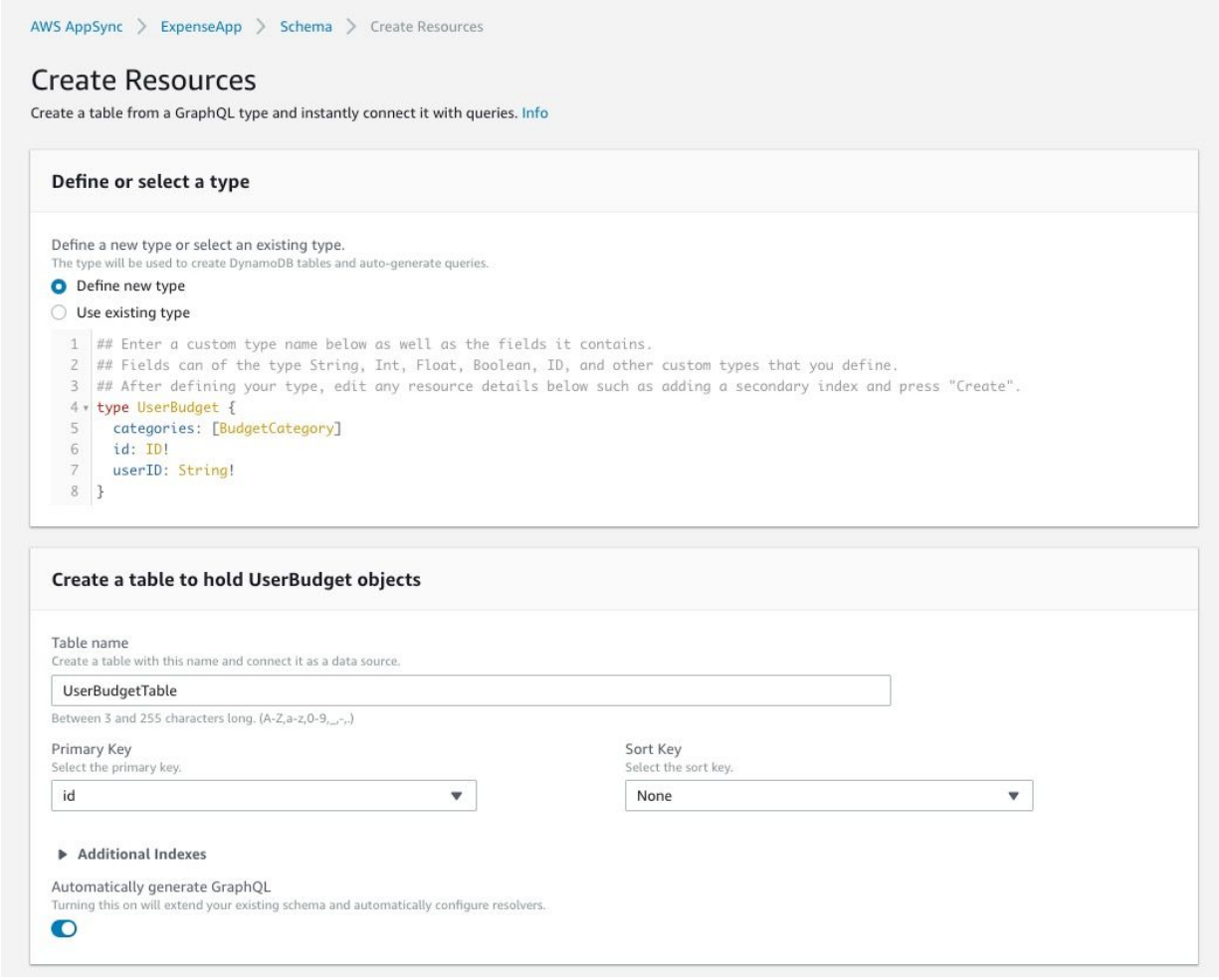
Milestone #4
11/15/18
CSCI 3308
Lab Section 204
afterschoolspecials

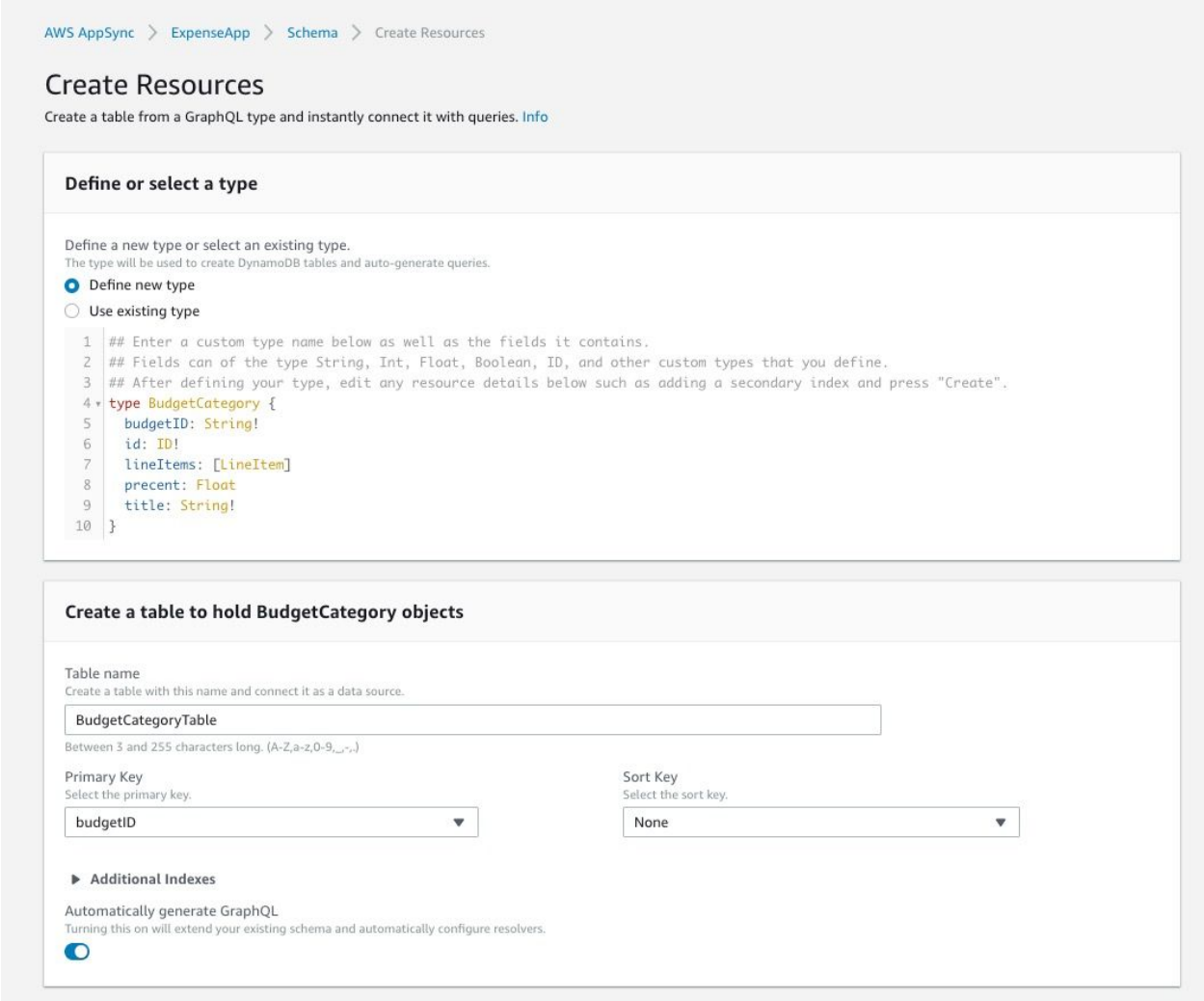
Part 1:

For our project we used Amazon's DynamoDB, which is a nosql database. The process for setting up the tables and queries are a little different than the typical MySQL queries and script files. The query language we used is GraphQL. Our database consists of three different tables: UserBudget, BudgetCategory and LineItem table. Outlined below is the steps we went through to set the tables up.

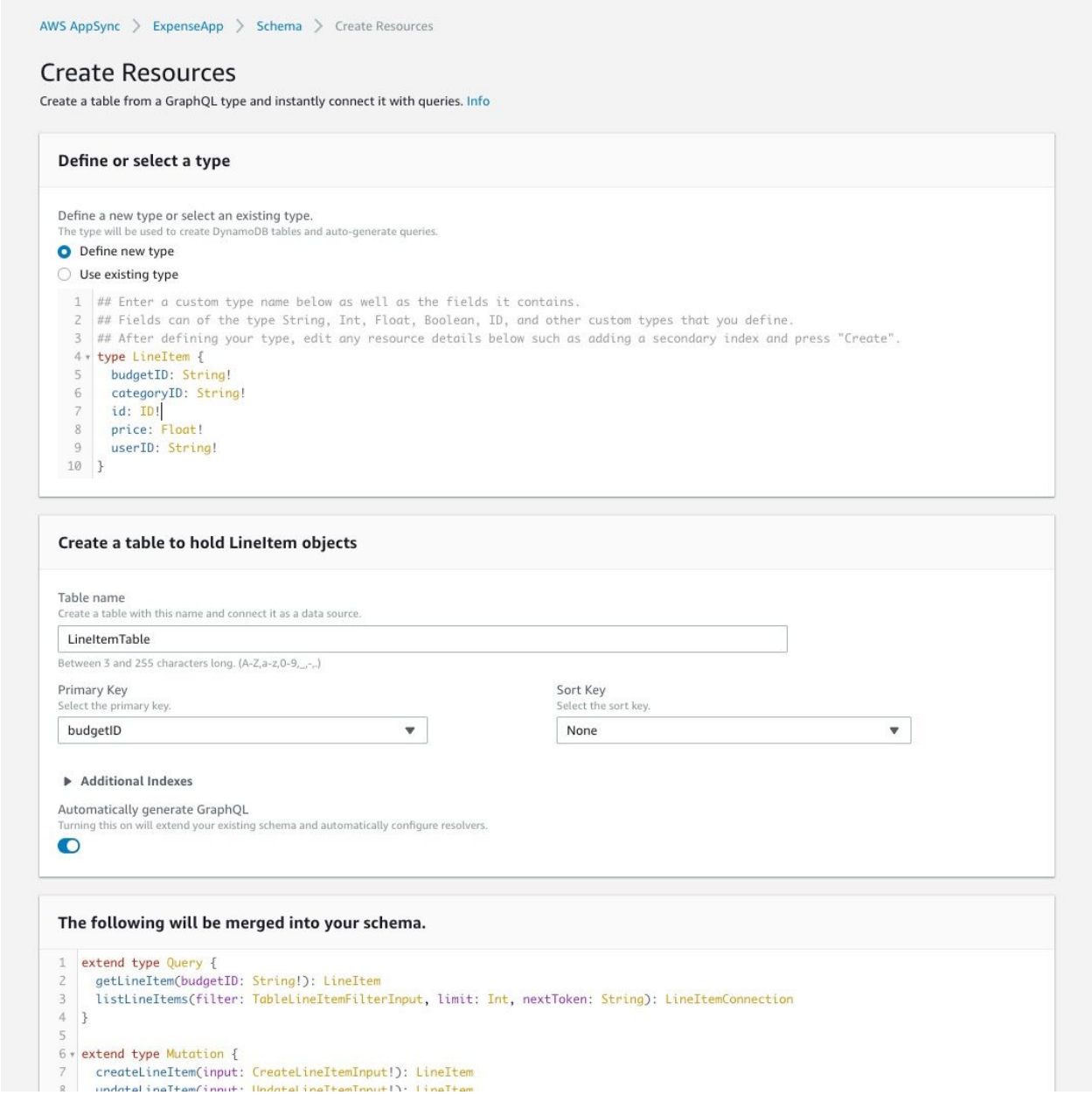
This is AWS AppSync tool. This tool is used to create our apis that create our tables. Through their services the tables can be generated as seen in the screenshot below. We set the primary key, add the columns to the tables and then AWS automatically generates the GraphQL. The first table, UserBudget, has three different columns, id, userid and categories.



The next table we generated was BudgetCategory. This table stores all the categories with the associated percent for all the users and is related to the UserBudget table through budgetID.



The final table we generated was LineItem. This table stores all of the expenses entered by the users. Each “Lineitem” (expense) is associated with a categoryID. It also contains the price and we also added a title section for each expense.



Once all of the tables were set up and the queries were generated we were able to connect our app to AWS AppSync allowing our app to communicate with Amazon's Web Services. We used Amazons authenticator, which stores all of our users in AWS Cognito, allowing us to keep track of all of the users. Below is a query we use to get all of the information for a certain user. The query takes the userID and fetches the user's budget, as well as all of the categories associated with that budget, and every line item (expense) associated with each category. With one query, we are able to pull all of the necessary information for that user and use that data as we need. This is why GraphQL is such an efficient language, we set up resolvers on AWS, which basically tells the machine what table to look in and how to find the data associated with IDs.

```
1
2   // GET USER BUDGET BY USER ID
3   // query {
4   //     listUserBudgets(filter: {userID: {eq: "Bridger"}}) {
5   //       items {
6   //         id
7   //         userID
8   //         categories {
9   //           title
10  //           precent
11  //           lineItems {
12  //             id
13  //             price
14  //           }
15  //         }
16  //       }
17  //     }
18  //   }
```

This query is used when a user creates a new category for their budget. This query takes the budgetID, the title of the category, and the percent and stores that information in the BudgetCategory table.

```
29  // CREATE A CATEGORY FOR BUDGET
30  // mutation ($budgetId: String!) {
31  //     createBudgetCategory(input: {budgetID: $budgetId, title: "Food", precent: 0.33}) {
32  //       id
33  //       budgetID
34  //       title
35  //       precent
36  //     }
37  //   }
38
```

Part 2:

Below is our ER diagram for our database. Starting with the UserPool table, AWS cognito stores our users and we take the clientid from cognito and store it as our userID. Users can have a budget and each budget (UserBudget) can have many categories (BudgetCategory) and each category (BudgetCategory) can have many expenses (LineItems), but each expense cannot have more than one category and each category cannot have more than one budget, and each budget cannot have more than one user.