

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.

AWS AppSync > ExpenseApp > Schema > Create Resources

Create Resources

Create a table from a GraphQL type and instantly connect it with queries. [Info](#)

Define or select a type

Define a new type or select an existing type.
The type will be used to create DynamoDB tables and auto-generate queries.

☒ Define new type
☐ Use existing type

```
1 ## Enter a custom type name below as well as the fields it contains.
2 ## Fields can of the type String, Int, Float, Boolean, ID, and other custom types that you define.
3 ## After defining your type, edit any resource details below such as adding a secondary index and press "Create".
4 type UserBudget {
5   categories: [BudgetCategory]
6   id: ID!
7   userID: String!
8 }
```

Create a table to hold UserBudget objects

Table name
Create a table with this name and connect it as a data source.

UserBudgetTable

Between 3 and 255 characters long. (A-Z,a-z,0-9,_,-)

Primary Key
Select the primary key.

id

Sort Key
Select the sort key.

None

► Additional Indexes

Automatically generate GraphQL
Turning this on will extend your existing schema and automatically configure resolvers.

☒

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.

[AWS AppSync](#) > [ExpenseApp](#) > [Schema](#) > Create Resources

Create Resources

Create a table from a GraphQL type and instantly connect it with queries. [Info](#)

Define or select a type

Define a new type or select an existing type.
The type will be used to create DynamoDB tables and auto-generate queries.

☒ Define new type

☐ Use existing type

```
1  ## Enter a custom type name below as well as the fields it contains.
2  ## Fields can be of the type String, Int, Float, Boolean, ID, and other custom types that you define.
3  ## After defining your type, edit any resource details below such as adding a secondary index and press "Create".
4  type BudgetCategory {
5    budgetID: String!
6    id: ID!
7    lineItems: [LineItem]
8    percent: Float
9    title: String!
10 }
```

Create a table to hold BudgetCategory objects

Table name
Create a table with this name and connect it as a data source.

BudgetCategoryTable

Between 3 and 255 characters long. (A-Z,a-z,0-9,-,_)

Primary Key
Select the primary key.

budgetID

Sort Key
Select the sort key.

None

► **Additional Indexes**

Automatically generate GraphQL
Turning this on will extend your existing schema and automatically configure resolvers.

☒

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.

[AWS AppSync](#) > [ExpenseApp](#) > [Schema](#) > Create Resources

Create Resources

Create a table from a GraphQL type and instantly connect it with queries. [Info](#)

Define or select a type

Define a new type or select an existing type.
The type will be used to create DynamoDB tables and auto-generate queries.

☒ Define new type

☐ Use existing type

```
1  ## Enter a custom type name below as well as the fields it contains.
2  ## Fields can of the type String, Int, Float, Boolean, ID, and other custom types that you define.
3  ## After defining your type, edit any resource details below such as adding a secondary index and press "Create".
4  type LineItem {
5    budgetID: String!
6    categoryID: String!
7    id: ID!
8    price: Float!
9    userID: String!
10 }
```

Create a table to hold LineItem objects

Table name
Create a table with this name and connect it as a data source.

Between 3 and 255 characters long. (A-Z,a-z,0-9,_,-,.)

Primary Key
Select the primary key.

Sort Key
Select the sort key.

Additional Indexes

Automatically generate GraphQL
Turning this on will extend your existing schema and automatically configure resolvers.

☒

The following will be merged into your schema.

```
1  extend type Query {
2    getLineItem(budgetID: String!): LineItem
3    listLineItems(filter: TableLineItemFilterInput, limit: Int, nextToken: String): LineItemConnection
4  }
5
6  extend type Mutation {
7    createLineItem(input: CreateLineItemInput!): LineItem
8    updateLineItem(input: UpdateLineItemInput!): LineItem
```

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 Amazon's 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
39
```

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.

