Direction will turn the directions from an enum into a string, which can then be read by the database.

### Context

As previously mentioned, the context is the object that is passed to every resolver. It contains the user’s ID, if they are logged in, and Express’s request and response objects. The context is created with a function that I am going to create called context. This function will be run each time a request is received. The first thing it will do is check the user’s access token. If the user has an access token, the context will check the database to see if it is expired. If the access token is still valid, the context will return the user’s ID. If the access token is expired, the context will check the refresh token. If the refresh token is valid, the context will refresh the access and refresh tokens, and then return the user’s ID. If the refresh token is invalid, the context will return a null user ID. This will give the resolvers the ability to know if the user is logged in or not.

### Validation

The validation of the user’s input is done in the resolvers. Each resolver will have different validation rules, so they are implemented on a case-by-case basis. The bulk of the validation will go in the createUser and editUser resolvers. They will both have the same rules, here they are:

1. The username must be between 3 and 20 characters long.
2. The username must only contain letters, or hyphens.
3. The username must not start or end with a hyphen.
4. The username must not contain two consecutive hyphens.
5. The email must contain an @ symbol.
6. The email must not be greater than 100 characters long.
7. The password must be greater than 8 characters long.
8. The password must be less than 100 characters long.
9. The username must be supplied.
10. The email must be supplied.
11. The password must be supplied.
12. The email must be unique.
13. The username must be unique.

### Testing

I will test this sprint with a combination of unit tests and integration tests. The unit tests will take place in the same way as the database client, in the test/test.ts file. The integration tests will be done by hand using Apollo Studio, a GraphQL client.

## Development

The first thing to do was to create a new branch for the development of the project. I will name it dev.

$ git checkout -b dev

This will ensure that changes can be made to the project without affecting the master branch.

Next, I am going to install the dependencies relevant to this sprint. These are:

* apollo-server-express - The GraphQL server.
* jsonwebtoken - The library that will be used to create and verify access tokens.
* argon2 - The library that will be used to hash the password.
* cookie-parser - The library that will be used to populate the request object with an object containing the user’s cookies.
* cors - The library that will allow me to create a CORS policy.
* dotenv - The library that will be used to read the environment variables.
* express - A web framework for Node.js.
* graphql - The JavaScript implementation of GraphQL.
* graphql-tag - A library that will be used to parse GraphQL typedefs.

I install these dependencies using the command:

$ pnpm install apollo-server-express jsonwebtoken argon2 cookie-parser cors dotenv express graphql graphql-tag

Next, I am going to flesh out the src/app.ts file. It needs to initialize the Apollo server. I am going to do this with a function called startApolloServer. This function will be called in the src/app.ts file, but will reside in a new file called src/server.ts.

Now src/app.ts will look like this:

import { startApolloServer } from './server';  
import { config } from "dotenv";  
import { PrismaClient } from '@prisma/client';  
import resolvers from './schema/resolvers';  
import typeDefs from './schema/typeDefs';  
  
// Sets .env config as default.  
config();  
  
// Connects to database.  
export const prisma = new PrismaClient();  
  
// Starts Apollo server.  
startApolloServer(typeDefs, resolvers).then((server) => {  
 // Console message.  
 console.log(  
 `${process.env.NODE\_ENV?.charAt(  
 0  
 ).toUpperCase()}${process.env.NODE\_ENV?.slice(  
 1  
 )} server ready.\nGraphQL explorer at: http://localhost:4000${  
 server.graphqlPath  
 }`  
 );  
});

Inside the callback function, I am logging the URL for Apollo Studio, this will be used for integration testing, so having the URL will be helpful. NODE\_ENV is a variable that will be set to either development or production, depending on the circumstances the server was started under. Production will have strict security rules, while development will have more lax rules that allow for easier testing.

src/server.ts will look like this:

// Stars Apollo Server.  
import { DocumentNode } from "graphql";  
import express from "express";  
import http from "http";  
import cookieParser from "cookie-parser";  
import { ApolloServer } from "apollo-server-express";  
import { ApolloServerPluginDrainHttpServer } from "apollo-server-core";  
import { context } from "./context";  
import { prisma } from "./app";  
  
export async function startApolloServer(  
 typeDefs: DocumentNode,  
 resolvers: any  
) {  
 // Required logic for integrating with Express.  
 const app = express();  
 const httpServer = http.createServer(app);  
  
 // This creates req.cookies  
 app.use(cookieParser());  
  
 app.use(express.json({ limit: "11mb" }));  
  
 app.use((res, req, next) => {  
 // This is required for Apollo Server to work with Express.  
 // @ts-ignore  
 req.context = {};  
 next();  
 });  
  
 // Same ApolloServer initialization as before, plus the drain plugin.  
 const server = new ApolloServer({  
 typeDefs: typeDefs,  
 resolvers: resolvers,  
 plugins: [ApolloServerPluginDrainHttpServer({ httpServer })],  
 context,  
 });  
  
 // More required logic for integrating with Express  
 await server.start();  
 server.applyMiddleware({  
 app,  
 path: "/",  
 cors: {  
 origin:  
 process.env.NODE\_ENV === "development"  
 ? "https://studio.apollographql.com"  
 : "http://localhost:3000",  
 credentials: true,  
 },  
 });  
  
 // Modified server startup  
 await httpServer.listen({ port: 4000 });  
  
 return server;  
}

This starts up the server and applies the necessary middleware and configuration. The cors property is used to allow the server to communicate with the GraphQL explorer, or the client, depending on the value of NODE\_ENV.

The file references a function, context(), that does not yet exist, so I will create that in src/context.ts. Here is the contents of the file.

import jwt, { JwtPayload } from "jsonwebtoken";  
import { cookies } from "./cookies";  
import { prisma } from "./app";  
  
export async function context({ req, res }: any): Promise<Object> {  
 try {  
 // Access Token  
 const { id, iat } = jwt.verify(  
 req.cookies["accessToken"] ?? null,  
 process.env.ACCESS\_TOKEN as string  
 ) as JwtPayload;  
  
 // If the amount of logins is valid.  
 const user = await prisma.user.findFirst({  
 where: { id, iat: { lt: iat } },  
 });  
  
 // If the user is logged in hydrate context.  
 if (user) return { req, res, id };  
 else {  
 // User has been logged out on different device.  
 // Clears the user's cookies.  
 res.clearCookie("accessToken");  
 res.clearCookie("refreshToken");  
 return { req, res, id: null, logouts: null };  
 }  
 } catch (e) {  
 // Refresh Token  
 // If the user's access token is expired.  
 try {  
 const { id, iat } = jwt.verify(  
 req.cookies["refreshToken"] ?? null,  
 process.env.REFRESH\_TOKEN as string  
 ) as JwtPayload;  
  
 // If the amount of logins is valid.  
 const user = await prisma.user.findFirst({  
 where: { id, iat: { lt: iat } },  
 });  
  
 // If the user is logged in hydrate context.  
 if (user) {  
 // User's access token has expired, needs to be reissued.  
 cookies(res, id);  
 return { req, res, id };  
 } else {  
 // User has been logged out on different device.  
 // Clears the user's cookies.  
 res.clearCookie("accessToken");  
 res.clearCookie("refreshToken");  
 return { req, res, id: null };  
 }  
 } catch (e) {  
 res.clearCookie("accessToken");  
 res.clearCookie("refreshToken");  
 return { req, res, id: null };  
 }  
 }  
}

This function is called each time a request is made to the server. It is the basis for the sites’ authentication because it provides the resolvers with the user’s id, which abstracts the cookies away from the resolvers.

The function cookies() also needs to be created. It resides in the file src/cookies.ts. Here it is:

import { Secret, sign } from "jsonwebtoken";  
import { CookieOptions } from "express";  
  
export function cookies(res: any, id: string) {  
 // Clear old cookies  
 res.clearCookie("accessToken");  
 res.clearCookie("refreshToken");  
  
 // Access token expires in an hour, used for authentication.  
 res.cookie(  
 "accessToken",  
 sign({ id: id }, process.env.ACCESS\_TOKEN as Secret),  
 {  
 expires: new Date(Date.now() + 60 \* 60 \* 1000),  
 httpOnly: true,  
 sameSite: process.env.NODE\_ENV === "production" ? "lax" : "none",  
 secure: true,  
 } as CookieOptions  
 );  
 // Refresh token expires in a week, used to generate new access tokens  
 res.cookie(  
 "refreshToken",  
 sign({ id: id }, process.env.REFRESH\_TOKEN as Secret),  
 {  
 expires: new Date(Date.now() + 28 \* 24 \* 60 \* 60 \* 1000),  
 httpOnly: true,  
 sameSite: process.env.NODE\_ENV === "production" ? "lax" : "none",  
 secure: true,  
 } as CookieOptions  
 );  
}

This is a great example of the NODE\_ENV property. The sameSite property is used to determine whether the cookie should be sent to different domains. If the NODE\_ENV is set to production, the cookie will be sent to different domains. If the NODE\_ENV is set to development, the cookie will be sent to the same domain. This allows the cookies to be easily tested, because the GraphQL explorer sits on a different domain than the actual server, so it wil be able to see the cookies, when the sameSite property is set to lax.

Now that the groundwork for the schema to be implemented has been laid out, the next step is to implement the type definitions in the GraphQL schema language. I will do this in the file src/schema/typeDefs.ts.

import gql from "graphql-tag";  
  
export default gql`  
 type User {  
 id: ID!  
 name: String  
 username: String!  
 profilePicture: Image  
 createdAt: String!  
 recipes(take: Int, from: ID, sort: RecipeSort): [Recipe]  
 savedRecipes(take: Int, from: ID, sort: RecipeSort): [Recipe]  
 following(take: Int, from: ID, sort: UserSort): [User]  
 followers(take: Int, from: ID, sort: UserSort): [User]  
 }  
  
 type Image {  
 id: ID!  
 uploader: User!  
 url: String!  
 recipe: Recipe  
 Step: Step  
 }  
  
 type Recipe {  
 id: ID!  
 name: String!  
 author: User!  
 description: String  
 image: Image  
 createdAt: String!  
 steps(take: Int, from: ID, sort: StepSort): [Step]  
 category: String  
 cuisine: String  
 ingredients(take: Int, from: ID, sort: IngredientSort): [Ingredient]  
 cookTime: Int  
 prepTime: Int  
 savedBy(take: Int, from: ID, sort: UserSort): [User]  
 }  
  
 type Step {  
 id: ID!  
 name: String  
 createdAt: String  
 content: String!  
 image: Image  
 recipe: Recipe!  
 }  
  
 type Ingredient {  
 id: ID!  
 name: String!  
 quantity: String!  
 recipe: Recipe  
 }  
  
 type Error {  
 code: Int!  
 message: String!  
 }  
  
 type Query {  
 user(id: ID, username: String): UserResponse!  
 image(id: ID!): ImageResponse!  
 recipe(id: ID!): RecipeResponse!  
 step(id: ID!): StepResponse!  
 ingredient(id: ID!): IngredientResponse!  
  
 users(query: String, take: Int, from: ID, sort: UserSort): [User]  
 recipes(query: String, take: Int, from: ID, sort: RecipeSort): [Recipe]  
 steps(query: String, take: Int, from: ID, sort: StepSort): [Step]  
 ingredients(  
 query: String  
 take: Int  
 from: ID  
 sort: IngredientSort  
 ): [Ingredient]  
 }  
  
 type Mutation {  
 createUser(user: CreateUserInput!, image: ImageInput): UserResponse!  
 loginUser(usernameOrEmail: String!, password: String!): UserResponse!  
 logoutUser: Boolean!  
 editUser(user: EditUserInput, image: ImageInput): UserResponse!  
 deleteUser: Boolean!  
 followUser(id: ID!): UserResponse!  
 unfollowUser(id: ID!): UserResponse!  
  
 createRecipe(recipe: CreateRecipeInput!, image: ImageInput): RecipeResponse!  
 editRecipe(  
 id: ID!  
 recipe: EditRecipeInput  
 image: ImageInput  
 ): RecipeResponse!  
 deleteRecipe(id: ID!): Boolean!  
 saveRecipe(id: ID!): RecipeResponse!  
 unsaveRecipe(id: ID!): RecipeResponse!  
  
 createStep(recipe: ID!, step: StepInput, image: ImageInput): StepResponse!  
 editStep(id: ID!, step: StepInput, image: ImageInput): StepResponse!  
 deleteStep(id: ID!): Boolean!  
  
 createIngredient(  
 recipe: ID!  
 ingredient: IngredientInput  
 ): IngredientResponse!  
 editIngredient(id: ID!, ingredient: IngredientInput): IngredientResponse!  
 deleteIngredient(id: ID!): Boolean!  
 }  
  
 input CreateUserInput {  
 name: String  
 email: String!  
 password: String!  
 username: String!  
 }  
  
 input EditUserInput {  
 name: String  
 username: String  
 }  
  
 input ImageInput {  
 base64: String!  
 }  
  
 input CreateRecipeInput {  
 name: String!  
 description: String  
 category: String  
 cuisine: String  
 cookTime: Int  
 prepTime: Int  
 }  
  
 input EditRecipeInput {  
 name: String  
 description: String  
 category: String  
 cuisine: String  
 cookTime: Int  
 prepTime: Int  
 }  
  
 input StepInput {  
 name: String  
 content: String!  
 }  
  
 input IngredientInput {  
 name: String!  
 quantity: String!  
 }  
  
 input UserSort {  
 \_count: Direction  
 id: Direction  
 name: Direction  
 username: Direction  
 createdAt: Direction  
 recipes: RecipeSort  
 savedRecipes: RecipeSort  
 following: UserSort  
 followers: UserSort  
 }  
  
 input RecipeSort {  
 \_count: Direction  
 id: Direction  
 name: Direction  
 author: UserSort  
 description: Direction  
 createdAt: Direction  
 steps: StepSort  
 category: Direction  
 cuisine: Direction  
 ingredients: IngredientSort  
 cookTime: Direction  
 prepTime: Direction  
 savedBy: UserSort  
 }  
  
 input StepSort {  
 \_count: Direction  
 id: Direction  
 name: Direction  
 createdAt: Direction  
 content: Direction  
 recipe: RecipeSort  
 }  
  
 input IngredientSort {  
 \_count: Direction  
 id: Direction  
 name: Direction  
 quantity: Direction  
 recipe: Direction  
 }  
  
 union UserResponse = User | Error  
 union ImageResponse = Image | Error  
 union RecipeResponse = Recipe | Error  
 union StepResponse = Step | Error  
 union IngredientResponse = Ingredient | Error  
  
 enum Direction {  
 ASC  
 DESC  
 }  
`;

Now if I start the server and open the Apollo Playground, I can see the schema I just created. However, there is currently no script created to start the server. I am going to modify the package.json file to add two scripts to start the server in development mode and production mode. For development purposes, two small scripts are required, cross-env and nodemon. The cross-env script will set the NODE\_ENV environment variable and the nodemon script will restart the server whenever the source code changes. This is useful in development because it allows me to rapidly prototype the server without having to manually type a command each time I save a change to the source code.

I can install these scripts by running the following command in the terminal:

$ pnpm install -D cross-env nodemon

Now I can edit the package.json file and change the scripts section to this:

{  
 "scripts": {  
 "start": "cross-env NODE\_ENV=production ts-node src/app.ts",  
 "dev": "cross-env NODE\_ENV=development nodemon --exec ts-node src/app.ts",  
 "build": "tsc --build",  
 "test": "mocha -r ts-node/register test/test.ts"  
 }  
}

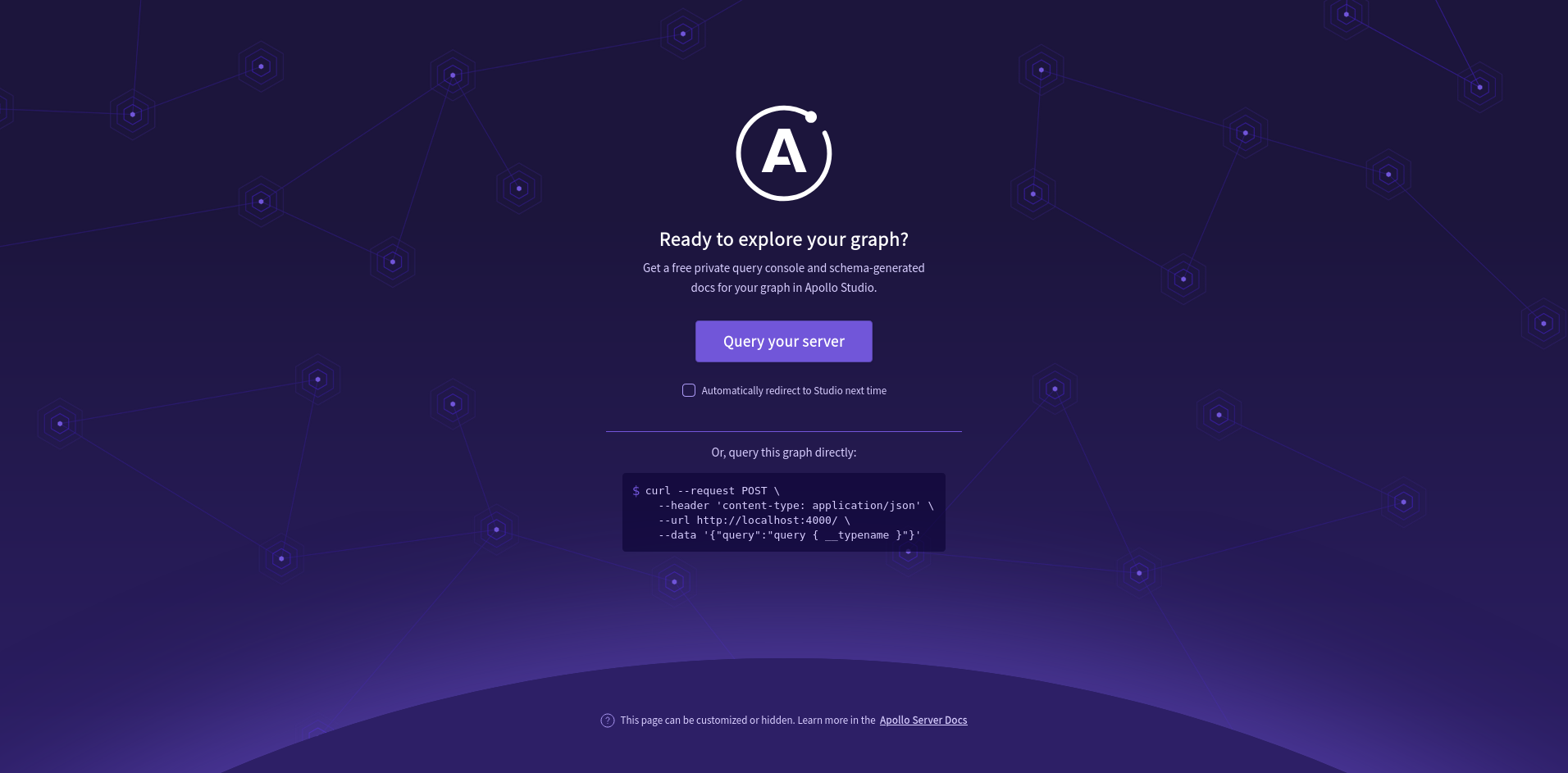
Now I can start the server by running the following command in the terminal:

$ npm run dev

This will output:

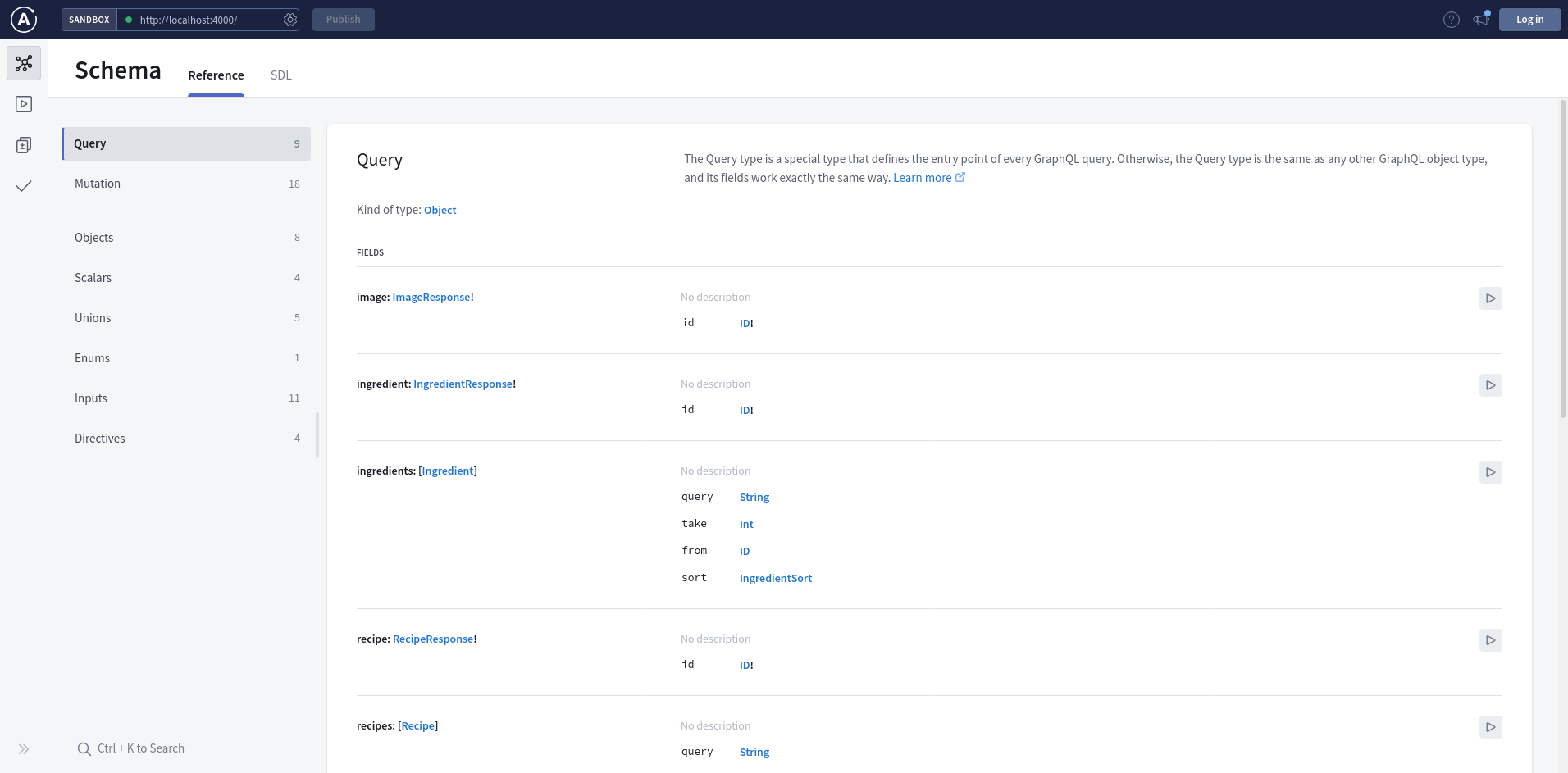
[nodemon] 2.0.13  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): \*.\*  
[nodemon] watching extensions: ts,json  
[nodemon] starting `ts-node src/app.ts`  
Development server ready.  
GraphQL explorer at: http://localhost:4000/

By clicking the link, I get bought to this page:



A landing page for Apollo Explorer

The button leads to the GraphQL explorer. I can now see the schema I just created.



A GraphQL schema

The type definitions have now been successfully created! However, I still need to create the resolvers. The typedefs on their own don’t do much, but the resolvers are the heart of the Apollo server.

I created a new file called src/schema/resolvers.ts and added the following code:

import query from "./query";  
import mutation from "./mutations";  
import user from "./user";  
import recipe from "./recipe";  
  
export default {  
 Query: {  
 ...query,  
 },  
 Mutation: {  
 ...mutation,  
 },  
 User: {  
 ...user,  
 },  
 Recipe: {  
 ...recipe,  
 },  
 UserResponse: {  
 \_\_resolveType(user: { code: number }) {  
 if (user.code) {  
 return "Error";  
 } else {  
 return "User";  
 }  
 },  
 },  
 ImageResponse: {  
 \_\_resolveType(image: { code: number }) {  
 if (image.code) {  
 return "Error";  
 } else {  
 return "Image";  
 }  
 },  
 },  
 RecipeResponse: {  
 \_\_resolveType(recipe: { code: number }) {  
 if (recipe.code) {  
 return "Error";  
 } else {  
 return "Recipe";  
 }  
 },  
 },  
 IngredientResponse: {  
 \_\_resolveType(recipe: { code: number }) {  
 if (recipe.code) {  
 return "Error";  
 } else {  
 return "Ingredient";  
 }  
 },  
 },  
 StepResponse: {  
 \_\_resolveType(recipe: { code: number }) {  
 if (recipe.code) {  
 return "Error";  
 } else {  
 return "Step";  
 }  
 },  
 },  
 Direction: {  
 ASC: "asc",  
 DESC: "desc",  
 },  
};

This file imports the resolvers from the other files and exports them as one object. This is so src/app.ts can pass them to the startApolloServer() function.

There are also some basic resolvers in the file I have already created. They are very short and simple, so it’s not worth extracting them into their own file.

The next file I am going to create is src/schema/query/index.ts. This file will export all the queries in the src/schema/query directory.

import singular from "./singular";  
import plural from "./plural";  
  
export default {  
 ...singular,  
 ...plural,  
};

This file calls the singular and plural files and exports them as one object. Here is the code for the singular:

import { prisma } from "../../app";  
  
export default {  
 user: async (  
 \_ = null,  
 { id, username }: { id: string; username: string },  
 { id: uuid }: { id: string }  
 ) => {  
 if (id) {  
 return (  
 (await prisma.user.findUnique({  
 where: { id: id },  
 })) ?? {  
 // Id is not found  
 code: 404,  
 message: "User not found",  
 }  
 );  
 } else if (username) {  
 return (  
 (await prisma.user.findUnique({  
 where: { username: username },  
 })) ?? {  
 // Username is not found  
 code: 404,  
 message: "User not found",  
 }  
 );  
 } else if (uuid) {  
 return (  
 (await prisma.user.findUnique({  
 where: { id: uuid },  
 })) ?? {  
 // Something is very broken  
 code: 500,  
 message: "Something is very broken",  
 }  
 );  
 } else {  
 return {  
 // Not logged in and no user provided.  
 code: 401,  
 message: "Not logged in and no user provided.",  
 };  
 }  
 },  
 recipe: async (\_ = null, { id }: { id: string }) => {  
 return (  
 (await prisma.recipe.findUnique({  
 where: { id: id },  
 })) ?? {  
 // Id is not found  
 code: 404,  
 message: "Recipe not found",  
 }  
 );  
 },  
 image: async (\_ = null, { id }: { id: string }) => {  
 return (  
 (await prisma.image.findUnique({ where: { id: id } })) ?? {  
 // Id is not found  
 code: 404,  
 message: "Image not found",  
 }  
 );  
 },  
 step: async (\_ = null, { id }: { id: string }) => {  
 return (  
 (await prisma.step.findUnique({ where: { id: id } })) ?? {  
 // Id is not found  
 code: 404,  
 message: "Step not found",  
 }  
 );  
 },  
 ingredient: async (\_ = null, { id }: { id: string }) => {  
 return (  
 (await prisma.ingredient.findUnique({ where: { id: id } })) ?? {  
 // Id is not found  
 code: 404,  
 message: "Ingredient not found",  
 }  
 );  
 },  
};

And here is the code for plural:

import { prisma } from "../../app";  
  
export default {  
 users: async (  
 \_ = null,  
 {  
 query,  
 take,  
 from,  
 sort,  
 }: { query: string; take: number; from: string; sort: any },  
 { req, res, id }: { req: any; res: any; id: string }  
 ) => {  
 return await prisma.user.findMany({  
 where: {  
 OR: [  
 {  
 name: {  
 search: query,  
 },  
 },  
 {  
 username: {  
 search: query,  
 },  
 },  
 ],  
 },  
 take,  
 cursor: from ? { id: from } : undefined,  
 orderBy: sort,  
 });  
 },  
 recipes: async (  
 \_ = null,  
 {  
 query,  
 take,  
 from,  
 sort,  
 }: { query: string; take: number; from: string; sort: any },  
 { req, res, id }: { req: any; res: any; id: string }  
 ) => {  
 return await prisma.recipe.findMany({  
 where: {  
 OR: [  
 {  
 name: {  
 search: query,  
 },  
 },  
 {  
 description: {  
 search: query,  
 },  
 },  
 {  
 cuisine: {  
 search: query,  
 },  
 },  
 {  
 category: {  
 search: query,  
 },  
 },  
 ],  
 },  
 take,  
 cursor: from ? { id: from } : undefined,  
 orderBy: sort,  
 });  
 },  
 steps: async (  
 \_ = null,  
 {  
 query,  
 take,  
 from,  
 sort,  
 }: { query: string; take: number; from: string; sort: any },  
 { req, res, id }: { req: any; res: any; id: string }  
 ) => {  
 return await prisma.step.findMany({  
 where: {  
 OR: [  
 {  
 name: {  
 search: query,  
 },  
 },  
 {  
 content: {  
 search: query,  
 },  
 },  
 ],  
 },  
 take,  
 cursor: from ? { id: from } : undefined,  
 orderBy: sort,  
 });  
 },  
 ingredients: async (  
 \_ = null,  
 {  
 query,  
 take,  
 from,  
 sort,  
 }: { query: string; take: number; from: string; sort: any },  
 { req, res, id }: { req: any; res: any; id: string }  
 ) => {  
 return await prisma.ingredient.findMany({  
 where: {  
 name: {  
 search: query,  
 },  
 },  
 take,  
 cursor: from ? { id: from } : undefined,  
 orderBy: sort,  
 });  
 },  
};

These are simple queries and don’t require explanation outside their own comments.

Next, I am going to add the src/schema/mutations/index.ts file. This is also a file that just imports and exports objects for use in the src/app.ts file. Here it is:

import auth from "./auth";  
import user from "./user";  
import recipe from "./recipe";  
import ingredient from "./ingredient";  
import step from "./step";  
  
export default {  
 ...auth,  
 ...user,  
 ...recipe,  
 ...ingredient,  
 ...step,  
};

This file also just imports and export other files that are in the same directory. Here is the code for auth:

import { prisma } from "../../app";  
import argon2 from "argon2";  
import { cookies } from "../../cookies";  
  
export default {  
 loginUser: async (  
 \_ = null,  
 {  
 usernameOrEmail,  
 password,  
 }: { usernameOrEmail: string; password: string },  
 { \_\_, res }: any  
 ) => {  
 //Check if usernameOrEmail is an email or username  
 const where = usernameOrEmail.includes("@")  
 ? { email: usernameOrEmail }  
 : { username: usernameOrEmail };  
  
 const user = await prisma.user.findUnique({ where });  
  
 if (!user) {  
 return {  
 code: 400,  
 message: "Email or username is incorrect",  
 };  
 }  
  
 if (!user.emailVerified) {  
 return {  
 code: 400,  
 message: "Email is not verified",  
 };  
 }  
  
 if (await argon2.verify(user.password, password)) {  
 cookies(res, user.id);  
  
 return user;  
 } else {  
 return {  
 code: 400,  
 message: "Password is incorrect",  
 };  
 }  
 },  
 logoutUser: async (\_ = null, \_\_: any, { res, id }: any) => {  
 //seconds since epoch  
 const iat = Math.floor(Date.now() / 1000);  
  
 await prisma.user.update({  
 where: { id },  
 data: {  
 iat,  
 },  
 });  
  
 res.clearCookie("accessToken");  
 res.clearCookie("refreshToken");  
  
 return true;  
 },  
};

Here is the code for user:

import { prisma, s3 } from "../../app";  
import argon2 from "argon2";  
  
export default {  
 createUser: async (  
 \_ = null,  
 {  
 user,  
 }: {  
 user: {  
 name?: string;  
 username: string;  
 email: string;  
 password: string;  
 };  
 }  
 ) => {  
 //Check username is valid  
 if (user.username.length < 3) {  
 return {  
 code: 400,  
 message: "Username must be at least 3 characters long",  
 };  
 }  
  
 if (user.username.length > 20) {  
 return {  
 code: 400,  
 message: "Username must be less than 20 characters long",  
 };  
 }  
  
 if (user.username.match(/^[a-z-]+$/) === null) {  
 return {  
 code: 400,  
 message: "Username must only contain lowercase letters and dashes",  
 };  
 }  
  
 if (user.username.match(/^[a-z][a-z-]+[a-z]$/) === null) {  
 return {  
 code: 400,  
 message: "Username must start and end with a lowercase letter",  
 };  
 }  
  
 //Check email is valid  
 if (user.email.indexOf("@") === -1) {  
 return {  
 code: 400,  
 message: "Email is invalid",  
 };  
 }  
  
 if (user.email.length > 100) {  
 return {  
 code: 400,  
 message: "Email must be less than 100 characters long",  
 };  
 }  
  
 //Check password is valid  
 if (user.password.length < 8) {  
 return {  
 code: 400,  
 message: "Password must be at least 8 characters long",  
 };  
 }  
  
 if (user.password.length > 100) {  
 return {  
 code: 400,  
 message: "Password must be less than 100 characters long",  
 };  
 }  
  
 // Check all required fields are present  
 if (!user.username) {  
 return {  
 code: 400,  
 message: "Username is required",  
 };  
 }  
  
 if (!user.email) {  
 return {  
 code: 400,  
 message: "Email is required",  
 };  
 }  
  
 if (!user.password) {  
 return {  
 code: 400,  
 message: "Password is required",  
 };  
 }  
  
 // Check if user already exists  
 const userExists = await prisma.user.findFirst({  
 where: {  
 OR: [  
 {  
 username: user.username,  
 },  
 {  
 email: user.email,  
 },  
 ],  
 },  
 });  
  
 if (userExists) {  
 if (  
 userExists.username === user.username &&  
 userExists.email === user.email  
 ) {  
 return {  
 code: 400,  
 message: "Username and email already exists",  
 };  
 }  
 if (userExists.email === user.email) {  
 return {  
 code: 400,  
 message: "Email already exists",  
 };  
 }  
 if (userExists.username === user.username) {  
 return {  
 code: 400,  
 message: "Username already exists",  
 };  
 }  
 }  
  
 user.password = await argon2.hash(user.password);  
  
 return await prisma.user.create({  
 data: {  
 ...user,  
 // TODO add email verification  
 emailVerified: true,  
 },  
 });  
 },  
 editUser: async (  
 \_ = null,  
 {  
 user: { name, username },  
 }: {  
 user: {  
 name: string;  
 username: string;  
 };  
 },  
 { id }: { id: string }  
 ) => {  
 if (!id) {  
 return {  
 // Not logged in  
 code: 401,  
 message: "Not logged in",  
 };  
 }  
  
 if (username) {  
 //Check username is valid  
 if (username.length < 3) {  
 return {  
 code: 400,  
 message: "Username must be at least 3 characters long",  
 };  
 }  
  
 if (username.length > 20) {  
 return {  
 code: 400,  
 message: "Username must be less than 20 characters long",  
 };  
 }  
  
 if (username.match(/^[a-z-]+$/) === null) {  
 return {  
 code: 400,  
 message: "Username must only contain lowercase letters and dashes",  
 };  
 }  
  
 if (username.match(/^[a-z][a-z-]+[a-z]$/) === null) {  
 return {  
 code: 400,  
 message: "Username must start and end with a lowercase letter",  
 };  
 }  
  
 // Check if username is taken  
 if (  
 await prisma.user.findFirst({  
 where: {  
 username,  
 },  
 })  
 ) {  
 return {  
 code: 400,  
 message: "Username is taken",  
 };  
 }  
 }  
  
 return prisma.user.update({  
 where: {  
 id,  
 },  
 data: {  
 name: name,  
 username: username,  
 },  
 });  
 },  
 deleteUser: async (\_ = null, \_\_ = null, { id }: { id: string }) => {  
 if (!id) {  
 return {  
 code: 401,  
 message: "Not logged in",  
 };  
 }  
  
 // Delete user's images  
 const images = await prisma.image.deleteMany({  
 where: {  
 OR: [  
 {  
 profile: {  
 id,  
 },  
 },  
 {  
 recipe: {  
 author: {  
 id,  
 },  
 },  
 },  
 {  
 step: {  
 recipe: {  
 author: {  
 id,  
 },  
 },  
 },  
 },  
 ],  
 },  
 });  
  
 // Delete user's ingredients  
 const ingredients = await prisma.ingredient.deleteMany({  
 where: {  
 recipe: {  
 author: {  
 id,  
 },  
 },  
 },  
 });  
  
 // Delete user's steps  
 const steps = await prisma.step.deleteMany({  
 where: {  
 recipe: {  
 author: {  
 id,  
 },  
 },  
 },  
 });  
  
 // Delete user's recipes  
 const recipes = await prisma.recipe.deleteMany({  
 where: {  
 author: {  
 id,  
 },  
 },  
 });  
  
 await prisma.user.delete({ where: { id } });  
  
 return true;  
 },  
 followUser: async (  
 \_ = null,  
 args: { id: string },  
 { id }: { id: string }  
 ) => {  
 if (!id) {  
 return {  
 code: 401,  
 message: "Not logged in",  
 };  
 }  
 if (args.id === id) {  
 return {  
 code: 400,  
 message: "You cannot follow yourself",  
 };  
 }  
 return await prisma.user.update({  
 where: {  
 id,  
 },  
 data: {  
 following: {  
 connect: {  
 id: args.id,  
 },  
 },  
 },  
 });  
 },  
 unfollowUser: async (  
 \_ = null,  
 args: { id: string },  
 { id }: { id: string }  
 ) => {  
 if (!id) {  
 return {  
 code: 401,  
 message: "Not logged in",  
 };  
 }  
 if (args.id === id) {  
 return {  
 code: 400,  
 message: "You cannot unfollow yourself",  
 };  
 }  
 return await prisma.user.update({  
 where: {  
 id,  
 },  
 data: {  
 following: {  
 disconnect: {  
 id: args.id,  
 },  
 },  
 },  
 });  
 },  
};

Here are the functions in recipe:

import { prisma } from "../../app";  
  
export default {  
 createRecipe: async (  
 \_ = null,  
 {  
 recipe,  
 }: {  
 recipe: {  
 name: string;  
 description?: string;  
 category?: string;  
 cuisine?: string;  
 cookTime?: number;  
 prepTime?: number;  
 };  
 },  
 args: { id: string }  
 ) => {  
 // Check if user is authenticated  
 if (!args.id) {  
 return {  
 code: 401,  
 message: "You must be logged in to perform this action",  
 };  
 }  
  
 // Create recipe  
 return await prisma.recipe.create({  
 data: {  
 ...recipe,  
 author: {  
 connect: {  
 id: args.id,  
 },  
 },  
 },  
 });  
 },  
 editRecipe: async (  
 \_ = null,  
 {  
 id,  
 recipe,  
 }: {  
 id: string;  
 recipe: {  
 name: string;  
 description?: string;  
 category?: string;  
 cuisine?: string;  
 cookTime?: number;  
 prepTime?: number;  
 };  
 },  
 args: { id: string }  
 ) => {  
 // Check if user is authenticated  
 if (!args.id) {  
 return {  
 code: 401,  
 message: "You must be logged in to perform this action",  
 };  
 }  
  
 // Check if recipe exists  
 const recipeExists = await prisma.recipe.findUnique({  
 where: {  
 id: id,  
 },  
 });  
  
 if (!recipeExists) {  
 return {  
 code: 404,  
 message: "Recipe not found",  
 };  
 }  
  
 if (recipeExists?.authorId !== args?.id) {  
 return {  
 code: 403,  
 message: "You are not authorized to edit this recipe",  
 };  
 }  
  
 if (recipeExists?.authorId !== args?.id) {  
 return {};  
 }  
  
 // Update recipe  
 return await prisma.recipe.update({  
 where: {  
 id,  
 },  
 data: {  
 ...recipe,  
 },  
 });  
 },  
 deleteRecipe: async (  
 \_ = null,  
 { id }: { id: string },  
 args: { id: string }  
 ) => {  
 // Check if user is authenticated  
 if (!args.id) {  
 return {  
 code: 401,  
 message: "You must be logged in to perform this action",  
 };  
 }  
  
 // Check if recipe exists  
 const recipeExists = await prisma.recipe.findUnique({  
 where: {  
 id: id,  
 },  
 select: {  
 authorId: true,  
 },  
 });  
  
 if (!recipeExists) {  
 return {  
 code: 404,  
 message: "Recipe not found",  
 };  
 }  
  
 if (recipeExists?.authorId !== args?.id) {  
 return {  
 code: 403,  
 message: "You are not authorized to edit this recipe",  
 };  
 }  
  
 // Update recipe  
 return await prisma.recipe.delete({  
 where: {  
 id,  
 },  
 });  
 },  
 saveRecipe: async (  
 \_ = null,  
 { id }: { id: string },  
 args: { id: string }  
 ) => {  
 // Check if user is authenticated  
 if (!args.id) {  
 return {  
 code: 401,  
 message: "You must be logged in to perform this action",  
 };  
 }  
  
 // Check if recipe exists  
 const recipeExists = await prisma.recipe.findUnique({  
 where: {  
 id: id,  
 },  
 });  
  
 if (!recipeExists) {  
 return {  
 code: 404,  
 message: "Recipe not found",  
 };  
 }  
  
 await prisma.user.update({  
 where: {  
 id: args.id,  
 },  
 data: {  
 saved: {  
 connect: {  
 id: id,  
 },  
 },  
 },  
 });  
 return recipeExists;  
 },  
 unsaveRecipe: async (  
 \_ = null,  
 { id }: { id: string },  
 args: { id: string }  
 ) => {  
 // Check if user is authenticated  
 if (!args.id) {  
 return {  
 code: 401,  
 message: "You must be logged in to perform this action",  
 };  
 }  
  
 // Check if recipe exists  
 const recipeExists = await prisma.recipe.findUnique({  
 where: {  
 id: id,  
 },  
 });  
  
 if (!recipeExists) {  
 return {  
 code: 404,  
 message: "Recipe not found",  
 };  
 }  
  
 await prisma.user.update({  
 where: {  
 id: args.id,  
 },  
 data: {  
 saved: {  
 disconnect: {  
 id: id,  
 },  
 },  
 },  
 });  
 return recipeExists;  
 },  
};

Here is the code in step:

import { prisma } from "../../app";  
  
export default {  
 createStep: async (  
 \_ = null,  
 {  
 recipe,  
 step,  
 }: { recipe: string; step: { name?: string; content: string } },  
 args: { id: string }  
 ) => {  
 // Check if user is authenticated  
 if (!args.id) {  
 return {  
 code: 401,  
 message: "You must be logged in to perform this action",  
 };  
 }  
  
 // Check if recipe exists  
 const recipeExists = await prisma.recipe.findMany({  
 where: {  
 id: recipe,  
 author: {  
 id: args.id,  
 },  
 },  
 });  
  
 if (recipeExists.length === 0) {  
 return {  
 code: 404,  
 message: "Recipe not found",  
 };  
 }  
  
 // Create step  
 return await prisma.step.create({  
 data: {  
 ...step,  
 recipe: {  
 connect: {  
 id: recipe,  
 },  
 },  
 },  
 });  
 },  
 editStep: async (  
 \_ = null,  
 { id, step }: { id: string; step: { name?: string; content: string } },  
 args: { id: string }  
 ) => {  
 // Check if user is authenticated  
 if (!args.id) {  
 return {  
 code: 401,  
 message: "You must be logged in to perform this action",  
 };  
 }  
  
 // Check if step exists  
 const recipeExists = await prisma.step.findUnique({  
 where: {  
 id: id,  
 },  
 select: {  
 recipe: {  
 select: {  
 authorId: true,  
 },  
 },  
 },  
 });  
  
 if (!recipeExists) {  
 return {  
 code: 404,  
 message: "Step not found",  
 };  
 }  
  
 if (recipeExists?.recipe?.authorId !== args?.id) {  
 return {  
 code: 403,  
 message: "You are not authorized to edit this step",  
 };  
 }  
  
 // Update step  
 return await prisma.step.update({  
 where: {  
 id,  
 },  
 data: {  
 ...step,  
 },  
 });  
 },  
 deleteStep: async (  
 \_ = null,  
 { id }: { id: string },  
 args: { id: string }  
 ) => {  
 // Check if user is authenticated  
 if (!args.id) {  
 return {  
 code: 401,  
 message: "You must be logged in to perform this action",  
 };  
 }  
  
 // Check if step exists  
 const recipeExists = await prisma.step.findUnique({  
 where: {  
 id: id,  
 },  
 select: {  
 recipe: {  
 select: {  
 authorId: true,  
 },  
 },  
 },  
 });  
  
 if (!recipeExists) {  
 return {  
 code: 404,  
 message: "Step not found",  
 };  
 }  
  
 if (recipeExists?.recipe?.authorId !== args?.id) {  
 return {  
 code: 403,  
 message: "You are not authorized to edit this step",  
 };  
 }  
  
 // Update step  
 return await prisma.step.delete({  
 where: {  
 id,  
 },  
 });  
 },  
};

Here is ingredient:

import { prisma } from "../../app";  
  
export default {  
 createIngredient: async (  
 \_ = null,  
 {  
 recipe,  
 ingredient,  
 }: { recipe: string; ingredient: { name: string; quantity: string } },  
 args: { id: string }  
 ) => {  
 // Check if user is authenticated  
 if (!args.id) {  
 return {  
 code: 401,  
 message: "You must be logged in to perform this action",  
 };  
 }  
  
 // Check if recipe exists  
 const recipeExists = await prisma.recipe.findMany({  
 where: {  
 id: recipe,  
 author: {  
 id: args.id,  
 },  
 },  
 });  
  
 if (recipeExists.length === 0) {  
 return {  
 code: 404,  
 message: "Recipe not found",  
 };  
 }  
  
 // Create ingredient  
 return await prisma.ingredient.create({  
 data: {  
 ...ingredient,  
 recipe: {  
 connect: {  
 id: recipe,  
 },  
 },  
 },  
 });  
 },  
 editIngredient: async (  
 \_ = null,  
 {  
 id,  
 ingredient,  
 }: { id: string; ingredient: { name: string; quantity: string } },  
 args: { id: string }  
 ) => {  
 // Check if user is authenticated  
 if (!args.id) {  
 return {  
 code: 401,  
 message: "You must be logged in to perform this action",  
 };  
 }  
  
 // Check if ingredient exists  
 const recipeExists = await prisma.ingredient.findUnique({  
 where: {  
 id: id,  
 },  
 select: {  
 recipe: {  
 select: {  
 authorId: true,  
 },  
 },  
 },  
 });  
  
 if (!recipeExists) {  
 return {  
 code: 404,  
 message: "Ingredient not found",  
 };  
 }  
  
 if (recipeExists?.recipe?.authorId !== args?.id) {  
 return {  
 code: 403,  
 message: "You are not authorized to edit this ingredient",  
 };  
 }  
  
 // Update ingredient  
 return await prisma.ingredient.update({  
 where: {  
 id,  
 },  
 data: {  
 ...ingredient,  
 },  
 });  
 },  
 deleteIngredient: async (  
 \_ = null,  
 { id }: { id: string },  
 args: { id: string }  
 ) => {  
 // Check if user is authenticated  
 if (!args.id) {  
 return {  
 code: 401,  
 message: "You must be logged in to perform this action",  
 };  
 }  
  
 // Check if ingredient exists  
 const recipeExists = await prisma.ingredient.findUnique({  
 where: {  
 id: id,  
 },  
 select: {  
 recipe: {  
 select: {  
 authorId: true,  
 },  
 },  
 },  
 });  
  
 if (!recipeExists) {  
 return {  
 code: 404,  
 message: "Ingredient not found",  
 };  
 }  
  
 if (recipeExists?.recipe?.authorId !== args?.id) {  
 return {  
 code: 403,  
 message: "You are not authorized to edit this ingredient",  
 };  
 }  
  
 // Update ingredient  
 return await prisma.ingredient.delete({  
 where: {  
 id,  
 },  
 });  
 },  
};

There are also resolvers for individual fields, they sit in the src/schema/ folder. There are two files recipe. ts and user.ts. Here is the recipe.ts:

import { prisma } from "../app";  
  
export default {  
 steps: async (  
 parent: any,  
 { take, from, sort }: { take: number; from: string; sort: any }  
 ) => {  
 return await prisma.step.findMany({  
 where: {  
 recipe: {  
 id: parent.id,  
 },  
 },  
 take,  
 cursor: from ? { id: from } : undefined,  
 orderBy: sort,  
 });  
 },  
 ingredients: async (  
 parent: any,  
 { take, from, sort }: { take: number; from: string; sort: any }  
 ) => {  
 return await prisma.ingredient.findMany({  
 where: {  
 recipe: {  
 id: parent.id,  
 },  
 },  
 take,  
 cursor: from ? { id: from } : undefined,  
 orderBy: sort,  
 });  
 },  
 savedBy: async (  
 parent: any,  
 { take, from, sort }: { take: number; from: string; sort: any }  
 ) => {  
 return await prisma.user.findMany({  
 where: {  
 saved: {  
 some: {  
 id: parent.id,  
 },  
 },  
 },  
 take,  
 cursor: from ? { id: from } : undefined,  
 orderBy: sort,  
 });  
 },  
};

And here is the user.ts:

import { prisma } from "../app";  
  
export default {  
 recipes: async (  
 parent: any,  
 { take, from, sort }: { take: number; from: string; sort: any }  
 ) => {  
 return await prisma.recipe.findMany({  
 where: {  
 authorId: parent.id,  
 },  
 take,  
 cursor: from ? { id: from } : undefined,  
 orderBy: sort,  
 });  
 },  
  
 savedRecipes: async (  
 parent: any,  
 { take, from, sort }: { take: number; from: string; sort: any }  
 ) => {  
 return await prisma.recipe.findMany({  
 where: {  
 savedBy: {  
 some: {  
 id: parent.id,  
 },  
 },  
 },  
 take,  
 cursor: from ? { id: from } : undefined,  
 orderBy: sort,  
 });  
 },  
  
 followers: async (  
 parent: any,  
 { take, from, sort }: { take: number; from: string; sort: any }  
 ) => {  
 return await prisma.user.findMany({  
 where: {  
 following: {  
 some: {  
 id: parent.id,  
 },  
 },  
 },  
 take,  
 cursor: from ? { id: from } : undefined,  
 orderBy: sort,  
 });  
 },  
  
 following: async (  
 parent: any,  
 { take, from, sort }: { take: number; from: string; sort: any }  
 ) => {  
 return await prisma.user.findMany({  
 where: {  
 followers: {  
 some: {  
 id: parent.id,  
 },  
 },  
 },  
 take,  
 cursor: from ? { id: from } : undefined,  
 orderBy: sort,  
 });  
 },  
};