Criterion C

Introduction

I used Flutter and Dart in Visual Studio Code to develop the frontend of my Android and iOS app, and used Firebase as the backend for authentication, database storage. I also used Javascript for coding Cloud Functions for aggregation queries on Cloud Firestore (Firebase database).

Summary List

Examples of techniques that were used in this program are included but not limited to:

- Main Algorithms
  - Login and Logout (Use of Firebase Auth package)
  - Checkout Calculations
  - Graphs showing Expenditure and Income (Use of flutter_charts package)
  - Creating/Updating/Deleting Products from Stock List, Notices (Use of Cloud Firestore package)
  - Creating/Updating/Deleting Transactions + Aggregation Queries (Use of Cloud Firestore package + Cloud Functions)
  - Checking user permissions for CRUD operations and displaying UI items
- Try and Catch Exception Handling
- Inheritance / Polymorphism/Encapsulation of Methods and Variables
- Asynchronous Programming
- Maps and Arrays
- Simple Compound Selection (if/else)
- For Loops
- Validation Checks
- Use of Sub classes and Super classes

Backend - Firestore Integration

CRUD Methods (Cloud Firestore)

As the app is heavily connected to the cloud database and accesses it in multiple places, it was more efficient to create a **super class encapsulating** the **methods** used for adding, reading, updating and deleting data on the client device.

```dart
import 'dart:async';
import 'package:cloud_firestore/cloud_firestore.dart';

class CrudMethods {
  var path; //Stores path in for file database
  var data; //Stores data to be added/deleted/updated
  CrudMethods(this.path, this.data);

//Asynchronous method to add new data
  Future<void> addData(data) async {
    Firestore.instance.collection(path)
      .add(data)
      .catchError(
        (e) {print(e); },
      )
  }
//Asynchronous method to get current data
  getData() async {
    return Firestore.instance.collection(path).snapshots()
;
  }
//Asynchronous method to update current data with new data
  Future<void> updateData(selectedDoc, newValues) async {
    Firestore.instance.collection(path).document
(selectedDoc)
        .updateData(newValues)
        .catchError(
      (e) {
        print(e);
      },
    );
  }
//Method to delete data from database
  deleteData(docId) {
    Firestore.instance.collection(path).document(docId)
        .delete()
        .catchError((e) {
      print(e);
    });
  }
}
```

Fig.1 **CRUD Methods**

The use of **asynchronous methods** was important to speed up the development process and to process simple and independent data as it allows for multiple things to happen at the same time. As these methods

call from the database, less time will be spent on waiting for data to be retrieved and thus the application will run faster. **Asynchronous methods** also allowed for the user to see action if the data is being loaded using a Progress Indicator. The call of an **async function** returns a **Future,** the cue to getting the result required. The **await** expression suspends the currently running function which uses the **Future** to wait until the **Future** is completed.

The use of **encapsulation** and **inheritance** reduces the amount of code as well as making maintenance easier through the ability to make independent code changes.

Cloud Functions (Aggregation Queries)

As there was no native support for aggregation queries in Cloud Firestore, I used Google Cloud Functions, a platform for building and connecting cloud services, to handle aggregation queries used for managing finances. I chose this option as it would require less client-side processing and data to load faster over the option of performing aggregation queries on device. I used **Javascript** as the language to write the function. An example of the function is given in Fig 2.

This function and others to manage deletion, updating in different categories was used to update fields for Total Inflow and Outflow of each category.

```
const functions = require('firebase-functions');
const admin = require('firebase-admin');
admin.initializeApp();
//aggregation function for new Outflow
exports.categoryOutflowsTotal = functions.firestore
    .document(
'transactionsCategories/{catId}/outflows/{outflowId}')
//Check for write type, (Create, Delete or Update)
    .onCreate((snap, context) => {
        // Get value of the newly added transaction
        var transVal = snap.data().amount;

        // Get a reference to the category
        var catRef = admin.firestore().collection(
'transactionsCategories').doc(context.params.catId);

        // Update aggregations in a transaction
        return admin.firestore().runTransaction(
transaction => {
            return transaction.get(catRef).then(catDoc
=> {
                var currentTotal = (catDoc.data() &&
catDoc.data().totalOutflows) || 0;

                var newTotal = transVal + currentTotal;

                return transaction.update(catRef, {
                    totalOutflows: newTotal
                });
            });
        })
            .catch(err => console.log(err))
    });
```

Fig.2 **Cloud Function**

The benefit of **server-side code** is being able to reduce calculation times and increase performance for the user as well as making the database more secure by not needing giving users permissions to update the aggregate data within the database. The chances of failure due to connectivity issues are also lower.

Cloud Firestore Database Rules

In order to ensure that data was stored safely and build the role-based access system required by the client, I used **Firebase Authentication** in conjunction with **Cloud Firestore Security Rules.** These rules (Figure 3) defined who could and could not access the database, providing a second layer of authentication for the user to write to the database.

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /transactionsCategories/{transactionsCategory} {
      allow read: if true;
      allow write: if exists(/databases/$(database)/
documents/admins/$(request.auth.uid))

      match /inflows/{inflow}{
      allow read: if true;
      allow create: if request.auth.uid != null;
      allow delete, update: if exists(/databases/$(
database)/documents/admins/$(request.auth.uid))
      }
    }
  }
}
```

Fig.3 **Cloud Firestore Security Rules**

The rules define access rules for each collection within the database(Figure 4), with this extract showing how all users are allowed to read the data in the main "transactionsCategories" and "inflows" sub categories, but in order for write operations, the user id must be in the list of "admins" with the exception for "inflows" allowing any signed in user to create data, this was

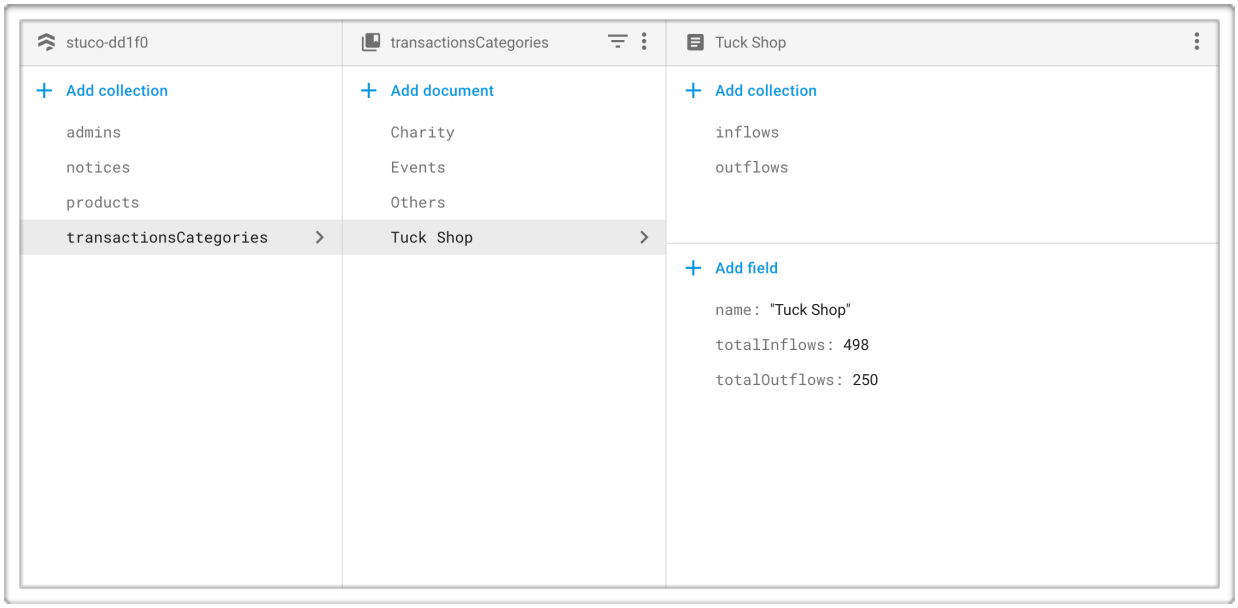

Fig.4 **Cloud Firestore Database Structure**

necessary to allow for the Checkout procedure to be carried out by all users.

Frontend - Flutter Architecture

Login Procedure

I used a Form and Two TextFormField Widgets to save the email and password inside **private variables**. The input is also **validated** with a series of **if statements** to makes sure that it meets requirements.

```
TextFormField(
  validator: (input) {
    if(input.isEmpty){
      return 'Please enter password';
    } //Checks for empty password
    if(input.length < 9){
      return 'Please enter 9 digit Student ID';
    }
//Checks for length of password to ensure high level of security
  },
  autocorrect: false,
  textCapitalization: TextCapitalization.none,
  maxLines: 1,
  onSaved: (input) => _password = input,
  obscureText: true, //prevents password from being seen
),
```

Fig.5 **Password Text Form Field**

The form state is then saved and sent to Firebase Auth for verification (Figure 6), I used the the firebase_auth **library** for this. This was beneficial as it reduced the amount of work. A **try and catch** was used to get any potential errors in the procedure.

```
void signIn() async {
  if (_formKey.currentState.validate()) {
    _formKey.currentState.save();
    //Check Validation
    try {
      FirebaseUser user = await FirebaseAuth.instance
        .signInWithEmailAndPassword(
          email: _email, password: _password
        );
      //If user is authenticated, the user is pushed to the home screen
      Navigator.push(context,
        MaterialPageRoute(
          builder: (context) => WillPopScope(
            onWillPop: () async {
              return false;
            },
            child: (MyHome(user: user)),
          ),
        ),
      );
    } catch (e) {
      print(e.message);

//The setState() is activated in console and the screen reloaded to display error
        _authHint = 'Sign In Error\n\n${e.toString()}';
      });
    }
  }
}
```

Fig.6 **Sign-in Method**

Displaying Data from Database

In order to display the list of products in the Stock Screen and the Checkout Screen as well as the list of Transactions in the finances screens and the Notices in the Notice screen, I used a StreamBuilder in conjunction with two other widgets, a ListView for most screens and a GridView for the Checkout Screen.

A StreamBuilder is a reactive widget that can build widgets from a stream of user defined objects(a sequence of **asynchronous** events). The ListView and GridView are just two widgets with the same purpose to build widgets from a list in different styles. Using these widgets in conjunction with the CRUD methods mentioned above, allows for the display of data (Figure 7).

```dart
import 'package:flutter/material.dart';

Widget _noticeList() {
  if (notices != null) {
    return StreamBuilder(
        stream: notices, //Gets stream from getData() method called earlier
        builder: (context, snapshot) {
          if (snapshot.hasError)
            return new Center(child: Text('${snapshot.error}'));
            //Returns error if shown
          switch (snapshot.connectionState) {
            case ConnectionState.waiting:
              return new Center(child: CircularProgressIndicator());
              //Shows loading screen if data not obtained yet
            default:
              return ListView.separated(
                  separatorBuilder: (context, index) => Divider(),
                  itemCount: snapshot.data.documents.length,//Gets amount of items in list
                  itemBuilder: (context, i) {
                    //Returns list of ListTiles that have data about notices
                    return new ListTile(
                        subtitle:
                            Text(snapshot.data.documents[i]['time'].toString()),
                            //Gets time information from object
                        onLongPress: () {
                          showDialog();//Shows Dialog for possible actions
                        },
                        title: Row(
                          children: <Widget>[
                            Expanded(
                              child: Text(snapshot.data.documents[i]['title']),
                              //Gets notice title and displays
                            ),
                          ],
                        ));
                  });
          }
        });
  } else {
    return Text('Loading, Please wait..');
  }
}
```

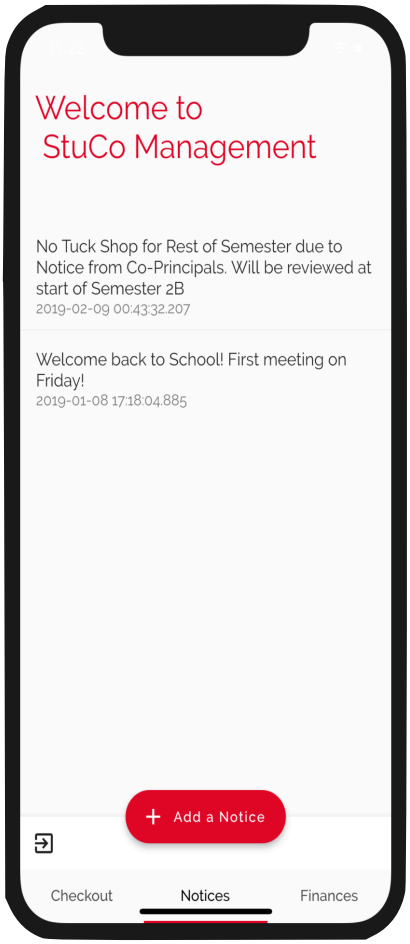

Fig.7 **StreamBuilder and ListView with UI displayed**

Checkout Procedure

The Checkout procedure (Figure 8) made use of three different **classes** named Product, CheckoutItem and CheckoutBasket. The Product **class** communicates with the database, getting each product and its attached information, while also holding a method for removing stock for each product when a checkout transaction is carried out. The CheckoutItem **class** holds the information of a Product **object** together with the quantity of that Product selected (together with methods for incrementing the quantity) and calculating the total of price of a selected quantity of one Product **object**. The CheckoutBasket class holds a List of CheckoutItem objects that are instantiated when the product in the GridView is tapped on. The CheckoutBasket class holds methods for updating the overall stock level for all products, adds the Checkout Transaction to the database as well as getting the total cost of all CheckoutItems in the CheckoutBasket. The CheckoutBasket class is dependent on the CheckoutItems class which is dependent on the Product class.

```dart
class Product {
  Product(); //Instantiates
  DocumentReference _ref;
  String name;
  double price;
  int stockLevel;

  factory Product.fromDoc(DocumentSnapshot doc) {...
  }//Gets Product Data from Database

  Future<void> removeStock(int count) async {...
  }//Runs transaction to update stock level after checkout
}

class CheckoutItem {
  CheckoutItem(this.product) : _quantity = 1;

  final Product product;
  int _quantity;
  int get quantity => _quantity; //Accessor method(Quantity of specified product)
  double get total => _quantity * product.price;
//Accessor method(Total Price of specified product)

  void incrementQuantity() {
    _quantity++;
  }
}

class CheckoutBasket {
  final items = Set<CheckoutItem>();//List of Checkout Items
  DateTime now = DateTime.now();

  void addProduct(Product product) {
    final newItem = CheckoutItem(product); //Instantiates new CheckoutItem
    final currentItem = items.lookup(newItem);//Searches if Item is in list
    if (currentItem != null) {
      currentItem.incrementQuantity();
    } else {
      items.add(newItem);
    }
  }
  double get total => items.fold(0.0, (prev, item) => prev + item.total);//Gets total price
  int get itemCount => items.length;//Gets total items in basket

  CheckoutItem operator [](int index) => items.elementAt(index);

  void updateStockLevel() {
    for (CheckoutItem item in items) {
      item.product.removeStock(item.quantity);//Uses removeStock method from Product
    }
  }
  Future<void> addTransaction(double basketTotal) async {...
  } //Logs Transaction
}
```

Fig.8 **Checkout Procedure**

Graphing Finances

In order to graph inflow and outflow data, I used charts_flutter library together with Firestore to get the data from the database. I embedded the chart as a widget (getChartWidget) within a StreamBuilder.

```
class Category {
  Category();
  String name;
  double totalInflows;
  double totalOutflows;
  factory Category.fromDoc(DocumentSnapshot doc) {...
  }//Gets Category total inflows and outflows from Cloud Firestore
}

Widget getChartWidget(List<Category> category,List data,measure,title) {
  var series = [
      new charts.Series<Category, String>(
        domainFn: (Category category, _) => category.name,//Seperation of categories
        measureFn: measure,
        id: 'Values',
        data: data,
        labelAccessorFn: (Category category, _) => '${category.name}',//Gets label name
      ),
    ];
    return charts.PieChart(
      series,
      animate: true,
    ),
}
```

Fig.9 **Graphing Finances**

(863 Words)