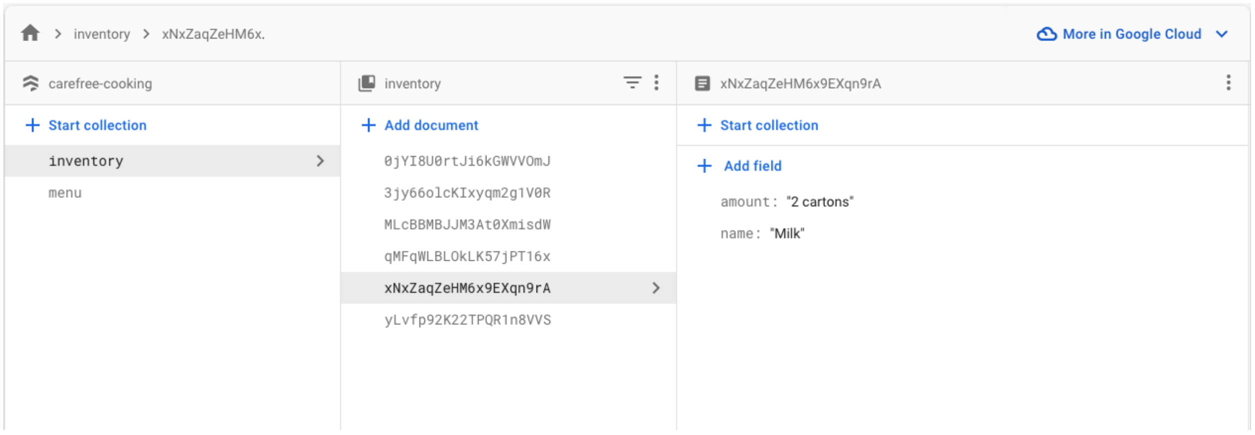
Criterion C: Development

Index of Techniques

- Editing and Adding Firestore Data
- Searching and Displaying Menu/Inventory Data
- Designing

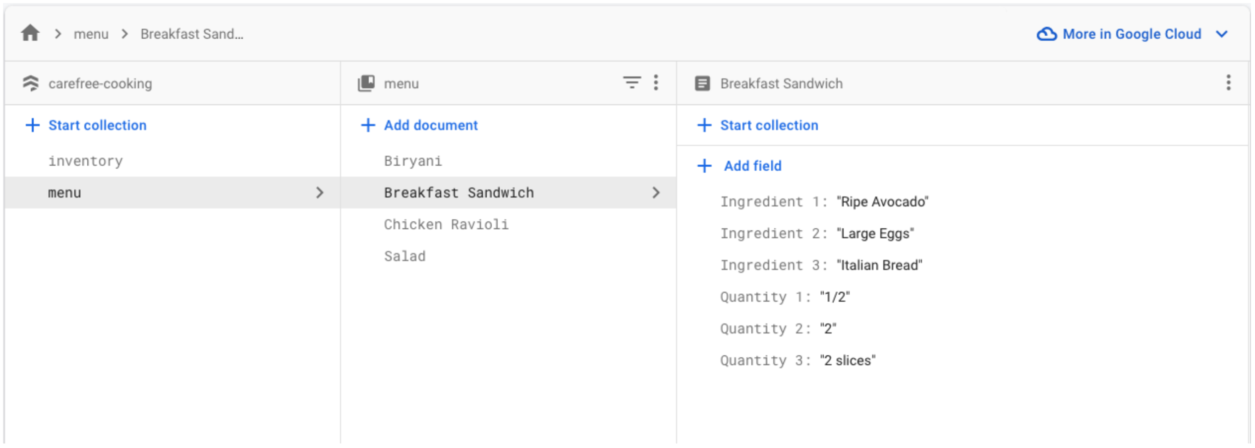
Firestore Data

I utilized the Firestore Cloud Database in Firebase as the backend to store the data for the Menu and Inventory. The different collections/documents were me using **objects as data records.** I also used a **file i/o**. My program reads data from the database through an input stream, and writes data that goes back to the database through an output stream.



```
amount: "2 cartons"
name: "Milk"
```

Figure 1: Sample Inventory Document

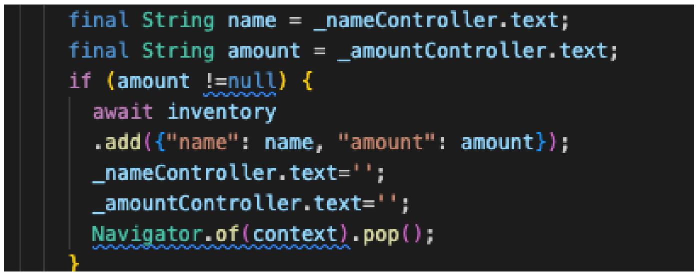


```
Ingredient 1: "Ripe Avocado"
Ingredient 2: "Large Eggs"
Ingredient 3: "Italian Bread"
Quantity 1: "1/2"
Quantity 2: "2"
Quantity 3: "2 slices"
```

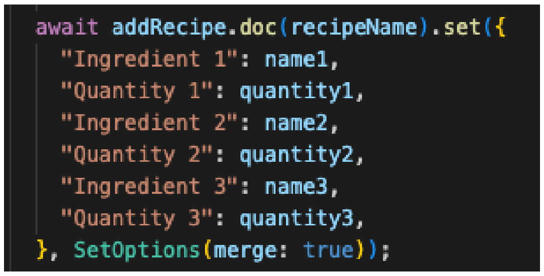
Figure 2: Sample Recipe Document in Menu

Adding Firestore Data

To add values to the inventory or menu collection in the Firestore database, I used TextFields with TextControllers. These accepted Strings, even for the quantity field, so the user could input their own units instead of a plain number (ex. 3 cups). The entered values held in the contents of the Text Controllers are then added to the Firestore database. I used a simple **if statement** to make sure values are only sent to the database if they are not empty. This addition was prompted once the button ‘Add’ was clicked in the inventory page, or when the plus icon was clicked in the addRecipe page. The user creates these instances in the database, meaning these instances are **user-defined objects.**



```
final String name = _nameController.text;
final String amount = _amountController.text;
if (amount !=null) {
  await inventory
  .add({"name": name, "amount": amount});
  _nameController.text='';
  _amountController.text='';
  Navigator.of(context).pop();
}
```



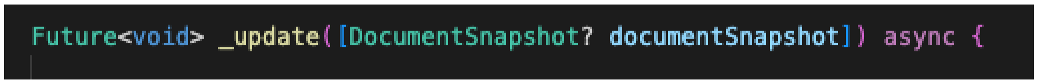
```
await addRecipe.doc(recipeName).set({
  "Ingredient 1": name1,
  "Quantity 1": quantity1,
  "Ingredient 2": name2,
  "Quantity 2": quantity2,
  "Ingredient 3": name3,
  "Quantity 3": quantity3,
}, SetOptions(merge: true));
```

Figure 3: Adding inventory items

Figure 4: Adding Menu items

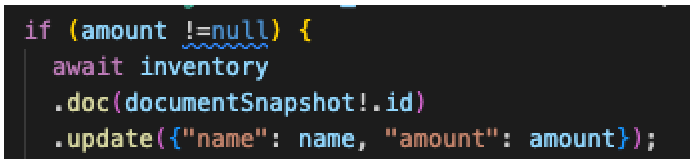
Updating Firestore Data

To update existing inventory items, once the edit Icon button was pressed, the method _update was called. _update is a **user-defined method with parameters**, as shown in Figure 5, where the documentSnapshot is passed through the method. Using TextFields, I grabbed the user’s new values and used .update to change the value in the database. The same **if statement** I used in the addition was used here.



```
Future<void> _update([DocumentSnapshot? documentSnapshot]) async {
```

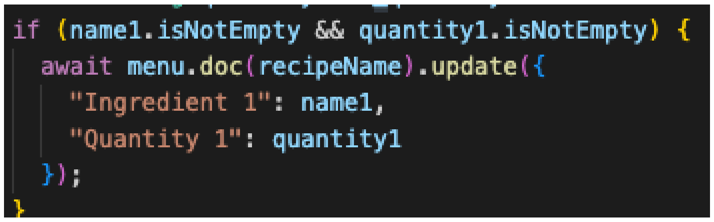
Figure 5: _update() method



```
if (amount !=null) {
  await inventory
  .doc(documentSnapshot!.id)
  .update({"name": name, "amount": amount});
```

Figure 6: _update() method updating FireStore

Similarly, in the Menu page, users can edit the recipes they made in the addRecipe page. Here, a **complex selection if statement** is used to make sure both fields are not empty before updating. To make displaying the Recipes easier in the menu page, I used the recipe Name as the document id instead of an auto-generated one, which would also aid in searching.

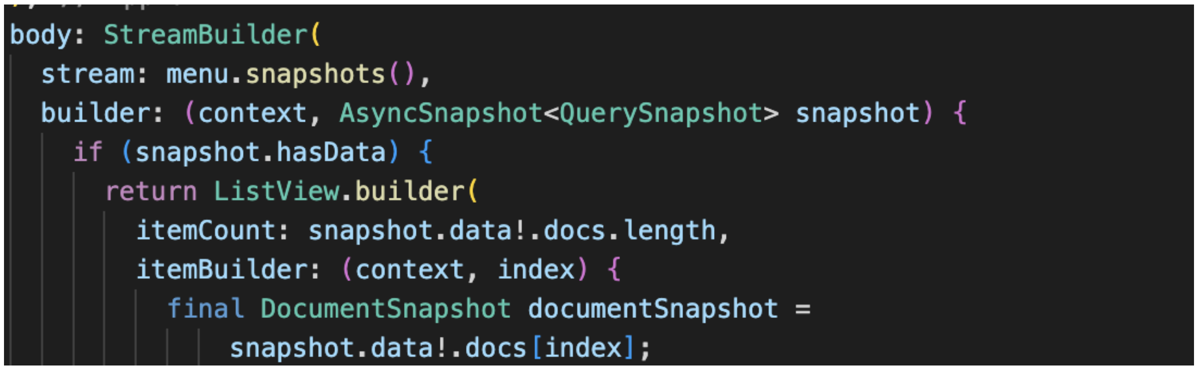


```
if (name1.isNotEmpty && quantity1.isNotEmpty) {
  await menu.doc(recipeName).update({
    "Ingredient 1": name1,
    "Quantity 1": quantity1
  });
}
```

Figure 7: if statement checking if empty and updating

Searching and Displaying Menu/Inventory Data

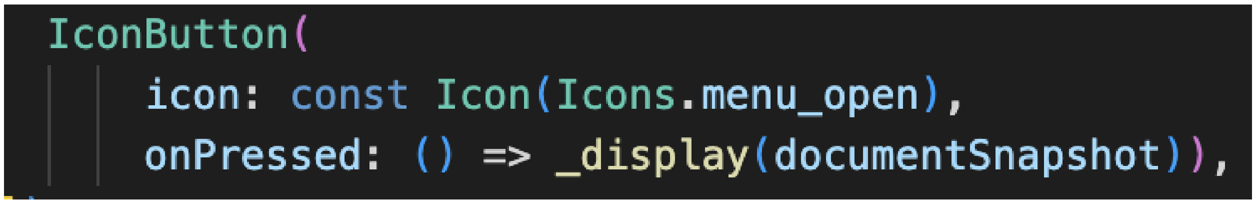
To **search** for the values in the database, I utilized **StreamBuilder** to **loop** through my stream, or each of the snapshots in the firestore collections for ‘inventory’ and ‘menu’, as shown in Figure 8. Through this search, I was able to find the data values that I wanted to display from the database.



```
body: StreamBuilder(
  stream: menu.snapshots(),
  builder: (context, AsyncSnapshot<QuerySnapshot> snapshot) {
    if (snapshot.hasData) {
      return ListView.builder(
        itemCount: snapshot.data!.docs.length,
        itemBuilder: (context, index) {
          final DocumentSnapshot documentSnapshot =
              snapshot.data!.docs[index];
```

Figure 8: StreamBuilder for Menu

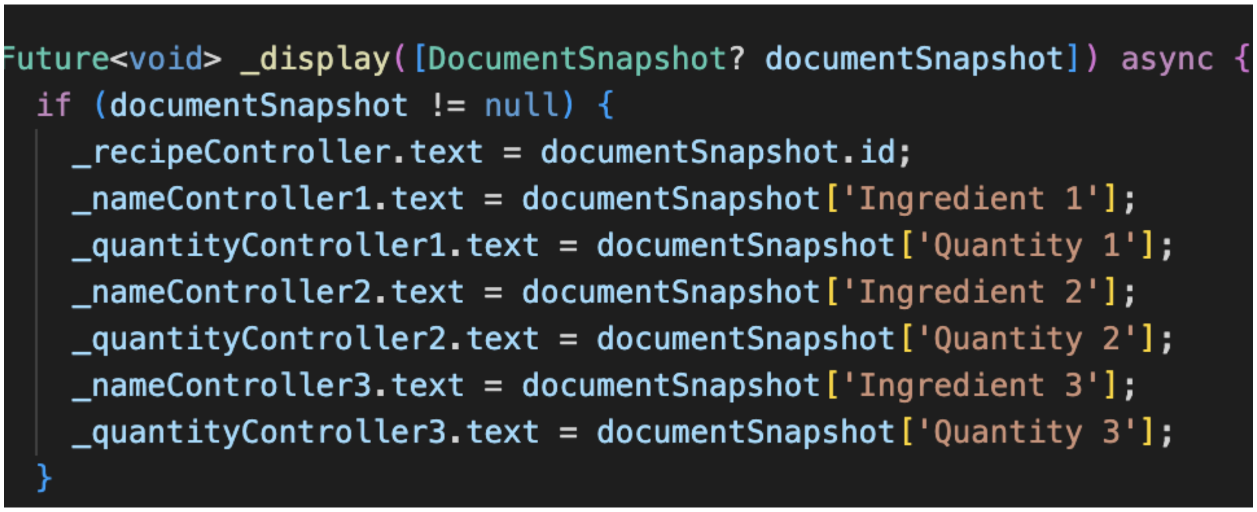
I called the _display method at the press of the icon button. The display method was a **user-defined method with parameters**.



```
IconButton(
    icon: const Icon(Icons.menu_open),
    onPressed: () => _display(documentSnapshot)),
```

Figure 9: Icon button calling _display() method

Using the parameter documentSnapshot, the _display() method sets the TextFields to the existing values (the ingredients and quantity) from the documentSnapshot (the specified recipe). The **method returned** these text fields filled in with the recipe the user had requested to see.

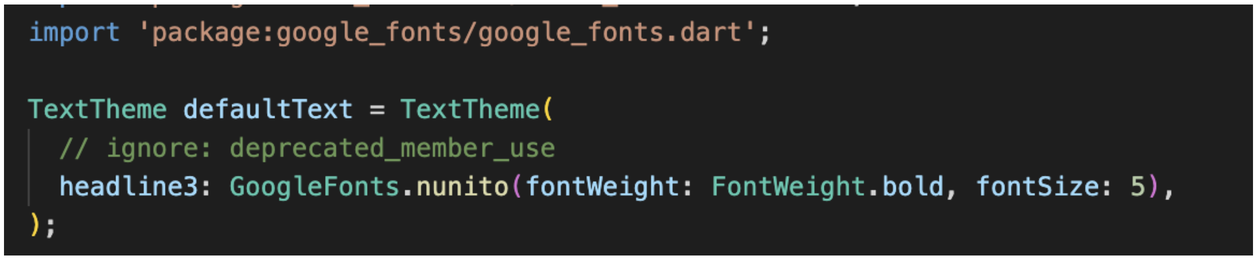


```
Future<void> _display([DocumentSnapshot? documentSnapshot]) async {
  if (documentSnapshot != null) {
    _recipeController.text = documentSnapshot.id;
    _nameController1.text = documentSnapshot['Ingredient 1'];
    _quantityController1.text = documentSnapshot['Quantity 1'];
    _nameController2.text = documentSnapshot['Ingredient 2'];
    _quantityController2.text = documentSnapshot['Quantity 2'];
    _nameController3.text = documentSnapshot['Ingredient 3'];
    _quantityController3.text = documentSnapshot['Quantity 3'];
  }
```

Figure 10: _display() method

Designing

To choose the best fonts that complimented the design of the app, I imported an **additional library**, google_fonts. With this, I could call the TextThemes I wanted.



```
import 'package:google_fonts/google_fonts.dart';

TextTheme defaultText = TextTheme(
  // ignore: deprecated_member_use
  headline3: GoogleFonts.nunito(fontWeight: FontWeight.bold, fontSize: 5),
);
```

Figure 11: google_fonts package