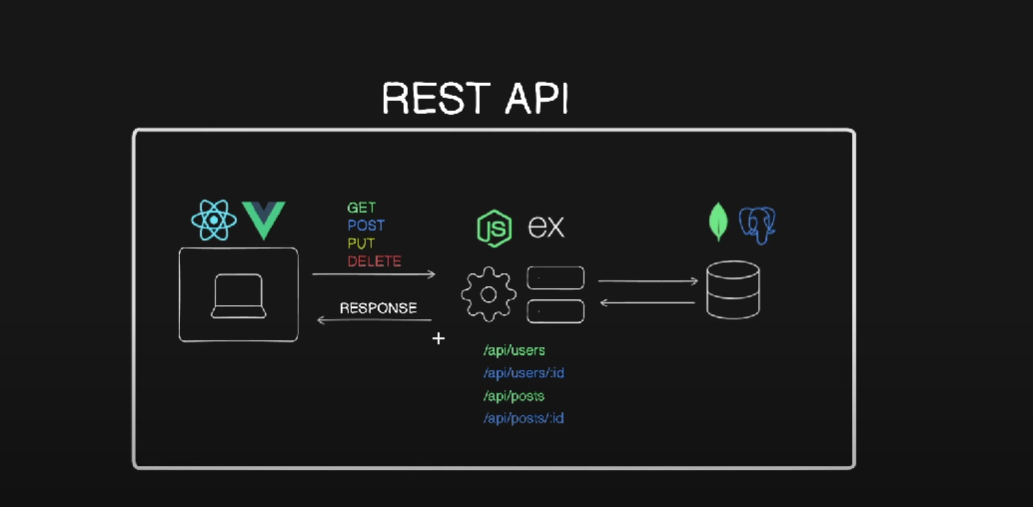
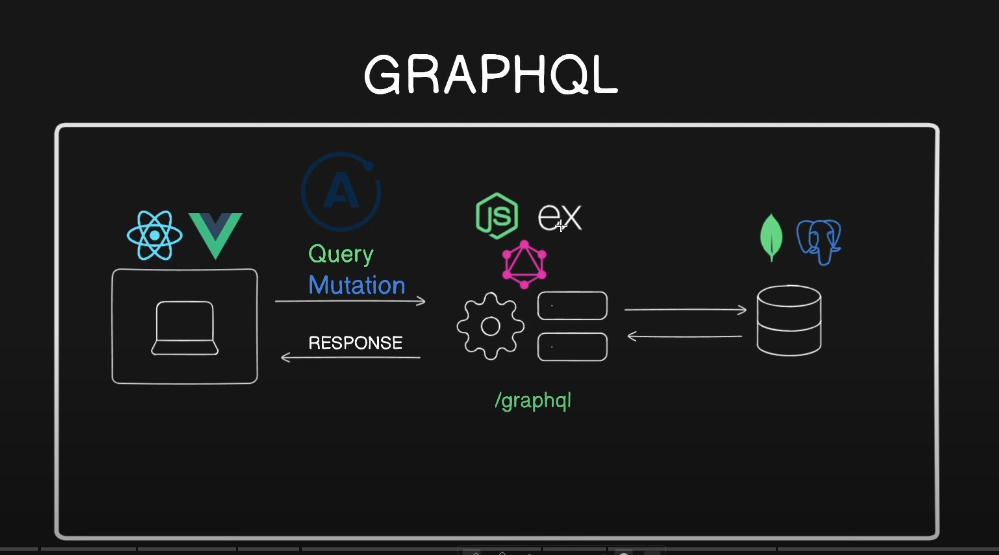


Rest API architecture:



GraphQL Architecture looks like this in comparison to REST API. All info available in single endpoint.



Some examples related to GraphQL Query:

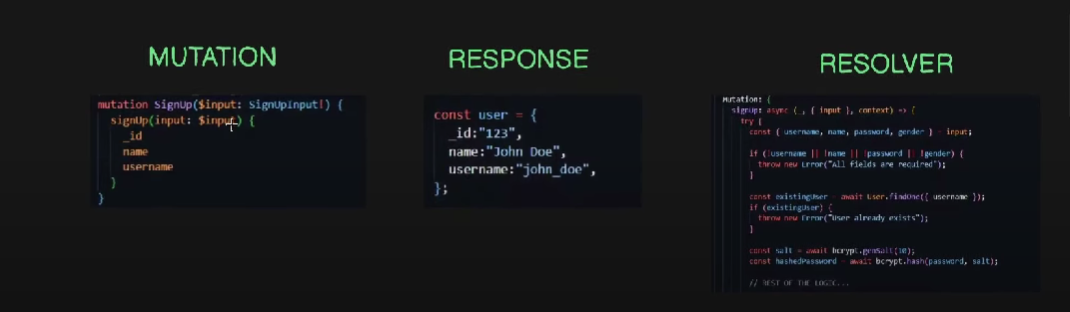
Normal-



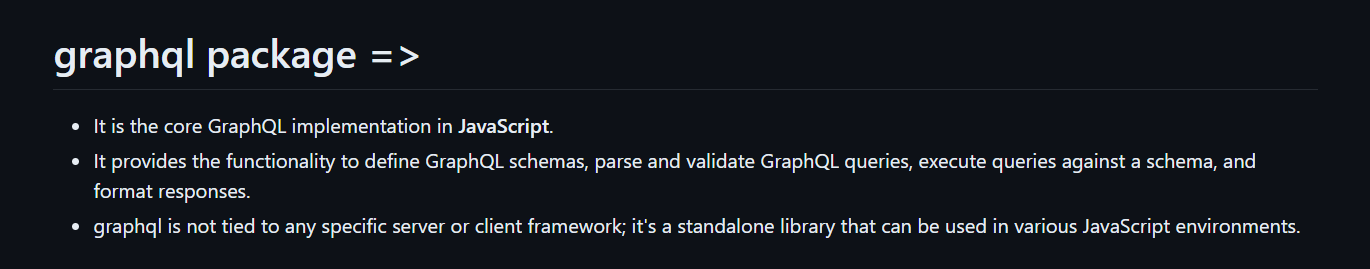
Parameterized-

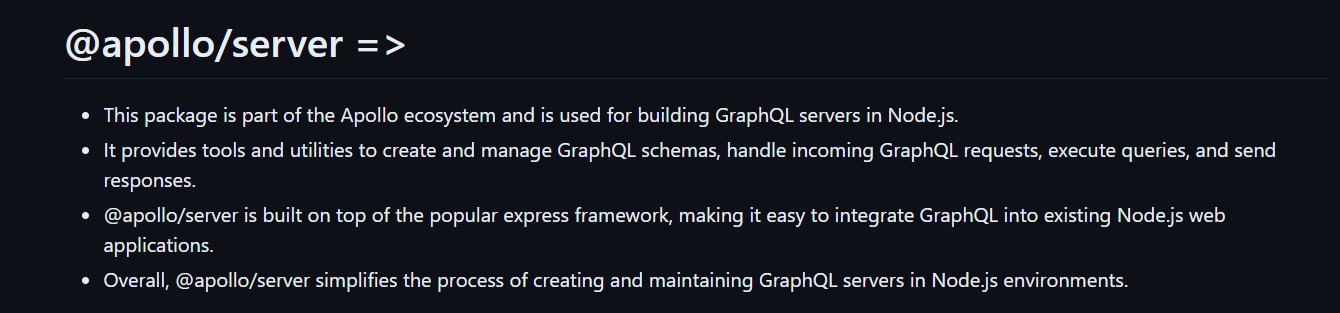


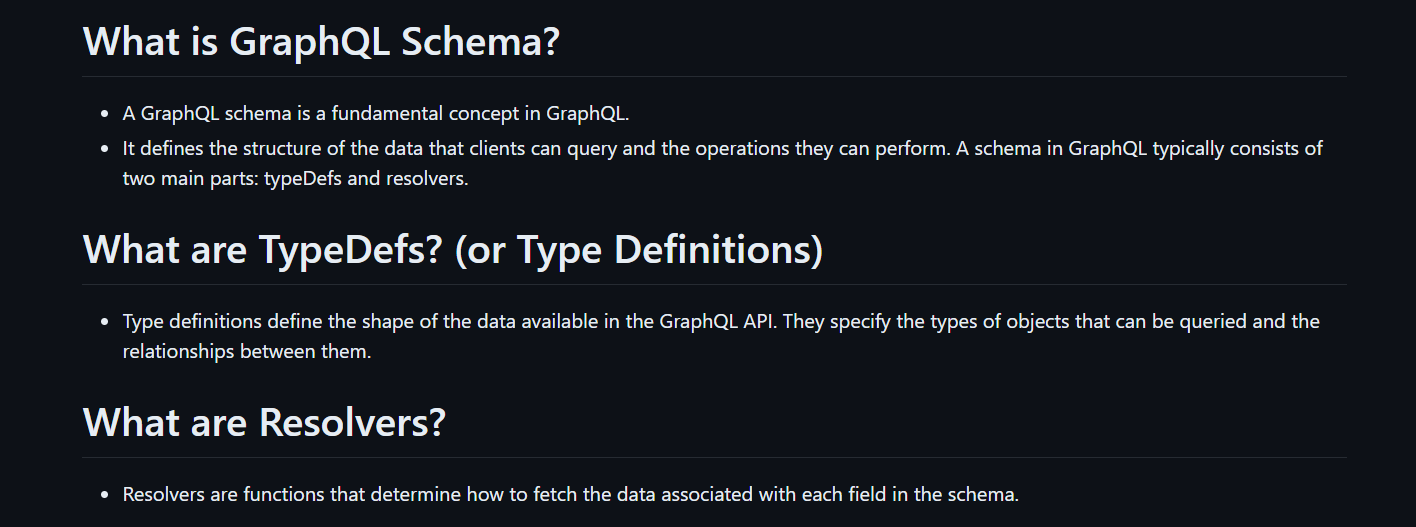
Some examples related to GraphQL resolvers:



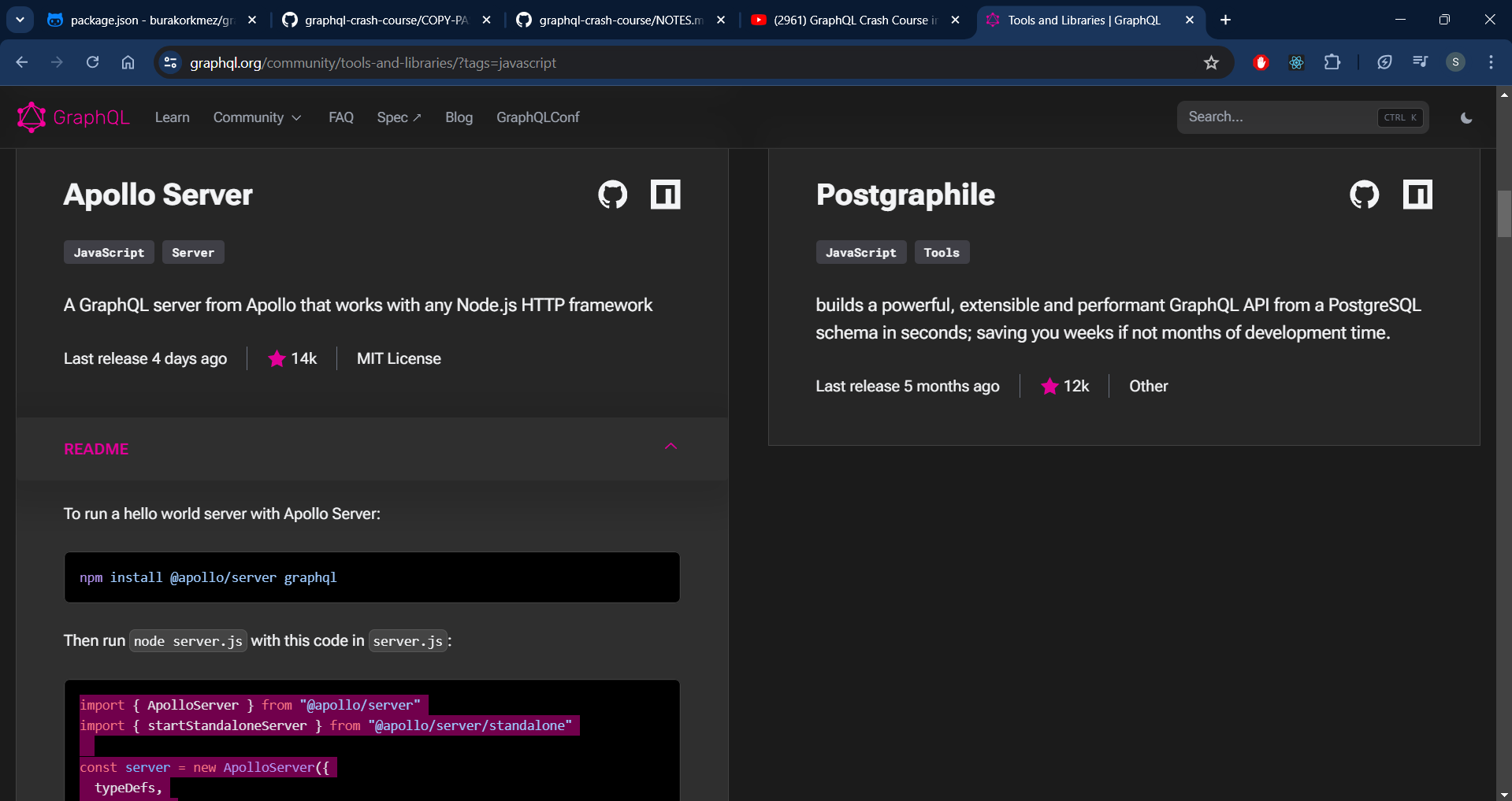
GraphQL Definations:







How to get code snippets:



Query Type:

In GraphQL, a Query type is one of the fundamental operation types used to request data from a server. It allows clients to specify what data they need and in what structure, enabling efficient data retrieval.

 **Root Type for Fetching Data**: The Query type is the entry point for read operations in a GraphQL schema. It defines what fields can be queried by the client.

 **Defining Queries**: In the schema definition, the Query type is typically defined as an object type with various fields. Each field represents a possible query operation that clients can execute.

 **Fields and Resolvers**: Fields in the Query type map to resolver functions on the server. These resolvers contain the logic for fetching the requested data, whether from a database, an external API, or another source.

 **Arguments**: Query fields can accept arguments, allowing clients to specify parameters that control what data is retrieved. This can include filtering, pagination, sorting, etc.

 **Return Types**: Each field in the Query type has a specified return type, which can be a scalar type (like String, Int, etc.), an object type, a list, or even another query.

Mutation Type:

In GraphQL, a Mutation type is used to define operations that modify data on the server. This includes creating, updating, or deleting records. While Query types are used for fetching data, Mutation types are used for changing it.

**Key Concepts of the Mutation Type:**

1. **Root Type for Data Modification**: The Mutation type is the entry point for write operations in a GraphQL schema. It defines what operations can be performed to modify the data.
2. **Defining Mutations**: Similar to the Query type, the Mutation type is an object type with fields representing different mutation operations.
3. **Fields and Resolvers**: Fields in the Mutation type map to resolver functions on the server. These resolvers contain the logic for performing the data modifications, such as inserting a new record into a database or updating an existing one.
4. **Arguments**: Mutation fields typically accept arguments that specify the data needed for the operation. For example, to create a new user, a mutation might require the user's name and email as arguments.
5. **Return Types**: Each mutation field has a specified return type, which can be a scalar type, an object type, or a list. The return type defines what data is returned to the client after the mutation operation is performed.

<https://chatgpt.com/share/e5c86bca-4ac9-4a1f-b59f-3e3e8c61f140>

**Resolvers**:

Resolvers in GraphQL are functions that handle the process of fetching the data associated with a specific type or field in your schema. They play a crucial role in determining how data is retrieved and manipulated. When a query is made to a GraphQL server, the server looks at the query and the schema to understand what data is being requested, and then uses resolvers to fetch and return that data.

Here’s a more detailed breakdown:

**How Resolvers Work**

1. **Schema Definition**: You define the schema for your GraphQL API, which includes types, queries, mutations, and subscriptions. Each field in these types can have a resolver associated with it.
2. **Field Resolution**: When a query is executed, each field in the query is resolved by a resolver. If a field doesn't have a specific resolver defined, a default resolver is used. This default resolver simply looks for a property with the same name on the parent object.
3. **Resolver Functions**: Resolvers are just JavaScript functions. They can perform various actions, such as:
   * Fetching data from a database or external API
   * Performing calculations
   * Assembling data from multiple sources
   * Applying business logic
4. **Arguments and Context**: Resolvers can accept four arguments:
   * **Parent**: The result of the previous resolver in the chain.
   * **Args**: The arguments provided in the query for the field.
   * **Context**: A shared object that is passed to all resolvers. It can be used for things like authentication, caching, and configuration.
   * **Info**: Information about the execution state of the query, such as the field name and schema details.

**Schema Definition (schema.graphql)**

type Query {

hello: String

user(id: ID!): User

}

type User {

id: ID!

name: String!

age: Int

}

**Resolver Definition (resolvers.js)**

const resolvers = {

Query: {

hello: () => 'Hello, world!',

user: (parent, args, context, info) => {

// Fetch user by ID from a data source, e.g., database

return context.dataSources.userAPI.getUserById(args.id);

}

},

User: {

age: (parent) => {

// Compute or fetch the age of the user

return parent.birthYear ? new Date().getFullYear() - parent.birthYear : null;

}

}

};

module.exports = resolvers;

**Server Setup (server.js)**

const { ApolloServer } = require('apollo-server');

const typeDefs = require('./schema.graphql');

const resolvers = require('./resolvers');

const server = new ApolloServer({

typeDefs,

resolvers,

context: () => {

// Provide context to resolvers, e.g., data sources

return {

dataSources: {

userAPI: new UserAPI()

}

};

}

});

server.listen().then(({ url }) => {

console.log(`Server ready at ${url}`);

});

<https://chatgpt.com/share/d66055d3-b328-42b3-893a-2079688ee87b>

**TypeDefinations**:

In GraphQL, a type definition specifies the structure of data you can query. It defines the fields that can be requested and the types of those fields. Here's a basic example to illustrate the concept:

type Book {

title: String

author: Author

publishedYear: Int }

In this example, the Book type has three fields:

* title of type String
* author of type Author
* publishedYear of type Int

**Scalar Types**

GraphQL has several built-in scalar types:

* Int (a signed 32‐bit integer)
* Float (a signed double‐precision floating‐point value)
* String (a UTF‐8 character sequence)
* Boolean (true or false)
* ID (a unique identifier often used for refetching an object or as the key for a cache)

**Custom Object Types**

You can define custom object types like Author:

type Author {

name: String

books: [Book]

}

Here, the Author type has two fields:

* name of type String
* books, which is a list of Book objects

### Enum Types

Enums are special types that restrict a field to a particular set of values:

enum Genre {

FICTION

NONFICTION

SCIFI

FANTASY

}

### Input Types

Input types are used to define the structure of inputs for mutations:

input BookInput {

title: String

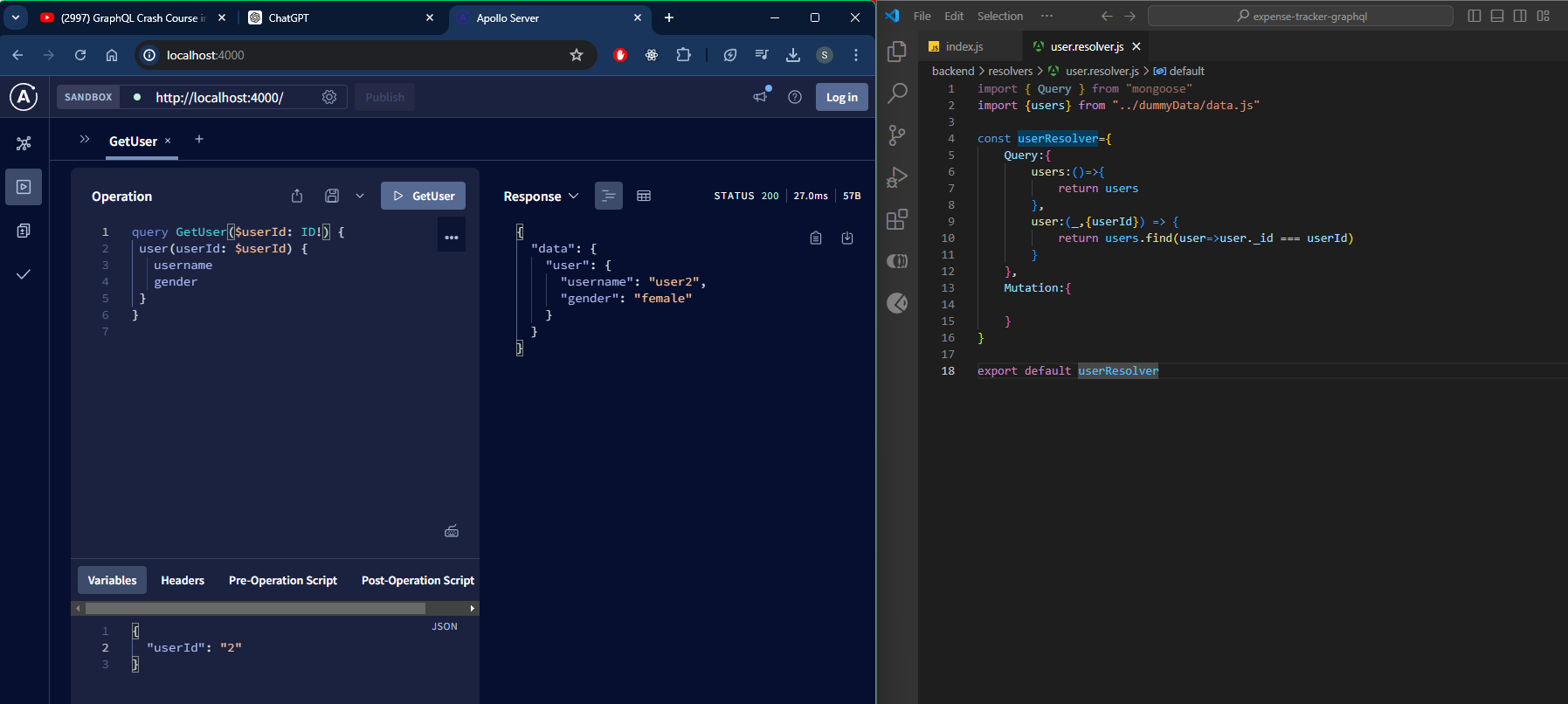
author: String

publishedYear: Int

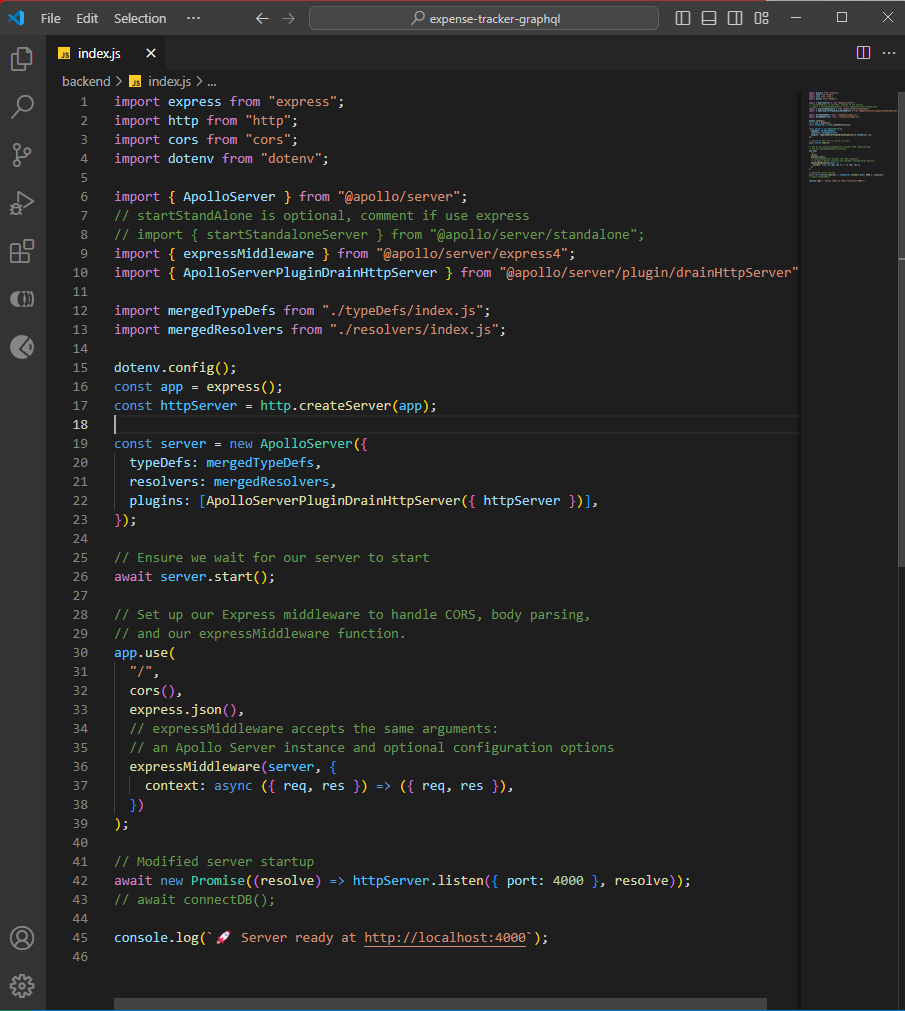
}

<https://chatgpt.com/share/4bb6c8f6-62c5-4392-b767-2af8906d84d8>

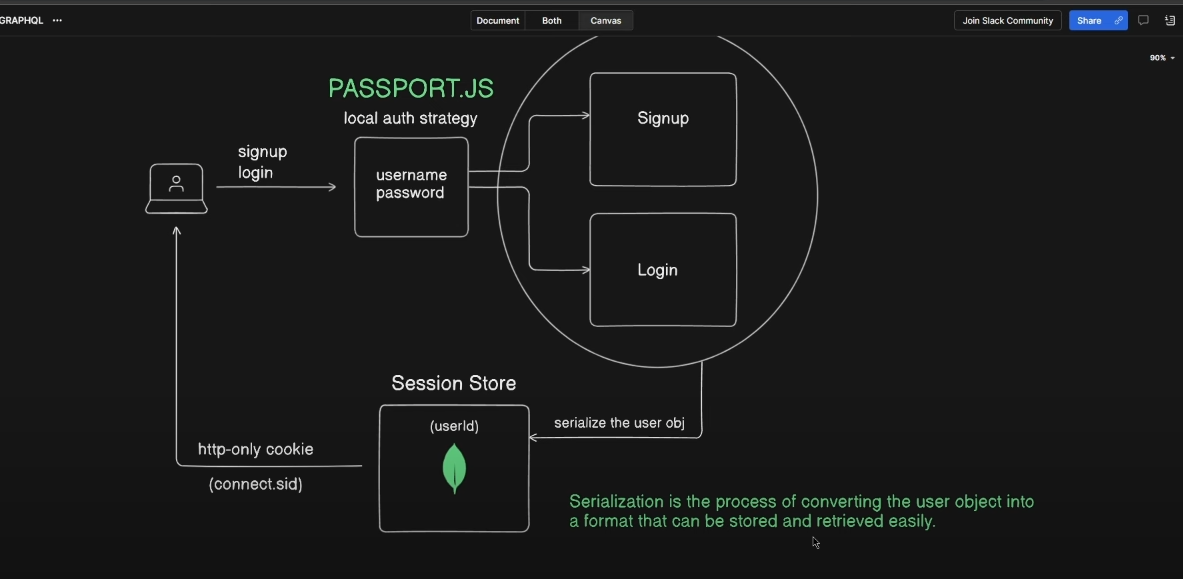
How to use the Appollo UI to query (Think it like a postman for graphql):



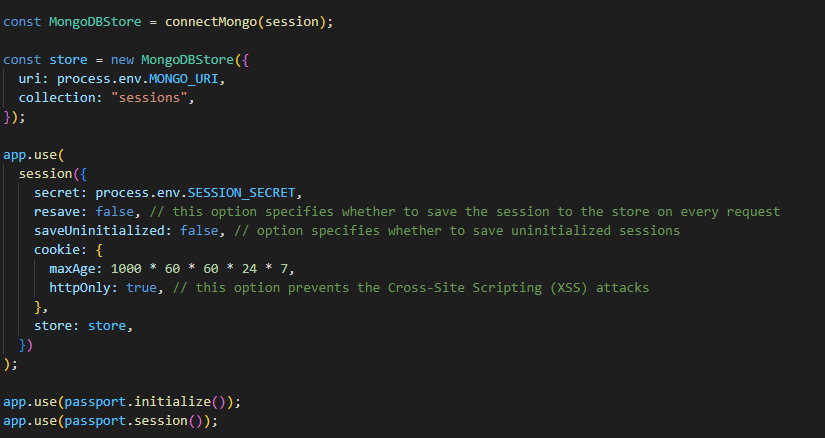
Converted the app from a standalone server to node express based.



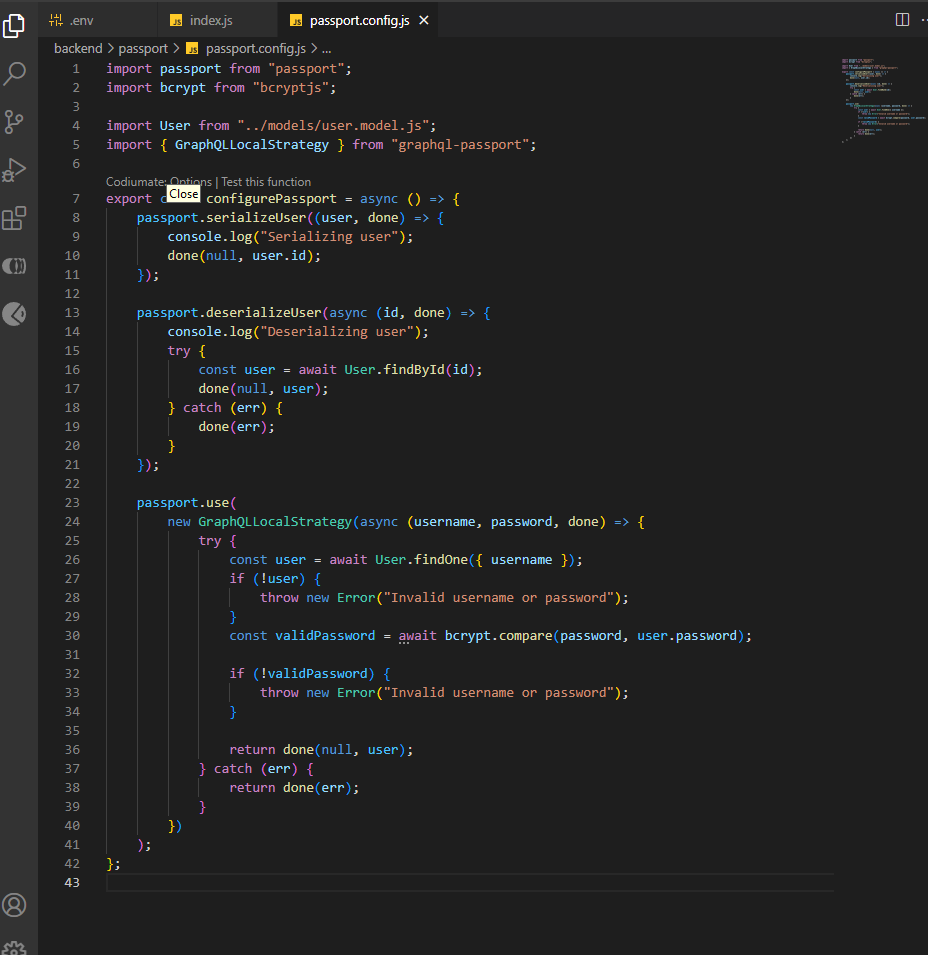
Authentication Flow Chart:



Configure the things in the index js:



Inside passport config js file:



**Serialise vs Deseialise in Passport JS:**

In Passport.js, a popular authentication middleware for Node.js, serialization and deserialization are essential concepts for managing user sessions. They allow Passport.js to handle user information efficiently when using session-based authentication.

### Serialization

**Serialization** is the process of converting a user object into a unique identifier (typically a user ID) that can be stored in the session. This helps reduce the amount of data stored in the session and improves performance. Here's an example:

**passport.serializeUser((user, done) => {**

**done(null, user.id); // user.id is the unique identifier**

**});**

In this function, user is the authenticated user object, and done is a callback function. The first argument of done is an error object (null if there is no error), and the second argument is the user ID that will be stored in the session.

### Deserialization

**Deserialization** is the process of converting the stored unique identifier back into a full user object. This is necessary because, during subsequent requests, you will need the full user details rather than just the ID. Here's an example:

**passport.deserializeUser((id, done) => {**

**User.findById(id, (err, user) => {**

**done(err, user);**

**});**

**});**

In this function, id is the unique identifier stored in the session, and done is a callback function. The first argument of done is an error object (if any), and the second argument is the full user object retrieved from the database (or another storage mechanism).

**Example Workflow**

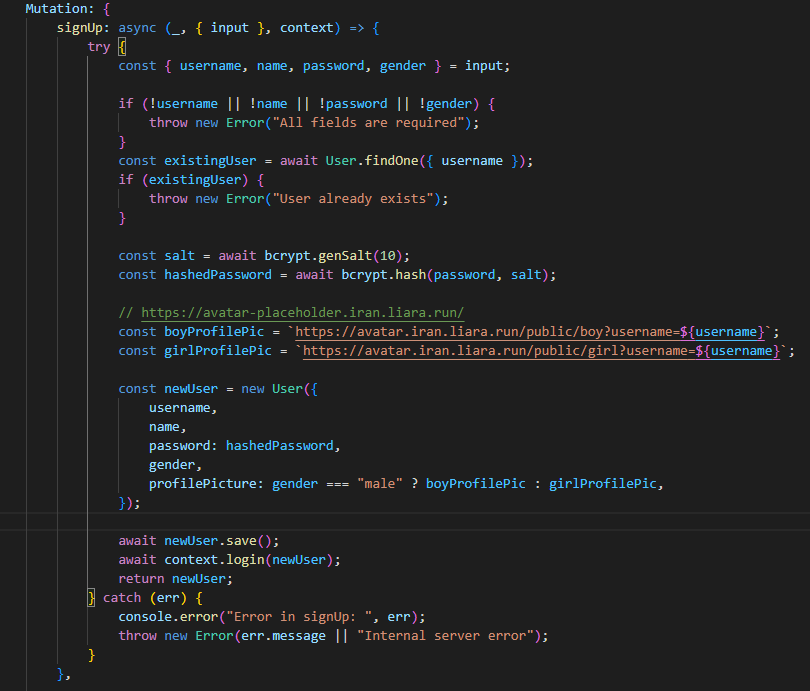
1. **User Authentication**: When a user logs in, Passport authenticates the user.
2. **Serialization**: After successful authentication, Passport serializes the user (converts the user object to a user ID) and stores the user ID in the session.
3. **Session Storage**: The session middleware (like express-session) stores the user ID in a session cookie.
4. **Subsequent Requests**: For every subsequent request, the session middleware retrieves the user ID from the session cookie.
5. **Deserialization**: Passport deserializes the user (converts the user ID back to the full user object) for use in the request.

[**https://chatgpt.com/share/d61cd9f5-0906-4eb4-a40d-98fe5b85608a**](https://chatgpt.com/share/d61cd9f5-0906-4eb4-a40d-98fe5b85608a)

**User Resolvers implementation:**This resolver file handles GraphQL mutations and queries related to user authentication and user data management. It uses the User and Transaction models, as well as bcryptjs for password hashing. Here's a step-by-step breakdown:

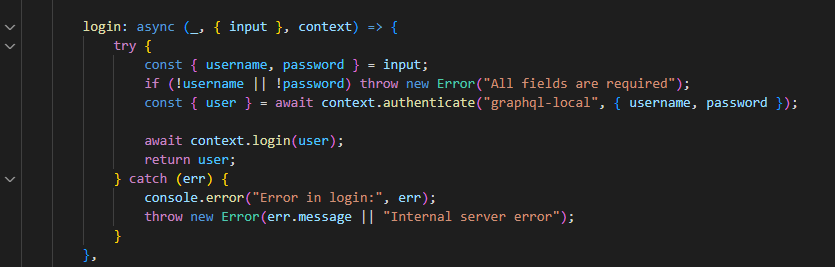
### Mutations

#### signUp



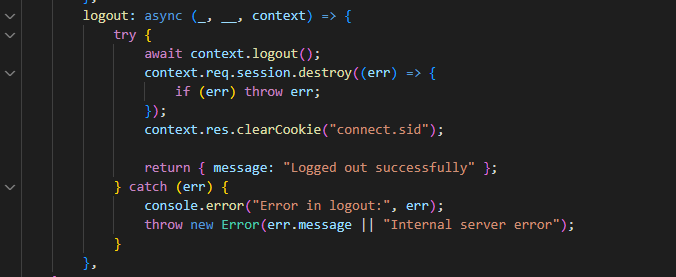
1. **Input Validation**: Checks if username, name, password, and gender fields are provided. If any are missing, it throws an error.
2. **User Existence Check**: Searches the database to see if a user with the given username already exists. If so, it throws an error.
3. **Password Hashing**: Generates a salt and hashes the password using bcrypt.
4. **Profile Picture Assignment**: Assigns a profile picture URL based on the gender field.
5. **User Creation**: Creates a new User object with the provided details and the hashed password.
6. **Saving User**: Saves the new user to the database.
7. **Logging In**: Logs in the newly created user using the context.login method.
8. **Return**: Returns the newly created user.

#### Login



1. **Input Validation**: Checks if username and password fields are provided. If not, it throws an error.
2. **Authentication**: Uses context.authenticate to authenticate the user with the given credentials.
3. **Logging In**: Logs in the authenticated user using the context.login method.
4. **Return**: Returns the authenticated user.

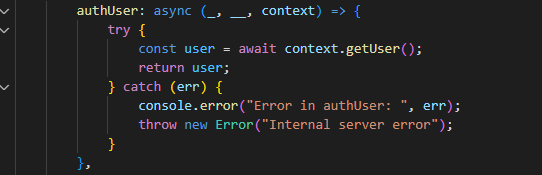
#### Logout



1. **Logout**: Uses context.logout to log out the current user.
2. **Session Destruction**: Destroys the session and clears the authentication cookie.
3. **Return**: Returns a success message.

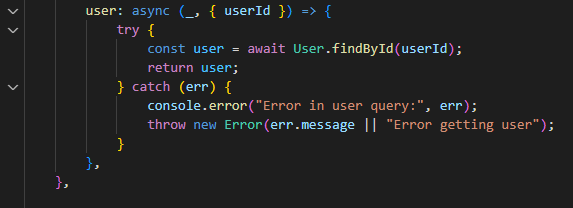
### Queries

#### authUser



1. **Get Authenticated User**: Retrieves the currently authenticated user using context.getUser.
2. **Return**: Returns the authenticated user.

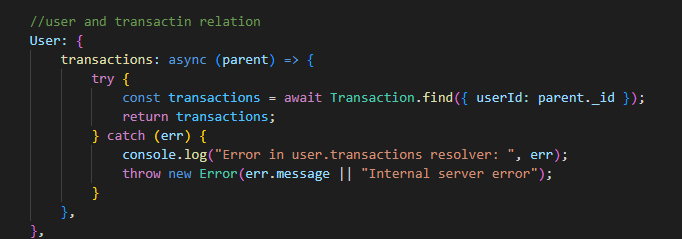
#### User



1. **Find User by ID**: Searches for a user in the database using the provided userId.
2. **Return**: Returns the found user.

### User Sub-Resolver

#### Transactions



1. **Find Transactions by User ID**: Finds all transactions related to the user by searching for transactions with the userId equal to the parent's \_id.
2. **Return**: Returns the list of transactions.

### Error Handling

Throughout the resolver, errors are caught and logged, and appropriate error messages are thrown.

### Summary

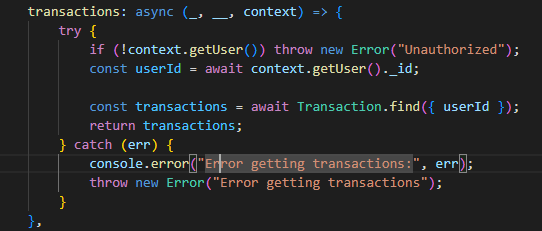
* **signUp Mutation**: Handles user registration, including validation, password hashing, and logging in the new user.
* **login Mutation**: Manages user login by validating credentials and logging in the user.
* **logout Mutation**: Handles user logout, including session destruction and cookie clearing.
* **authUser Query**: Retrieves the currently authenticated user.
* **user Query**: Retrieves a user by their ID.
* **User.transactions Sub-Resolver**: Retrieves all transactions associated with a user.

This resolver file provides comprehensive user authentication and user data retrieval functionalities within a GraphQL API.

**Transaction Resolver Implementation:**

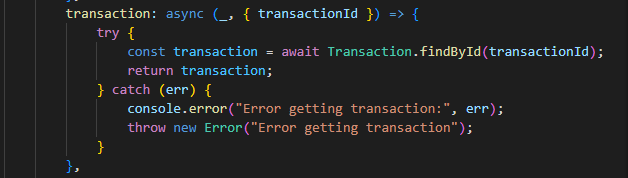
### Query Resolvers

#### Transactions



* This resolver fetches all transactions for the authenticated user.
* It checks if the user is authenticated by calling context.getUser().
* If the user is authenticated, it retrieves the user's ID and finds all transactions in the database that belong to that user.

#### transaction



* This resolver fetches a single transaction by its ID.
* It retrieves the transaction from the database using the transactionId parameter.

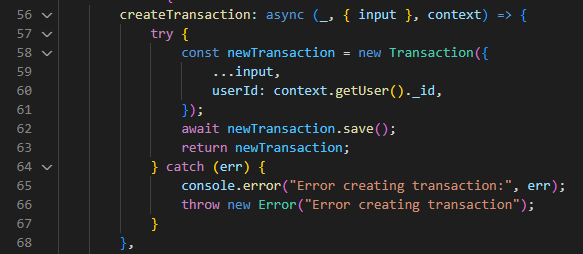
#### categoryStatistics



* This resolver calculates statistics for each transaction category for the authenticated user.
* It checks if the user is authenticated.
* If authenticated, it retrieves all transactions for the user and sums the amounts for each category.
* The result is an array of objects, each containing a category and the total amount for that category.

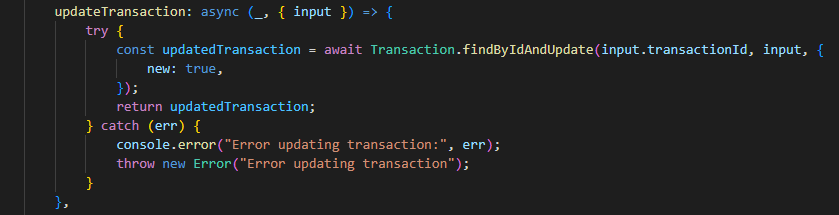
### Mutation Resolvers

#### createTransaction



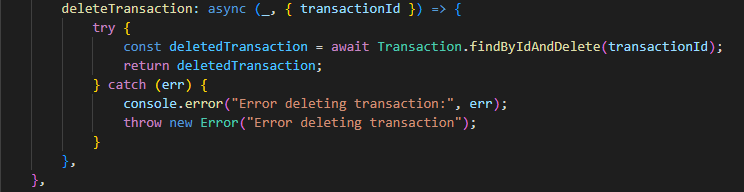
* This resolver creates a new transaction.
* It takes input as an argument, which contains the transaction details.
* The user ID is retrieved from the context and added to the new transaction.
* The new transaction is saved to the database and returned.

#### updateTransaction



* This resolver updates an existing transaction.
* It takes input as an argument, which contains the transaction details and the transaction ID.
* The transaction is updated in the database using findByIdAndUpdate and the updated transaction is returned

#### deleteTransaction



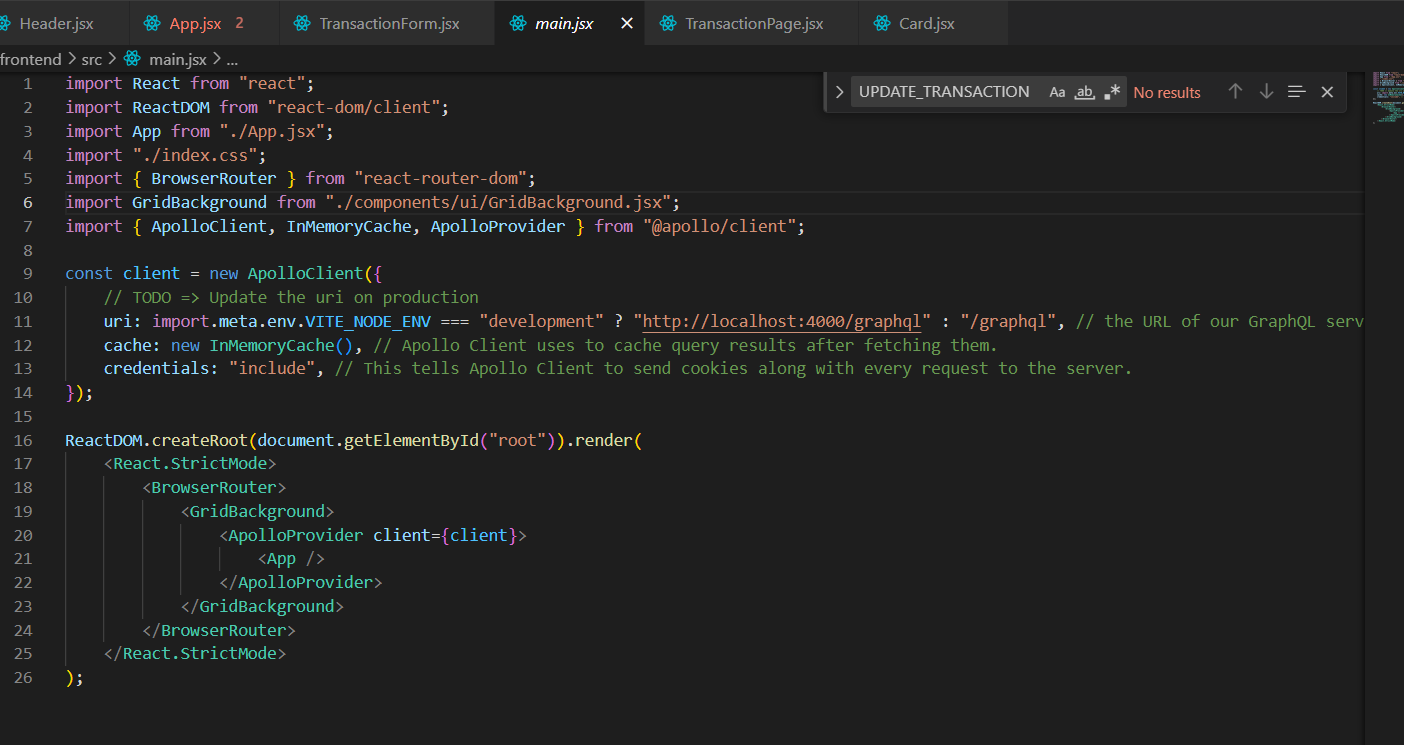
* This resolver deletes a transaction by its ID.
* It takes transactionId as an argument.
* The transaction is deleted from the database using findByIdAndDelete and the deleted transaction is returned.

**Transaction Type Resolver**

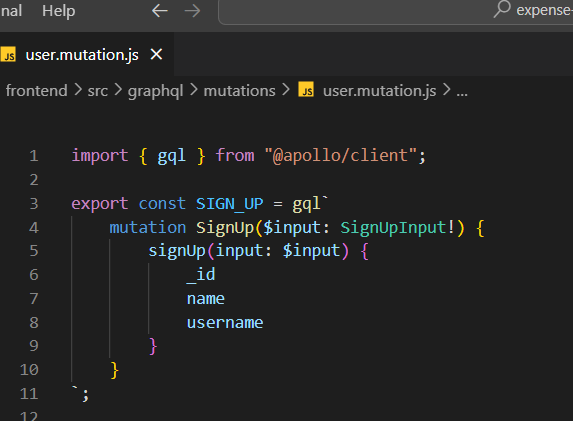
****

* This resolver resolves the user field for the Transaction type.
* It retrieves the user associated with the transaction from the database using the userId field of the transaction.

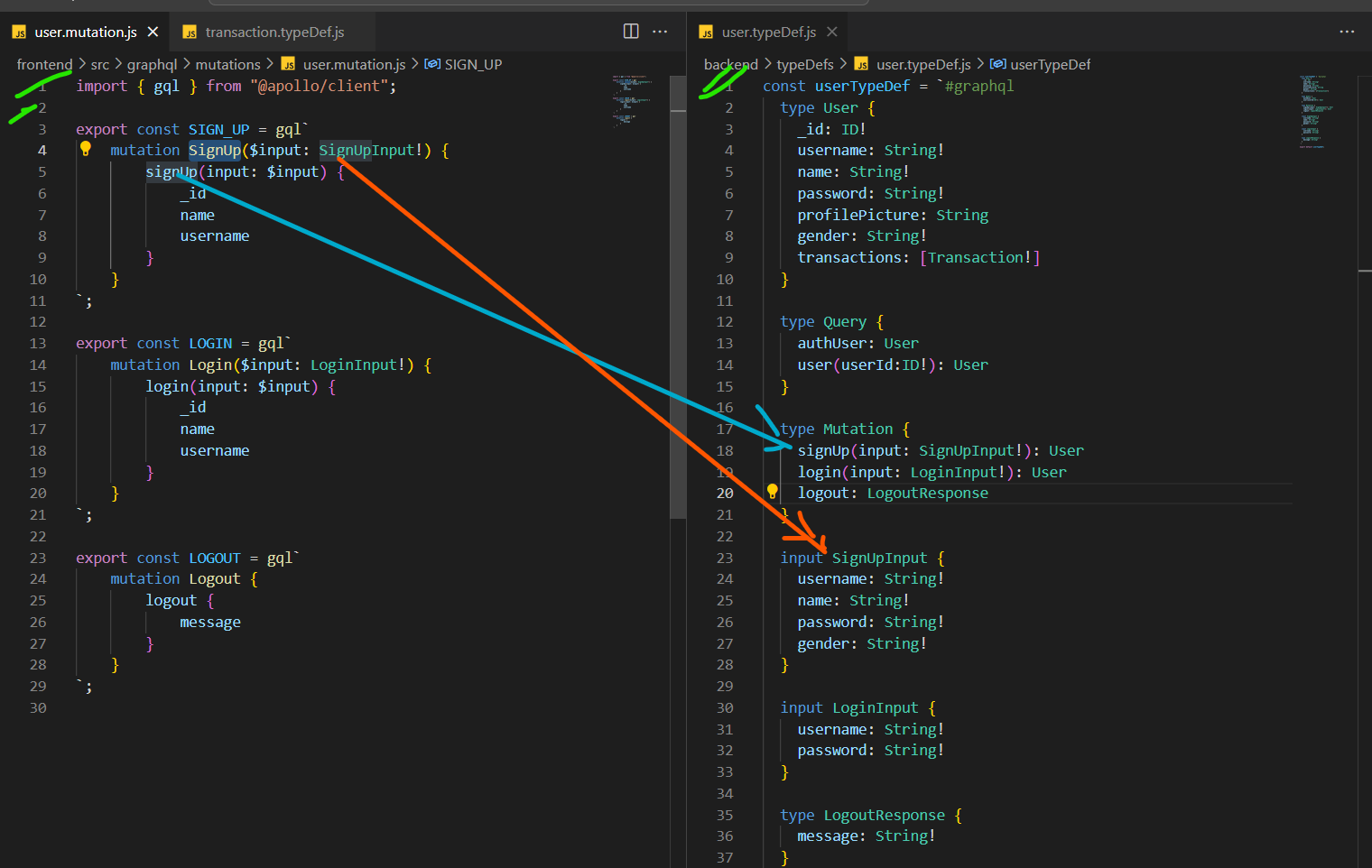
Connecting Apollo Client on Frontend:

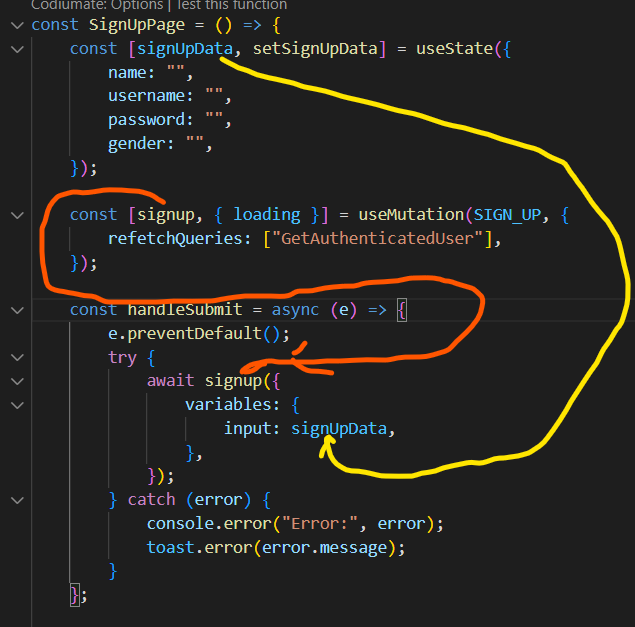


**Implementing Authentication**:  
- Create a SIGN\_UP mutation:



* **mutation SignUp($input: SignUpInput!) {** defines a GraphQL mutation named SignUp. The mutation accepts a single variable called $input, which is of the type SignUpInput. The exclamation mark (!) indicates that this input is required.
* **signUp(input: $input) {** Within the mutation, the signUp operation is called. The signUp operation takes an argument input whose value is provided by the $input variable.

****

**** **State Initialization**:

const [signUpData, setSignUpData] = useState({

name: "",

username: "",

password: "",

gender: "",

});

* useState is a React hook used to initialize and manage state within functional components.
* signUpData is the state variable holding the sign-up form data, and setSignUpData is the function to update this state.
* The state is initialized with an object containing empty strings for name, username, password, and gender.

 **GraphQL Mutation Hook**:

const [signup, { loading }] = useMutation(SIGN\_UP, {

refetchQueries: ["GetAuthenticatedUser"],

});

* useMutation is a hook from Apollo Client used to define a GraphQL mutation.
* signup is the function that executes the SIGN\_UP mutation.
* loading is a boolean indicating if the mutation is in progress.
* SIGN\_UP is the GraphQL mutation document (usually defined elsewhere in the code).
* refetchQueries is an option that specifies which queries to refetch after the mutation completes. Here, it refetches the GetAuthenticatedUser query to update the user data after sign-up.

 **Form Submission Handler**:

const handleSubmit = async (e) => {

e.preventDefault();

try {

await signup({

variables: {

input: signUpData,

},

});

} catch (error) {

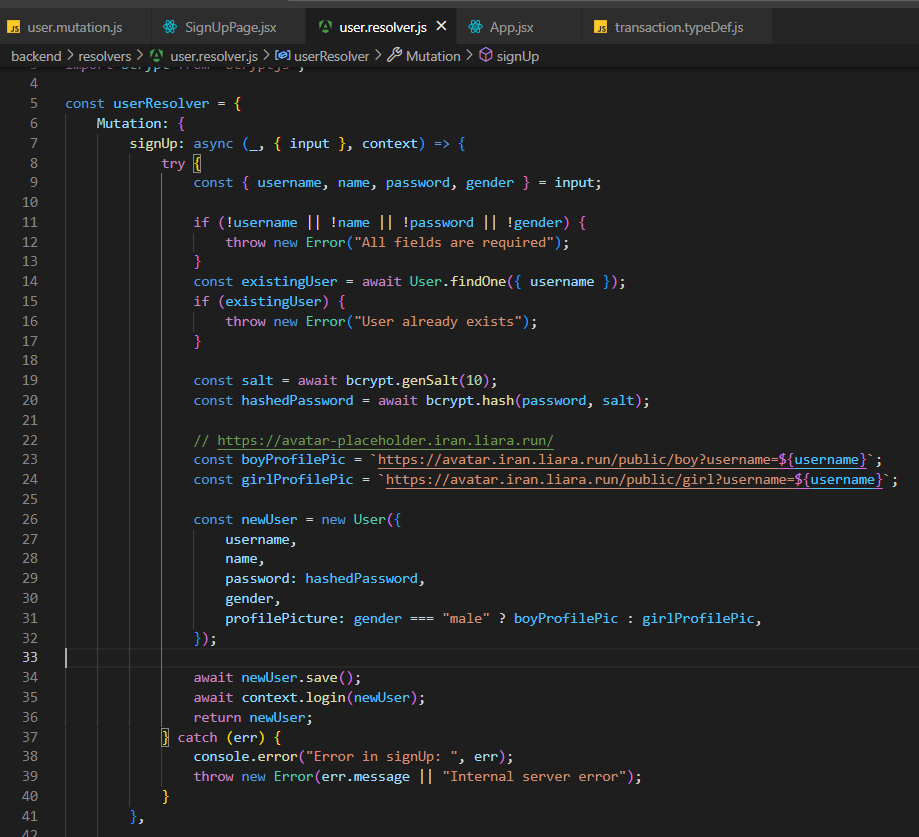
console.error("Error:", error);

toast.error(error.message);

}

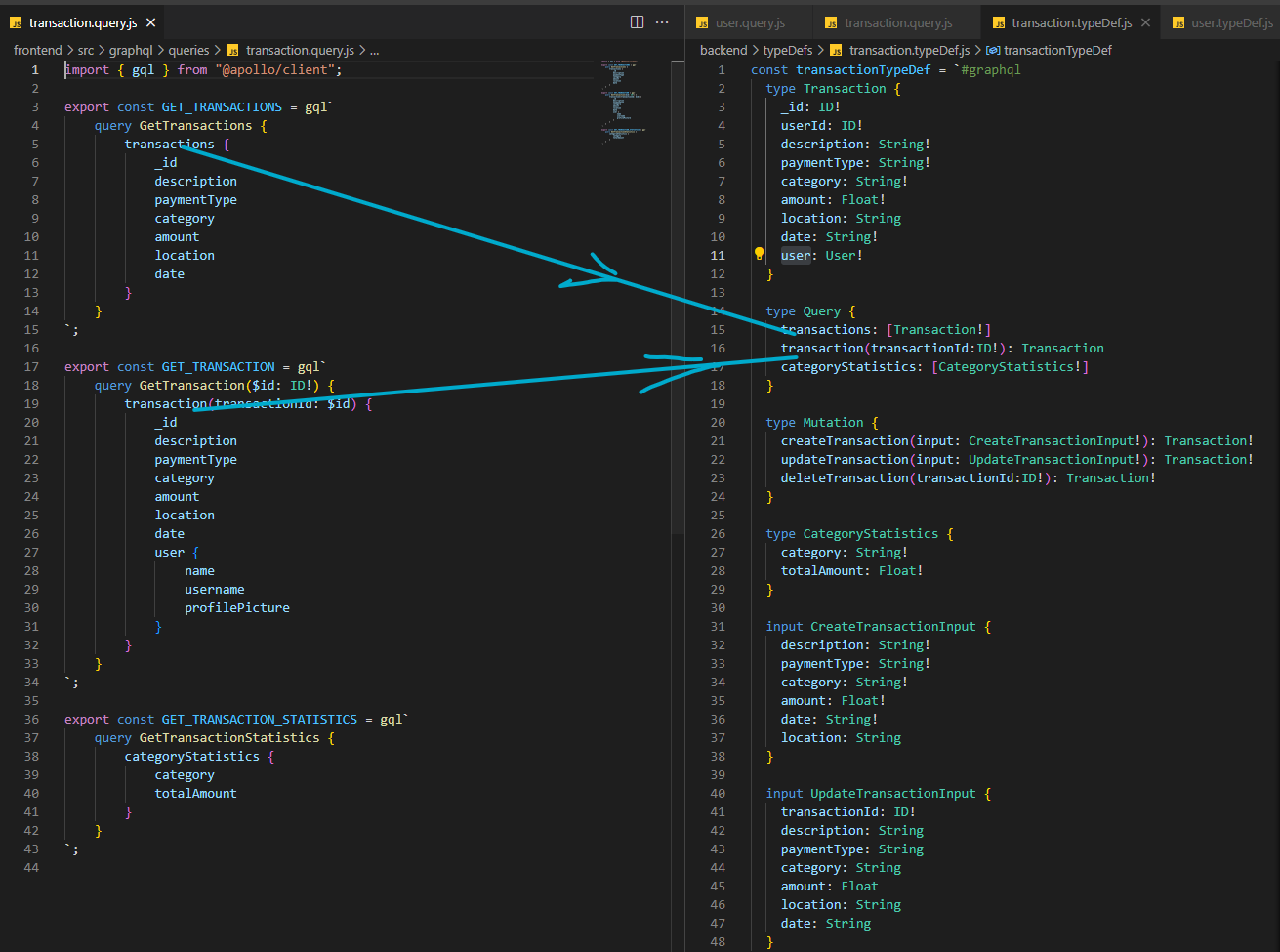
};

* handleSubmit is an asynchronous function that handles the form submission event.
* e.preventDefault() prevents the default form submission behavior, which is typically a page reload.
* signup function is called with the signUpData as variables for the mutation.
* If the mutation is successful, the code continues without issue.
* If an error occurs during the mutation, it is caught in the catch block, logged to the console, and displayed to the user using toast.error.
* **Then we are referring the resolver from the backend**

****

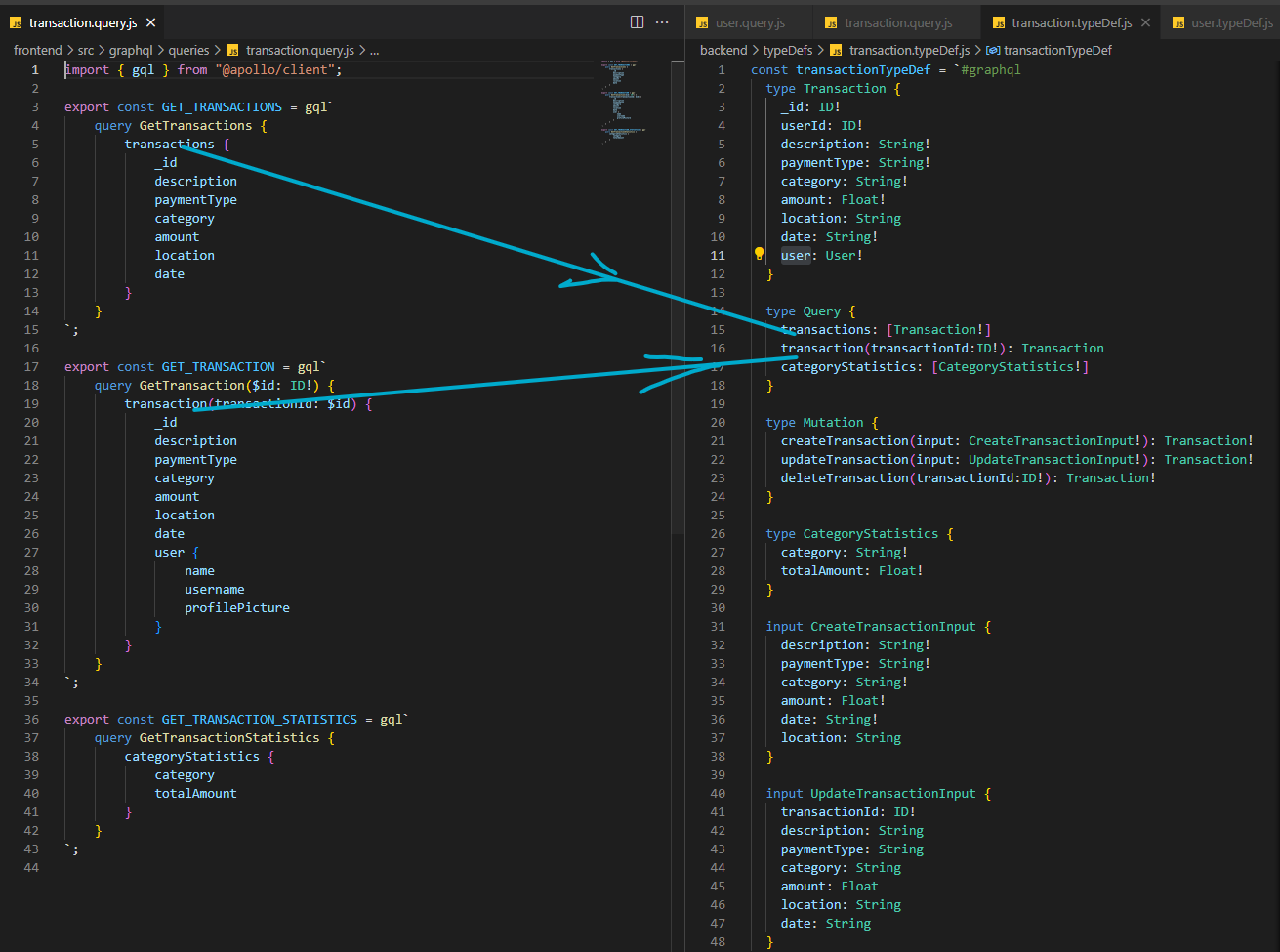
**CRUD Operations in Transactions:**

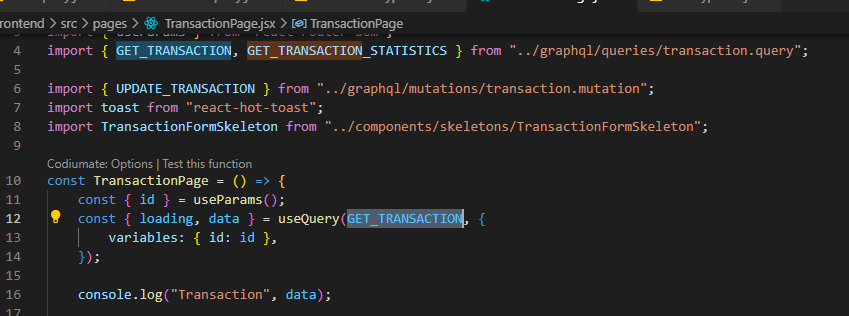
**Read-**

****

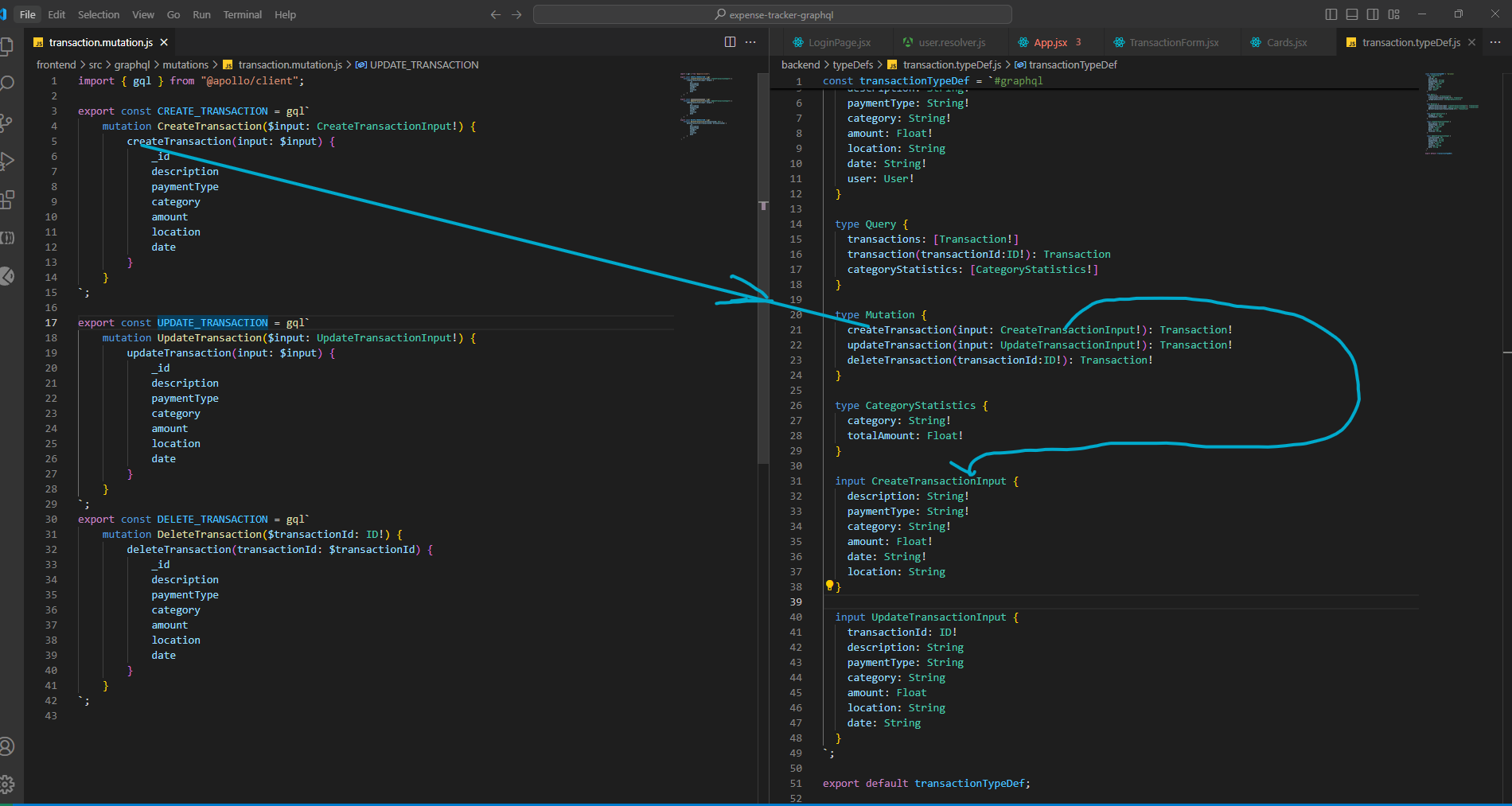
****

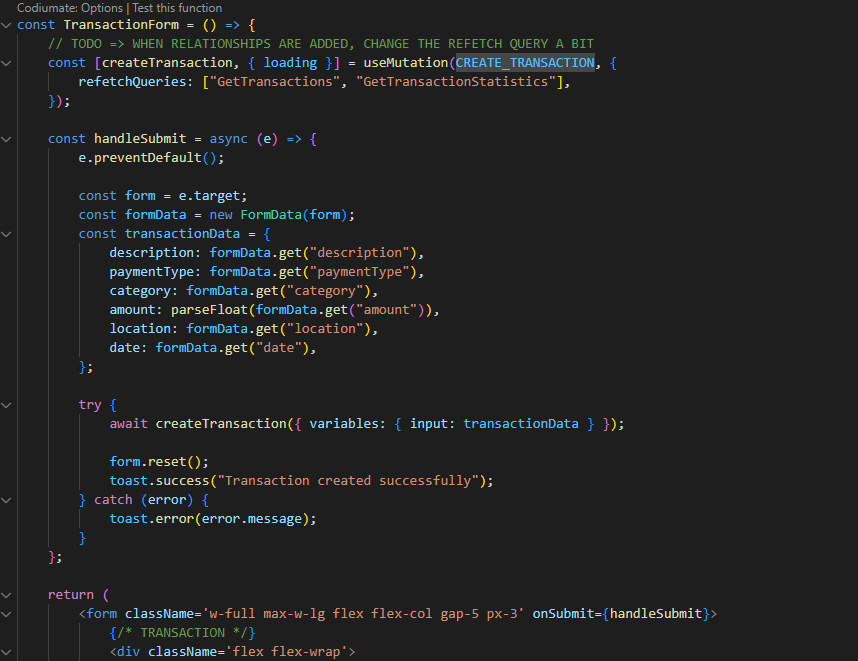
**Read with param-**

****

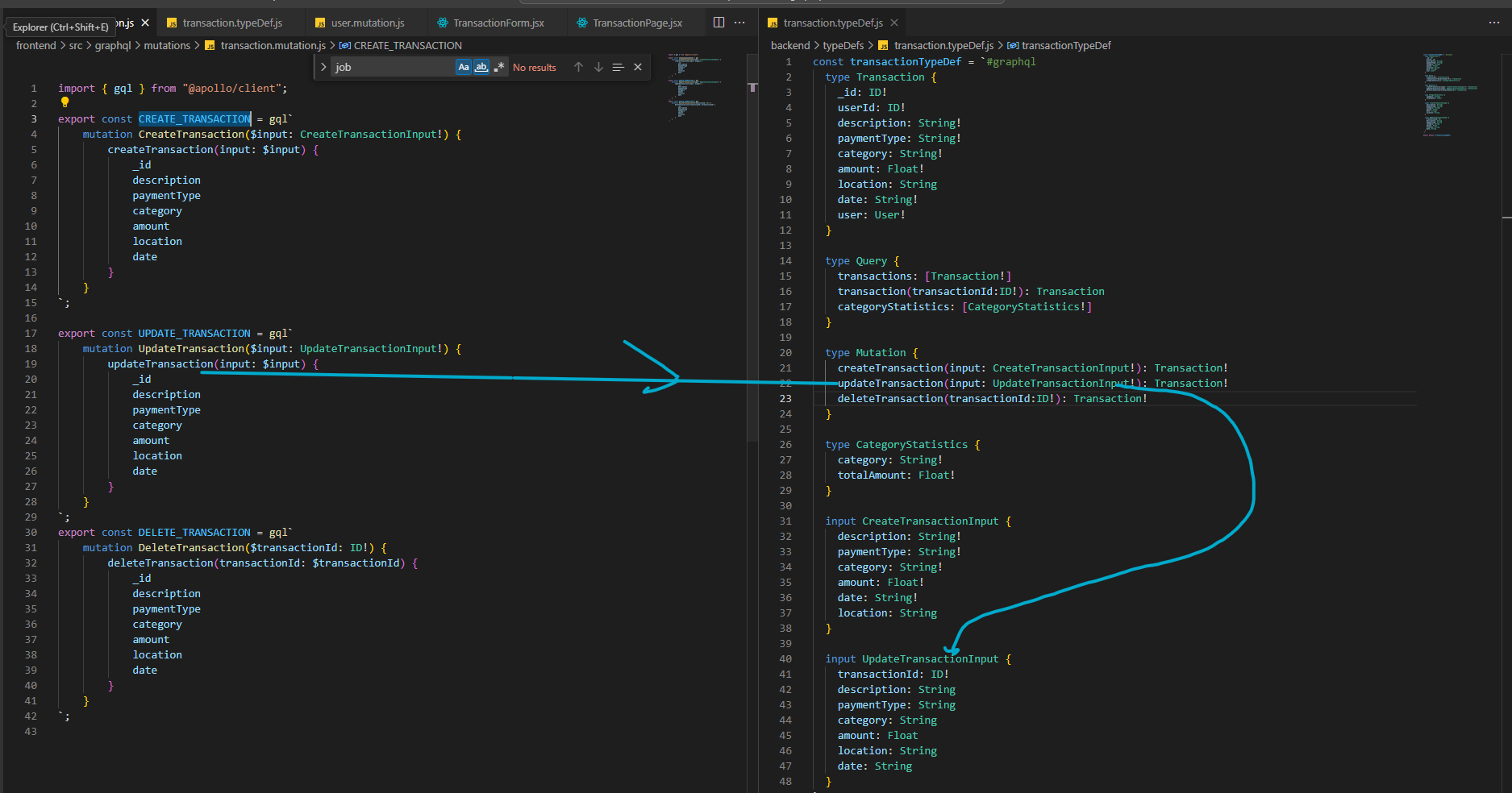
****

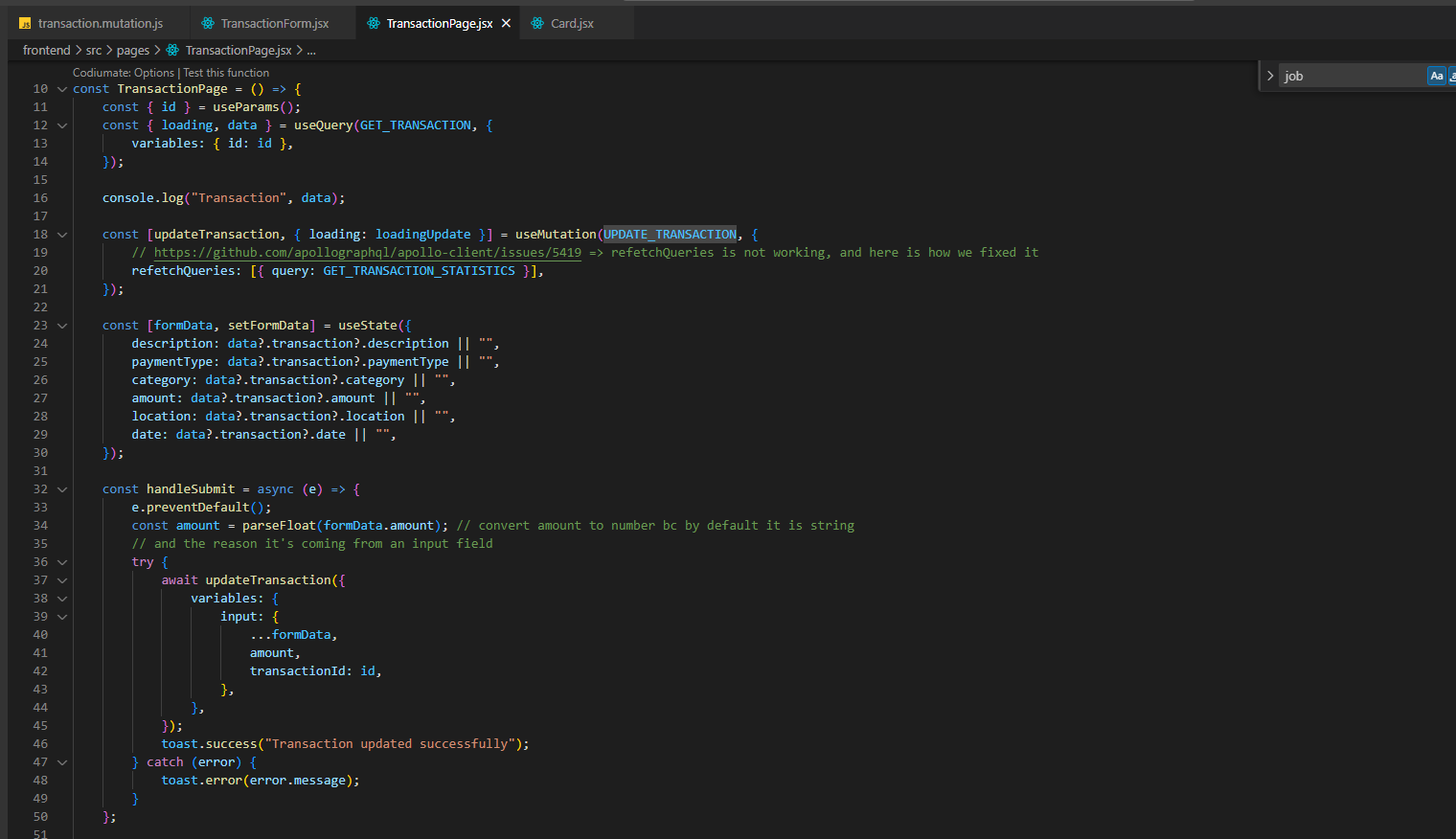
**Create-**

****

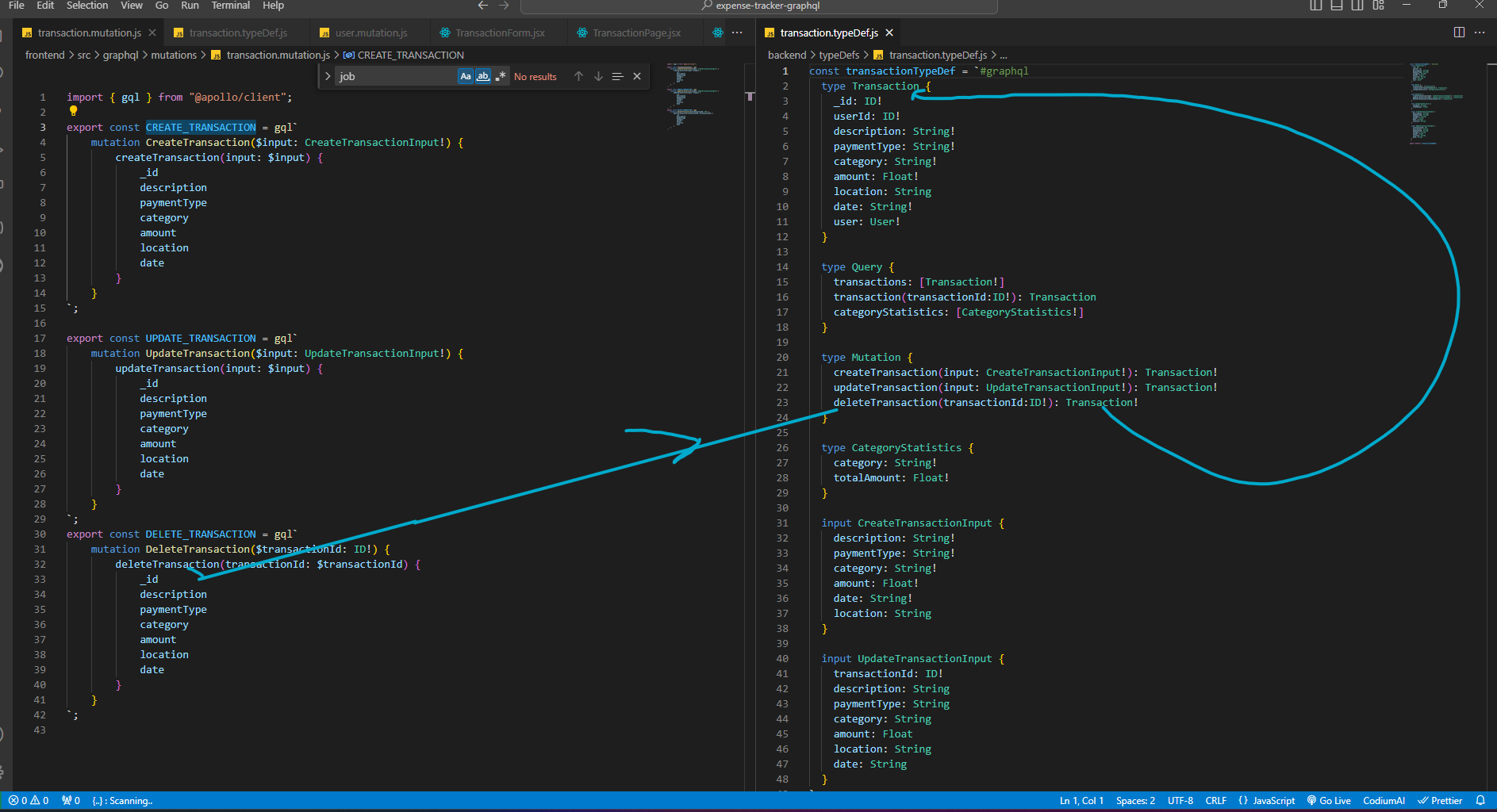
****

**Update Transaction:**

****

****

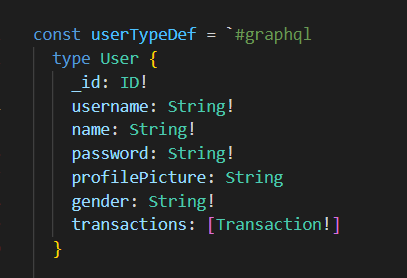
**Delete Transaction:**

****

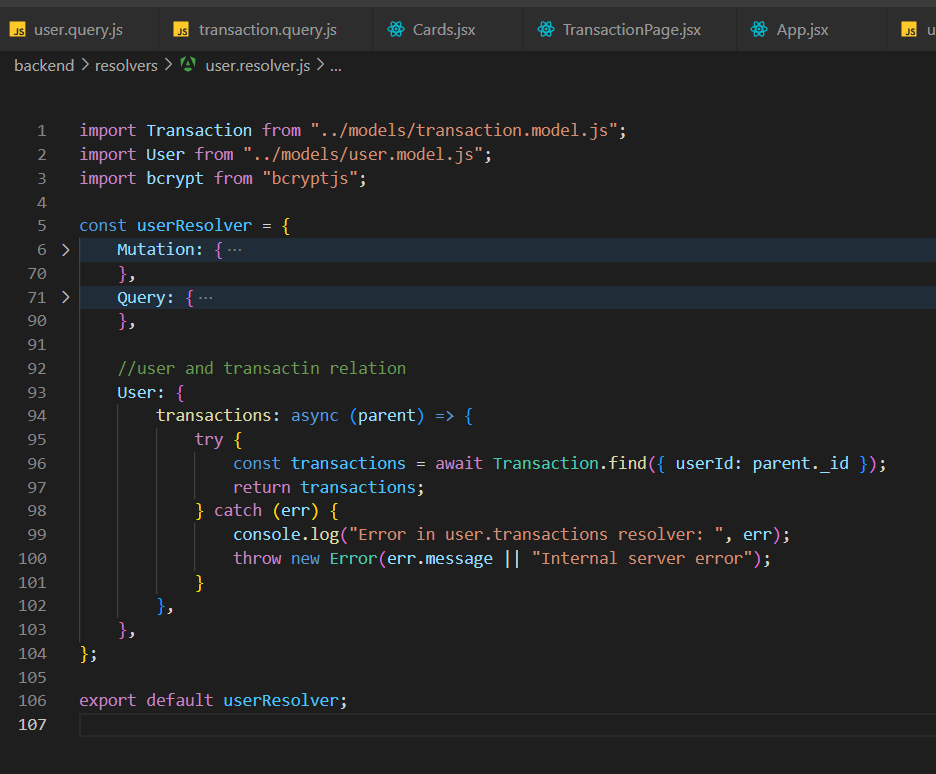
****

**Establishing Relationships In GraphQL:**

* Add the required type to the typedef field. In this case it is **transactions:[Transaction!]**



* Add the special type in the resolvers



* In the above code the ***parent*** argument inside the callback function represents the parent object or in this code it is the **User** entity not the **Transaction** entity.

In GraphQL, resolvers can be nested. This means that fields within a type can have their own resolvers to fetch and resolve data. In this case, the User type has a nested field transactions that needs to be resolved.

### transactions Resolver

#### Purpose

The purpose of the transactions resolver is to fetch and return all transactions associated with a specific user. This is typically used in a scenario where you want to retrieve a user along with their related transactions in a single query.

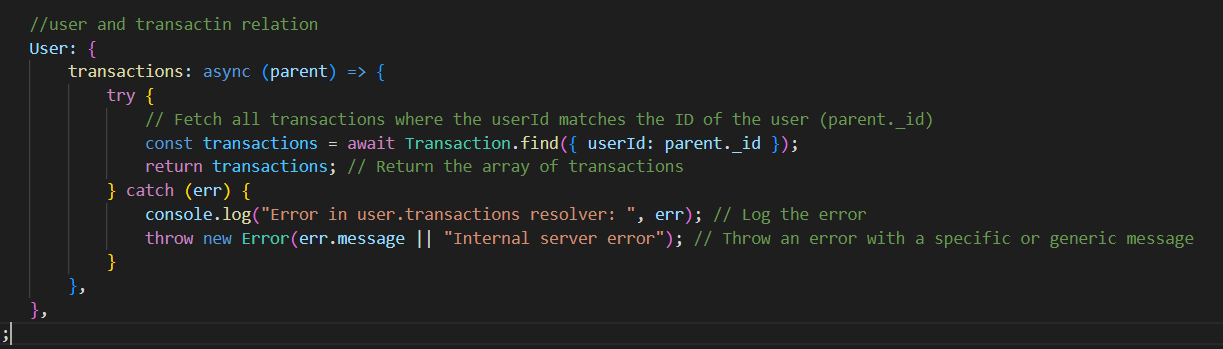
#### Parameters

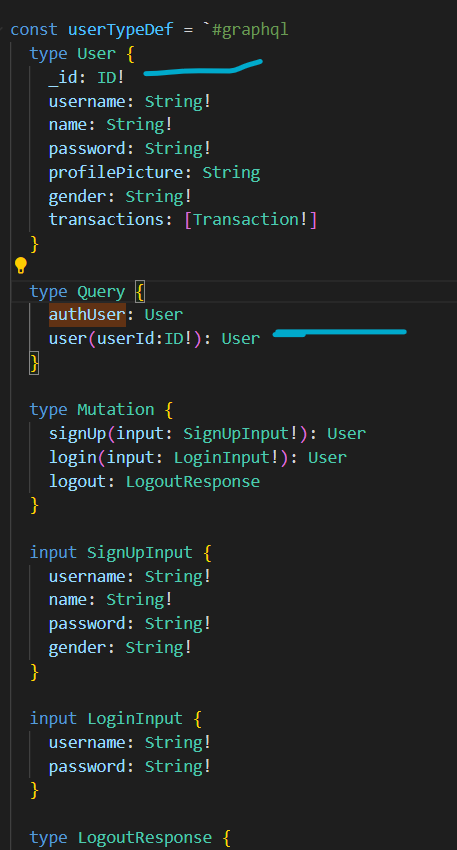
* parent: This parameter represents the resolved value of the parent field, which in this case is a User object. The parent object contains all the fields of the User that have already been resolved.

#### Implementation

The transactions resolver is an asynchronous function that performs the following steps:

1. **Extract User ID**
   * The parent object represents a User instance. To find the transactions related to this user, we need to extract the user's ID, which is available as parent.\_id.
2. **Fetch Transactions**
   * The resolver uses Mongoose's Transaction.find method to query the database and find all transactions where the userId matches the user's ID (parent.\_id). This returns an array of transaction documents related to the user.
3. **Return Transactions**
   * The resolver returns the array of transactions found.
4. **Error Handling**
   * A try-catch block is used to handle any potential errors that may occur during the database query.
   * If an error occurs, it is logged to the console, and an error is thrown with a message indicating either the specific error message or a generic "Internal server error".





* In the respective component we are passing the variables so that it will act as the variable for respective queries.

 const { data: userAndTransactions } = useQuery(GET\_USER\_AND\_TRANSACTIONS, {

    variables: {

      userId: authUser?.authUser?.\_id,

    },

  });



**Deploy using Render.com:**

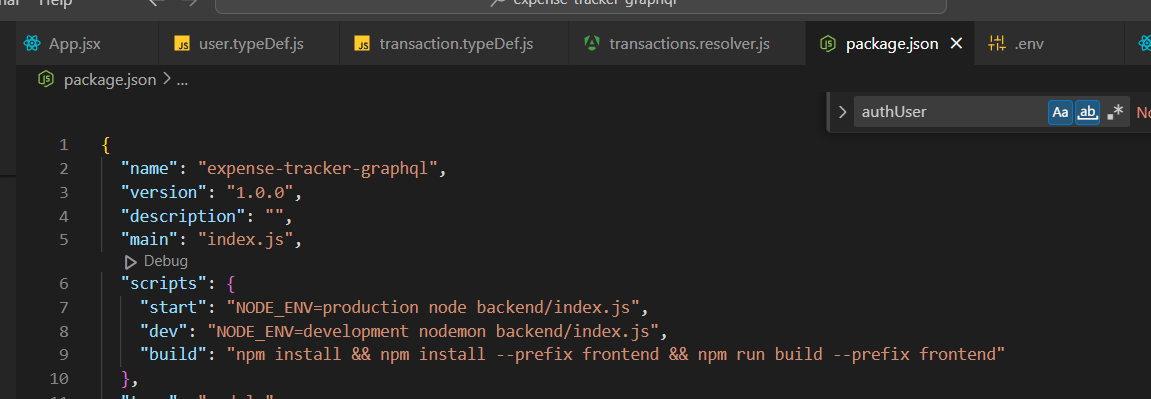
**Make sure to build the FE first.**

**Indexjs backend**

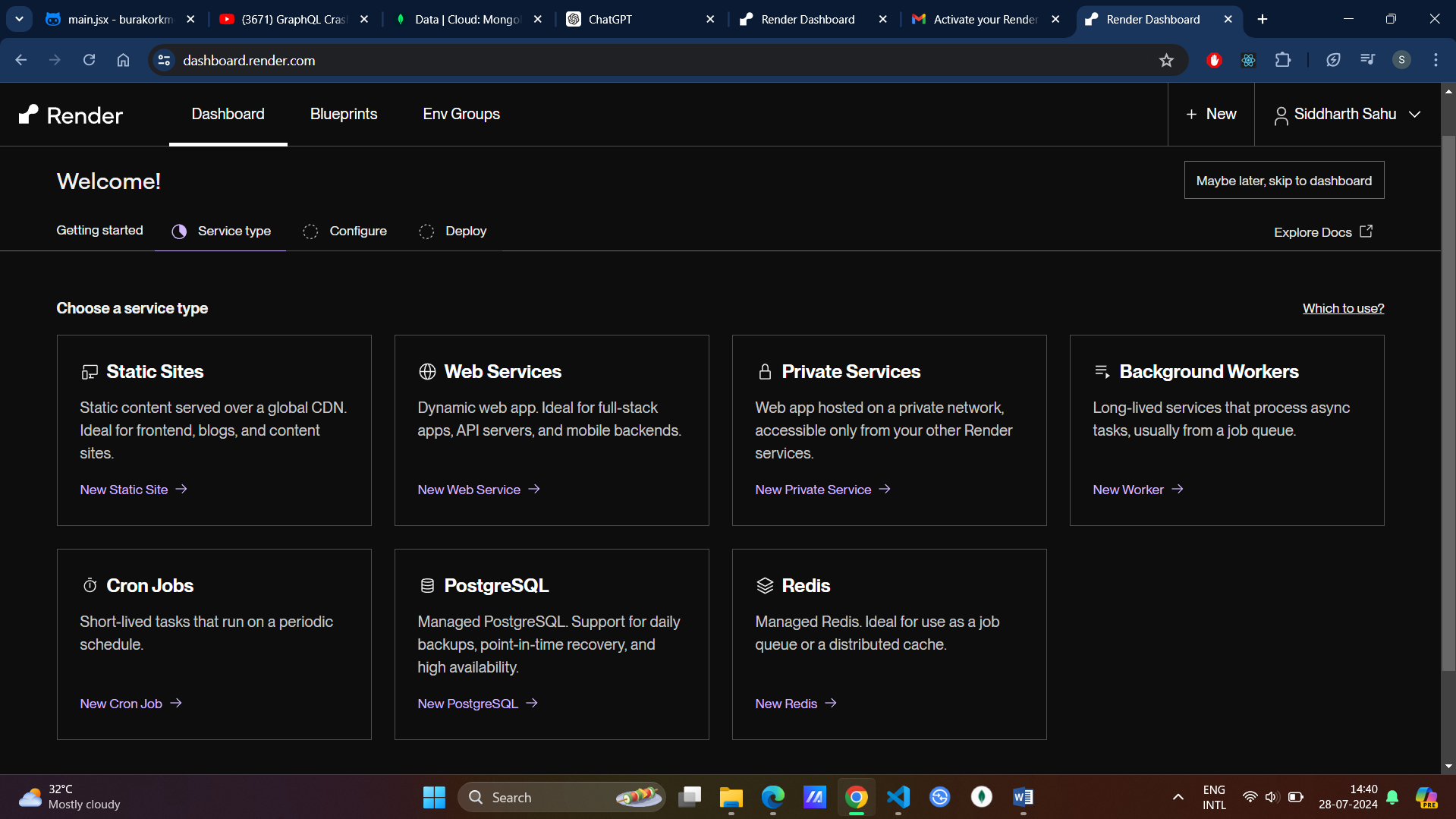
****

****

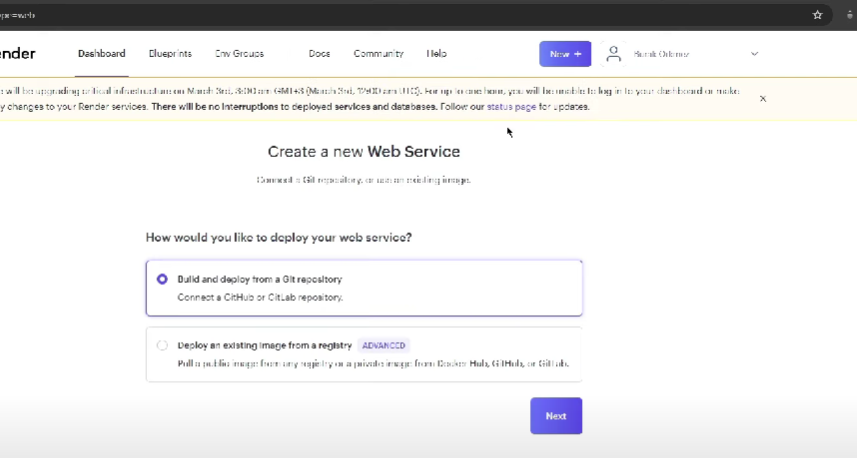
**Package json backend**

****

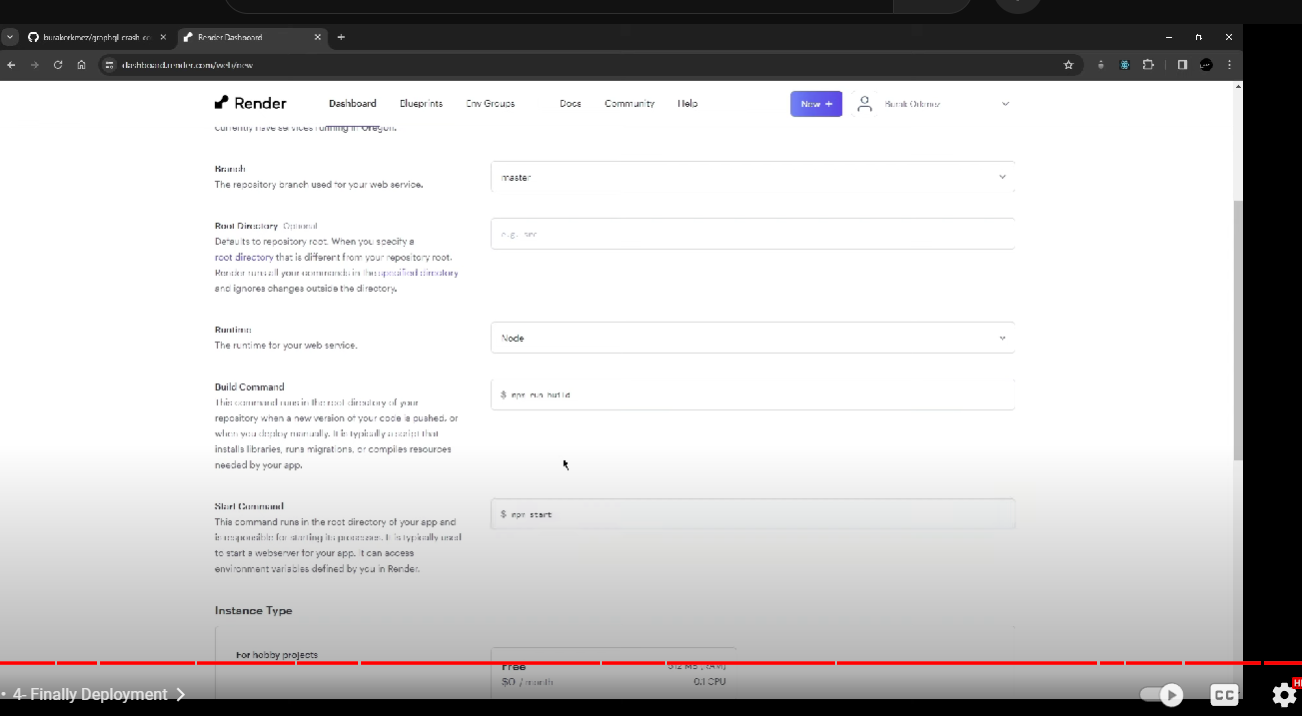
**Used github account for cron account:**

****

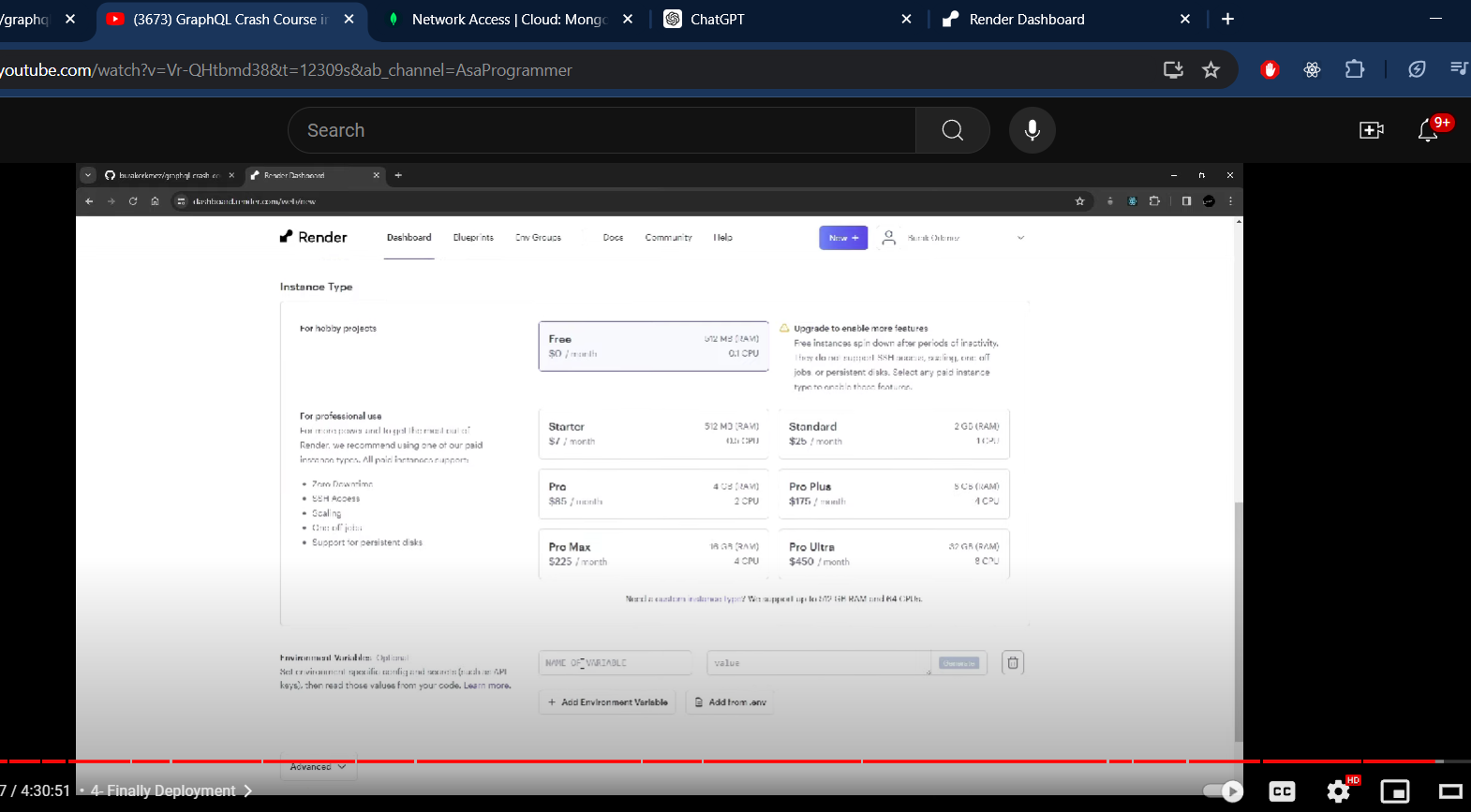
**Click on New -> Web Service**

****

**Connect Repo and Select the one then select and change build command npm run build**

****

**Add Environment variables: backend added in the tutorial no fe. Post this it will give public uri**

****