CRITERION C: DEVELOPMENT

TECHNIQUES:

The key techniques used in my application are the following, while there are many more:

1. Cloud Firestore
2. Custom Abstract Data Type: Map
3. Object Oriented Programming Fundamentals
4. Asynchronous Methods
5. Navigation Through Lists
6. StreamBuilder (receiving and parsing data)

EXPLANATION:

Implementation of Cloud Firestore (database structure)

My client requested that my approach to data storage allows for access from multiple different devices tailored towards the iOS platform. This is where the use of **Cloud Firestore (external library)** became highly relevant. Below are screenshots simulated using both an iPad Air Generation 5 (simulated, but same device client will be using), and the iPhone 13 pro, another device the client will be using. **(Appendix B.1)** Firestore allowed me to create a data system where user data was personalized and expandable. Pictured on the right is the data storage structure used for a single testing account. This data structure allows for user specific data to be accessed without compromising integrity. Furthermore, no matter the device, my client was able to access the data thanks to the implementation of a cloud storage solution.

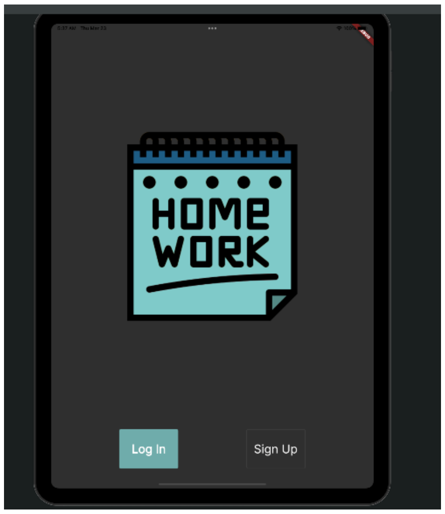


Figure 1: View from iPad Air

This data structure is highly scalable as users are added to the app, they are able to have their own data set up and sent to the cloud. Furthermore, this creates room for expandability, as it is a cloud-based system where the structure can be modified quickly directly through the console or the client itself. I decided to develop my app, using the Firestore system, due to its simplicity alongside the easy scalability as mentioned earlier. **This data structure is also hierarchical, which ensures data is personalized and each user's data is kept separate.**

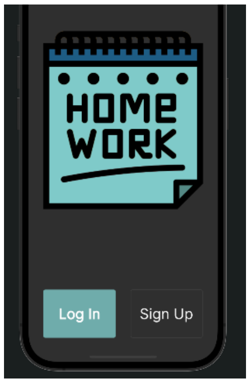
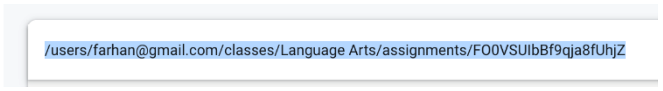


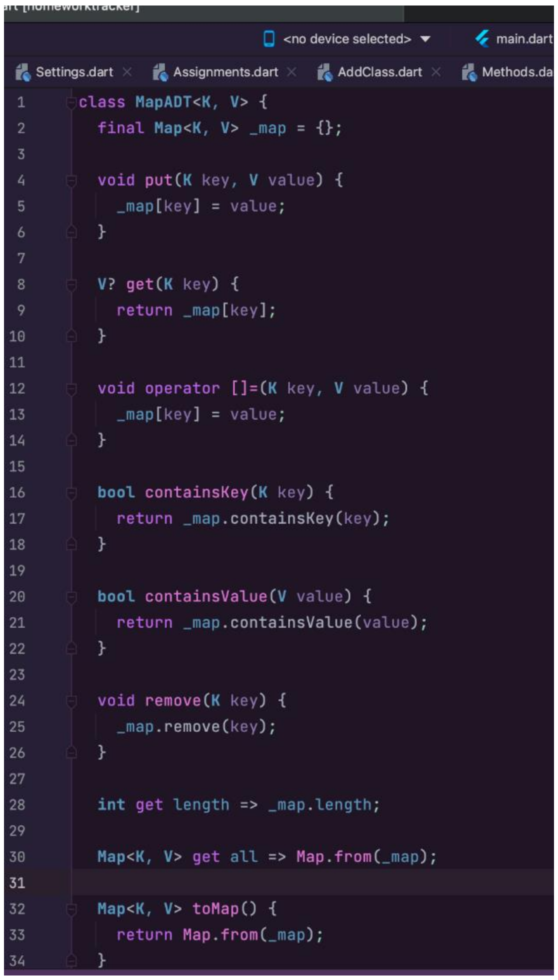
Figure 2: View from iPhone



/users/farhan@gmail.com/classes/Language Arts/assignments/FO0VSUlbBf9qja8fUhjZ

Figure 3: Hierarchical Data Structure (Firestore)

Custom ADT Map



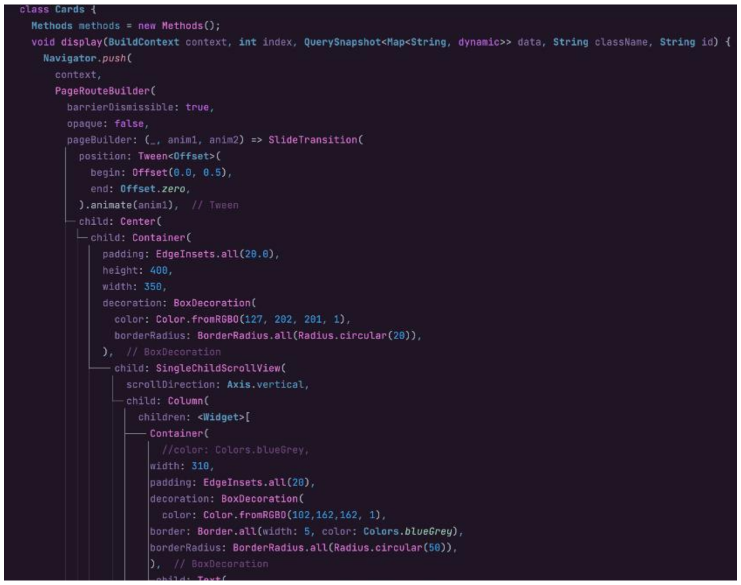
```dart
class MapADT<K, V> {
  final Map<K, V> _map = {};

  void put(K key, V value) {
    _map[key] = value;
  }

  V? get(K key) {
    return _map[key];
  }

  void operator []=(K key, V value) {
    _map[key] = value;
  }

  bool containsKey(K key) {
    return _map.containsKey(key);
  }

  bool containsValue(V value) {
    return _map.containsValue(value);
  }

  void remove(K key) {
    _map.remove(key);
  }

  int get length => _map.length;

  Map<K, V> get all => Map.from(_map);

  Map<K, V> toMap() {
    return Map.from(_map);
  }
```

Figure 4: MapADT.dart Class

Within the app, a **custom Map ADT** is used, conveniently named ADT. While this may seem unnecessary as Dart comes default with a Map data structure. Creating my own Map ADT, allowed me to have full control over the methods and constraints of the data structure, as well as making room for potential expansion. Furthermore, due to the major type safety approach that is implemented within dart, it was important to create a data structure, where I could ensure that the correct types of data were stored in the map and being uploaded to firebase. This structure becomes highly relevant when uploading the data to Firestore, however prior to uploading the data it is converted to a regular map as it is not compatible with Firestore's data structure upload compatibility. The MapADT structure was used within my own code just before upload. This once again allows for expandability as if was requested to modify the data within the MapADT, I would simply need to add a method to the implementation. Furthermore, this an excellent example of **abstraction**, simplifying the code, and allowing for a wide range of flexibility within the application. Below is the MapADT data structure used.

Object Oriented Programming Fundamental Concepts (OOP)

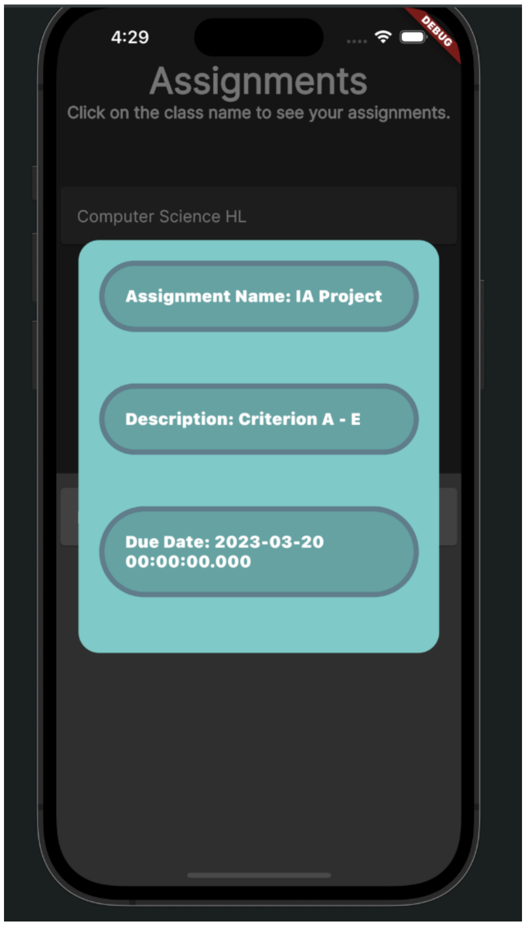
For the application I ensured to use OOP fundamentals such as **Polymorphism, Encapsulation, and Inheritance**, to improve the readability and maintainability of the code. Furthermore, it creates an easily maintainable structure more future development. Using the Flutter library I use inheritance of Widgets, to create the multiple different interfaces within my app, furthermore polymorphism is highlighted with the reusable widgets, presented within the UI Widgets directory within my app. This allows for reusability within my code, and ensuring that the card display can be



```dart
class Cards {
  Methods methods = new Methods();
  void display(BuildContext context, int index, QuerySnapshot<Map<String, dynamic>> data, String className, String id) {
    Navigator.push(
      context,
      PageRouteBuilder(
        barrierDismissible: true,
        opaque: false,
        pageBuilder: (_, anim1, anim2) => SlideTransition(
          position: Tween<Offset>(
            begin: Offset(0.0, 0.5),
            end: Offset.zero,
          ).animate(anim1),  // Tween
          child: Center(
            child: Container(
              padding: EdgeInsets.all(20.0),
              height: 400,
              width: 350,
              decoration: BoxDecoration(
                color: Color.fromRGBO(127, 202, 201, 1),
                borderRadius: BorderRadius.all(Radius.circular(20)),
              ),  // BoxDecoration
              child: SingleChildScrollView(
                scrollDirection: Axis.vertical,
                child: Column(
                  children: <Widget>[
                    Container(
                      //color: Colors.blueGrey,
                      width: 310,
                      padding: EdgeInsets.all(20),
                      decoration: BoxDecoration(
                        color: Color.fromRGBO(102,162,162, 1),
                      border: Border.all(width: 5, color: Colors.blueGrey),
                      borderRadius: BorderRadius.all(Radius.circular(50)),
                      ),  // BoxDecoration
```

Figure 5: cards.dart class

reused throughout the future implementations. The concepts of polymorphism are applied, the cards objects are changed and modified based on the data passed in allowing for different displays.



Assignment Name: IA Project

Description: Criterion A - E

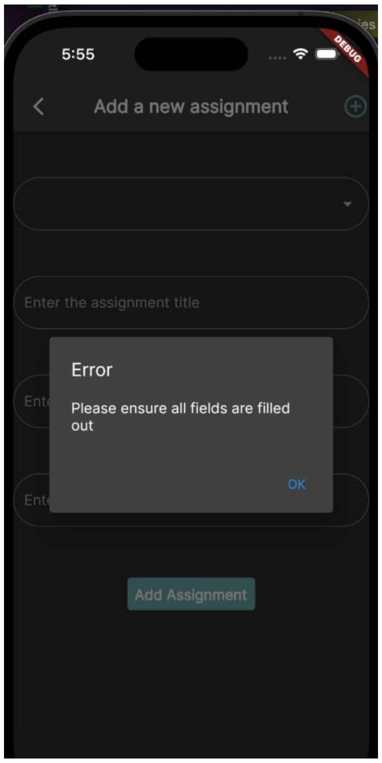
Due Date: 2023-03-20 00:00:00.000

Figure 7: Cards Class UI Screenshot

Asynchronous Methods

The multitude of asynchronous functions used throughout the code, are very important for the upload and addition of data within the application. There are many advantages of using the Future implementation, as it creates an instance where errors can be effectively managed through retrieval calls, and eliminate the chance of crashing because of an unhandled error. This also allows for data to be accessed, **written to, and read, without reading the file into RAM.**

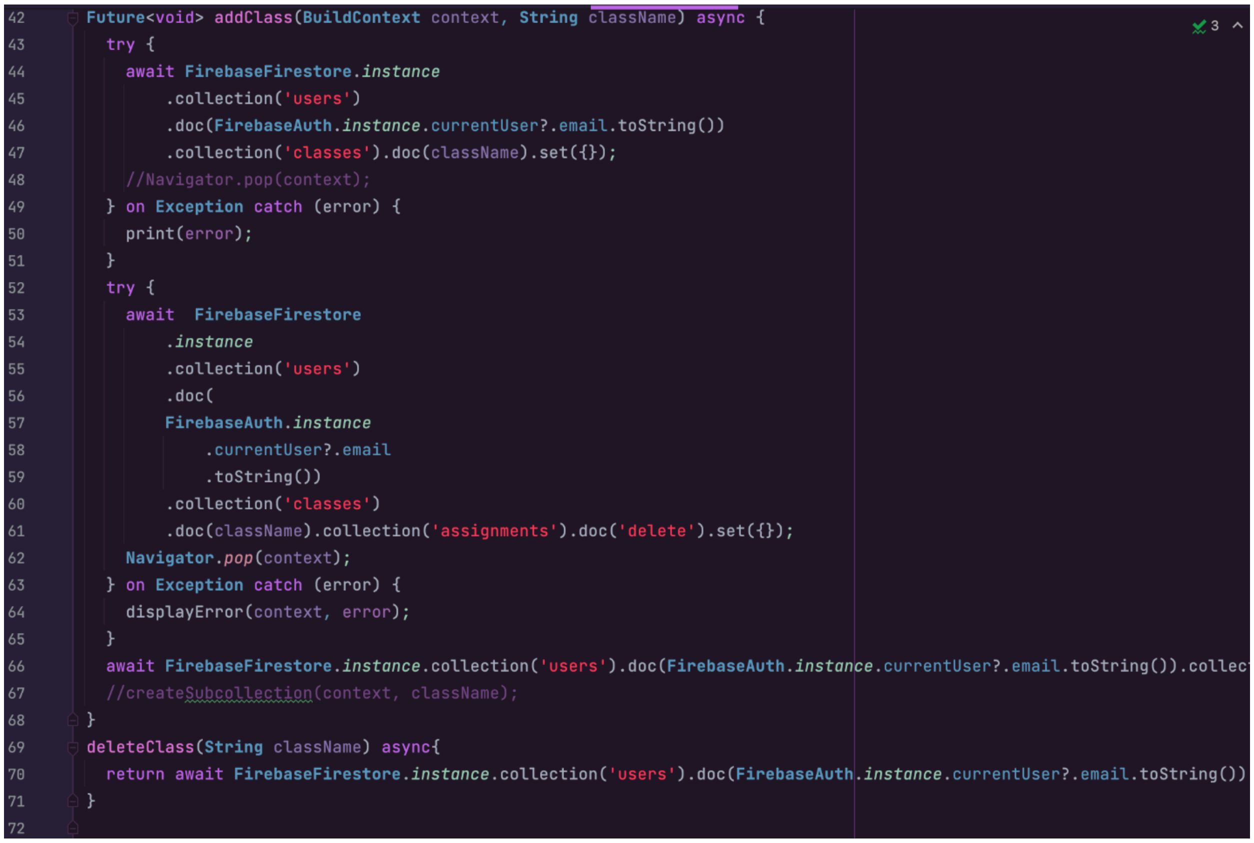
Furthermore, I leverage firebases **error handling where empty fields cannot be submitted, effectively eliminating the possibility that data could cause instabilities and errors throughout the code**.



Error

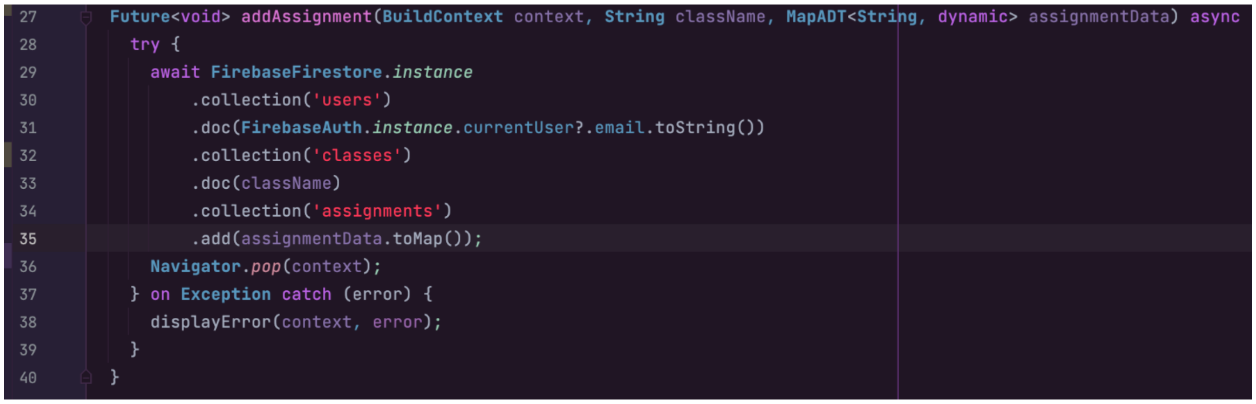
Please ensure all fields are filled out

Figure 6: Error Handling through Firestore



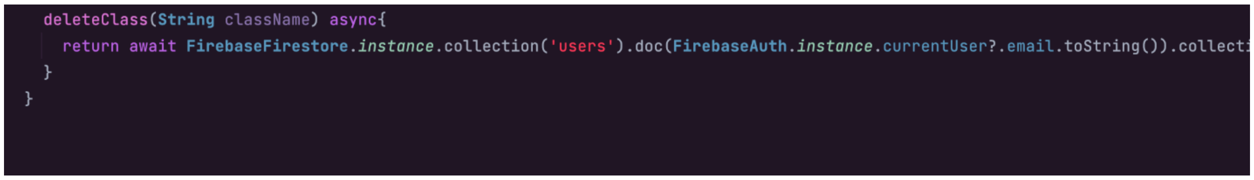
```
Future<void> addClass(BuildContext context, String className) async {
  try {
    await FirebaseFirestore.instance
        .collection('users')
        .doc(FirebaseAuth.instance.currentUser?.email.toString())
        .collection('classes').doc(className).set({});
    //Navigator.pop(context);
  } on Exception catch (error) {
    print(error);
  }
  try {
    await  FirebaseFirestore
        .instance
        .collection('users')
        .doc(
        FirebaseAuth.instance
            .currentUser?.email
            .toString())
        .collection('classes')
        .doc(className).collection('assignments').doc('delete').set({});
    Navigator.pop(context);
  } on Exception catch (error) {
    displayError(context, error);
  }
  await FirebaseFirestore.instance.collection('users').doc(FirebaseAuth.instance.currentUser?.email.toString()).collec
  //createSubcollection(context, className);
}
deleteClass(String className) async{
  return await FirebaseFirestore.instance.collection('users').doc(FirebaseAuth.instance.currentUser?.email.toString())
}
```

Figure 8: addClass async Method



```
Future<void> addAssignment(BuildContext context, String className, MapADT<String, dynamic> assignmentData) async
  try {
    await FirebaseFirestore.instance
        .collection('users')
        .doc(FirebaseAuth.instance.currentUser?.email.toString())
        .collection('classes')
        .doc(className)
        .collection('assignments')
        .add(assignmentData.toMap());
    Navigator.pop(context);
  } on Exception catch (error) {
    displayError(context, error);
  }
}
```

Figure 9: addAssignment async method

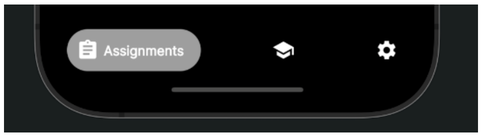


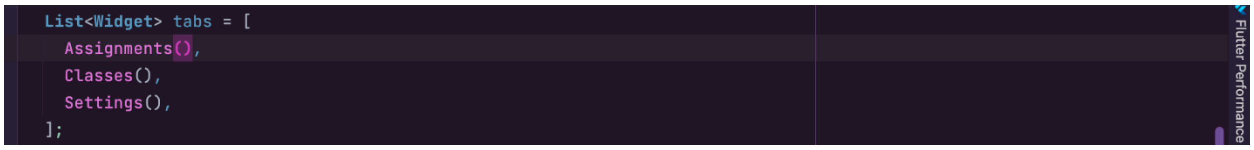
```
  deleteClass(String className) async{
    return await FirebaseFirestore.instance.collection('users').doc(FirebaseAuth.instance.currentUser?.email.toString()).collecti
  }
}
```

Figure 10: deleteClass async method

Navigation through Lists

By utilizing an **ArrayList to store the elements of different Widgets**, users are able to effectively navigate the app. This allows for easy access of the different tabs which then calls on the different widgets to display within the app.



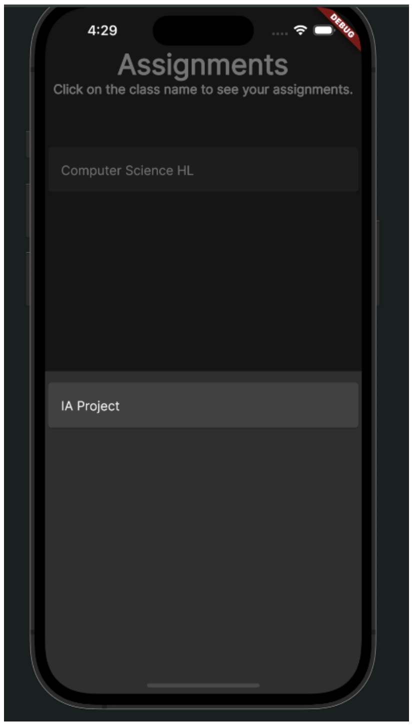


```
List<Widget> tabs = [
  Assignments(),
  Classes(),
  Settings(),
];
```

Figure 11: List Used to navigate to different widgets.

Streambuilder

Utilizing the implementation of a stream builder within the application is necessary to meet the success criteria of being able to view the different assignments and classes within the app. Data from the database, such as **DateTime and Strings are parsed and** fetched effectively by the Stream builder, and the data is loaded into a list view. This is by far the most integral solution used in the application as the success criteria revolves around the users ability to see their uploaded assignments and access the entries. Most importantly, this call is updated multiple times per minute, ensuring that data is up to date and accurate. The streambuilder effectively uses the Firestore libraries document snapshots query to return data and ensure that it is not null, for example if a collection with not documents are produced, within the Firestore database this file would appear italicized, and not be shown within the file. This once again demonstrates how errors are handled, as only relevant data will be shown to the user to ensure that no "ghost entries" are displayed.



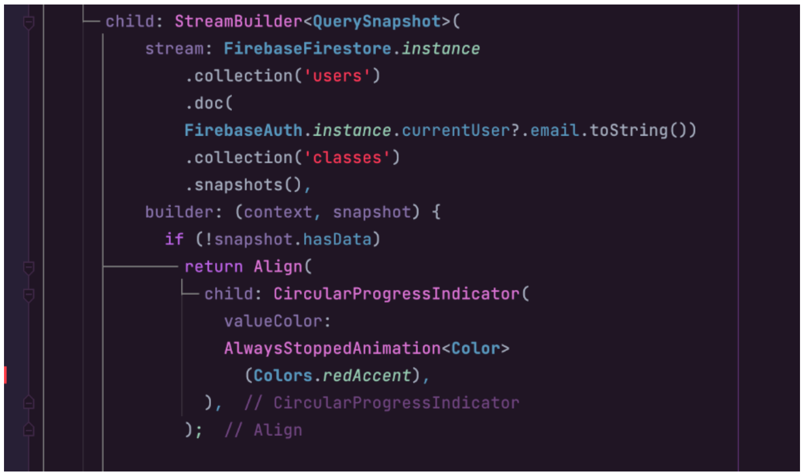
Assignments

Click on the class name to see your assignments.

Computer Science HL

IA Project

Figure 12: Assignments Streambuilder UI Screenshots



```
child: StreamBuilder<QuerySnapshot>(
  stream: FirebaseFirestore.instance
      .collection('users')
      .doc(
      FirebaseAuth.instance.currentUser?.email.toString())
      .collection('classes')
      .snapshots(),
  builder: (context, snapshot) {
    if (!snapshot.hasData)
      return Align(
        child: CircularProgressIndicator(
          valueColor:
          AlwaysStoppedAnimation<Color>
            (Colors.redAccent),
        ), // CircularProgressIndicator
      ); // Align
```

Figure 13: Streambuilder implementation

Wordcount: 995