Criterion C: Development

Vidhya Narayanan
Sakthi Kumar
Computer Science
2 March 2023

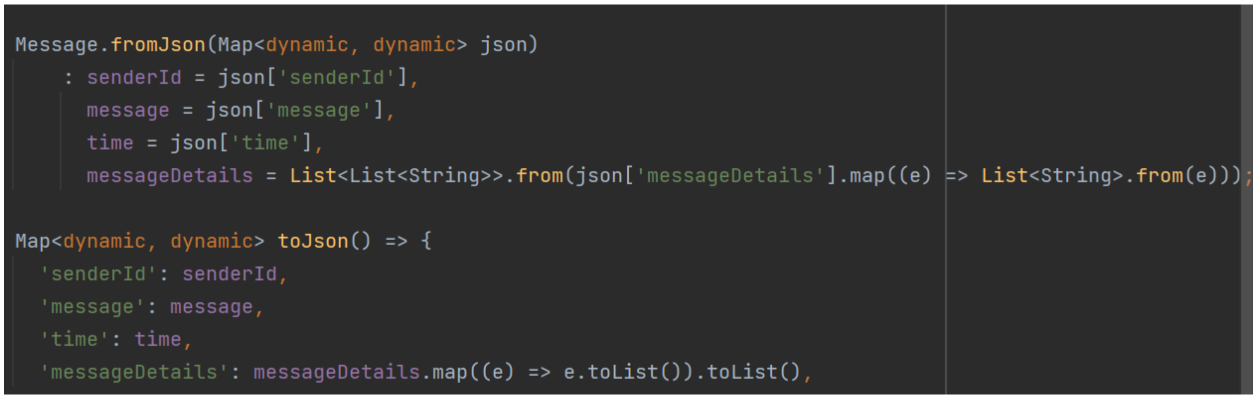
I PROPERLY CITE THE SOURCES USED TO HELP ME IN THE APPENDIX!

Key Techniques For Complexity:

1. Merging Sorted Data Structures
2. Searching For Specified Data Within a File
3. Inheritance
4. Polymorphism
5. Encapsulation
6. Parsing a Data Stream
7. Recursion
8. Hierarchal Composite Data Structure
9. Additional Libraries
10. Arrays of Two or More Dimensions

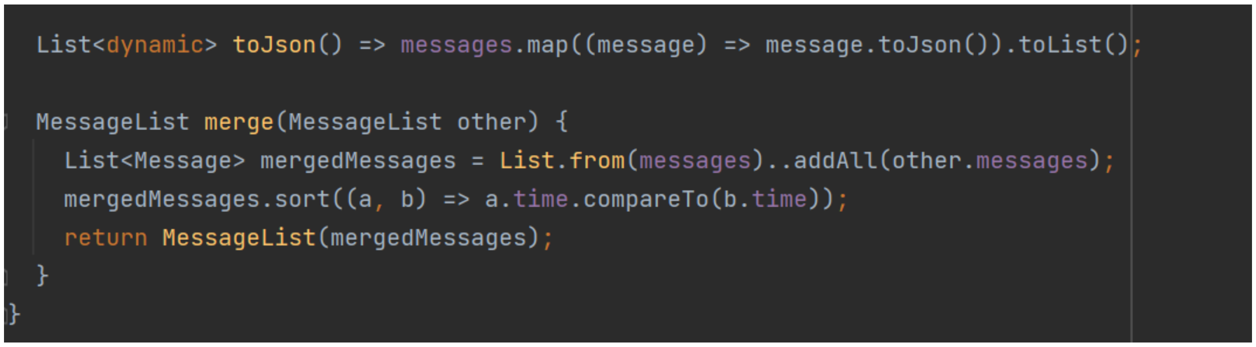
Merging Sorted Data Structures:

The first example of me merging two or more sorted data structures would be in the MessageList Class, a class that is dedicated towards merging **multiple** objects from the Message class together:



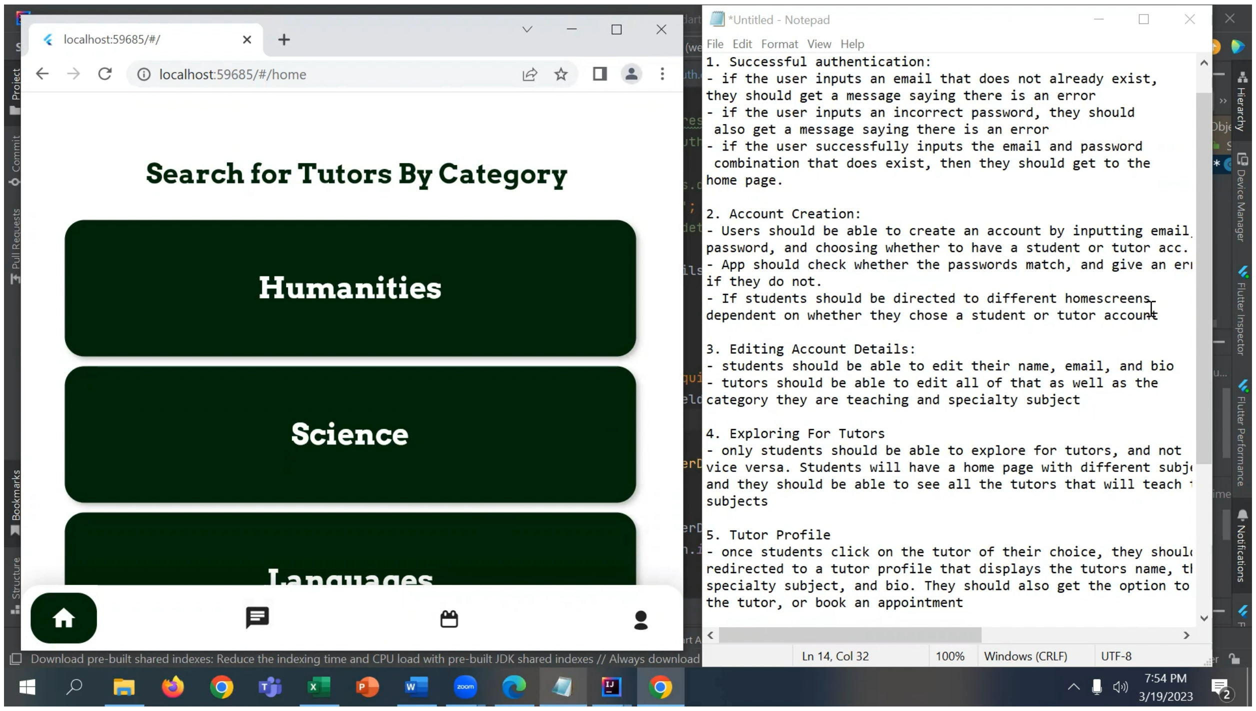
```
Message.fromJson(Map<dynamic, dynamic> json)
    : senderId = json['senderId'],
      message = json['message'],
      time = json['time'],
      messageDetails = List<List<String>>.from(json['messageDetails'].map((e) => List<String>.from(e)));

Map<dynamic, dynamic> toJson() => {
  'senderId': senderId,
  'message': message,
  'time': time,
  'messageDetails': messageDetails.map((e) => e.toList()).toList(),
```

As you can see, all messages are saved as Map Values in the message class that contain the ID of the person who sent it (Tutor or Student,) and these are sent to JSon to be uploaded to Firestore.


```
  List<dynamic> toJson() => messages.map((message) => message.toJson()).toList();

  MessageList merge(MessageList other) {
    List<Message> mergedMessages = List.from(messages)..addAll(other.messages);
    mergedMessages.sort((a, b) => a.time.compareTo(b.time));
    return MessageList(mergedMessages);
  }
}
```

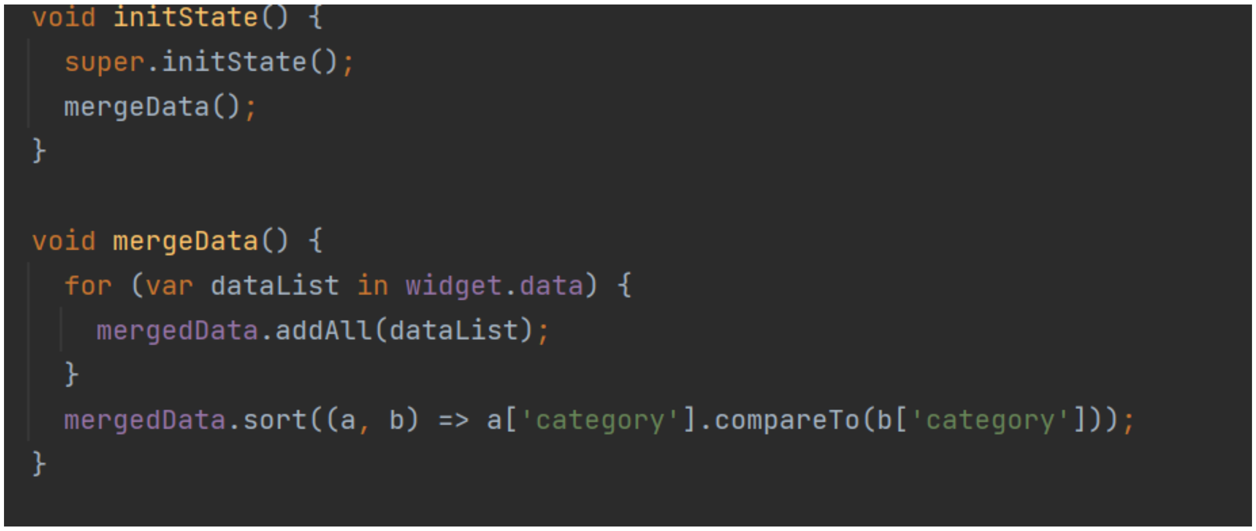
Using a merge() method, the MessageList class performs the merging, and then compares the times that different texts are sent in order to order them in the chat room. The ingenuity of this to my project is that since my app requires a chatroom, it is important that the messages are displayed in chronological order, regardless of who is sending the message, otherwise the conversation will make no sense (Source #1)

Another Example of Merging Sorted Data Structures in my project would be the explorelist class, which contains another mergeData() method. The purpose of this method is to loop through all of the users in the firestore “documents,” add them to the mergedList() and then sort them based on the specialized categories that different tutors focus on such as Humanities, Math, and Science so that when the user clicks on one of those tabs, the tutors are divided up by specialization.



```
1. Successful authentication:
- if the user inputs an email that does not already exist,
they should get a message saying there is an error
- if the user inputs an incorrect password, they should
 also get a message saying there is an error
- if the user successfully inputs the email and password
 combination that does exist, then they should get to the
home page.

2. Account Creation:
- Users should be able to create an account by inputting email
password, and choosing whether to have a student or tutor acc.
- App should check whether the passwords match, and give an er
if they do not.
- If students should be directed to different homescreens
dependent on whether they chose a student or tutor account

3. Editing Account Details:
- students should be able to edit their name, email, and bio
- tutors should be able to edit all of that as well as the
category they are teaching and specialty subject

4. Exploring For Tutors
- only students should be able to explore for tutors, and not
vice versa. Students will have a home page with different subj
and they should be able to see all the tutors that will teach
subjects

5. Tutor Profile
- once students click on the tutor of their choice, they shoul
redirected to a tutor profile that displays the tutors name, t
specialty subject, and bio. They should also get the option to
the tutor, or book an appointment
```

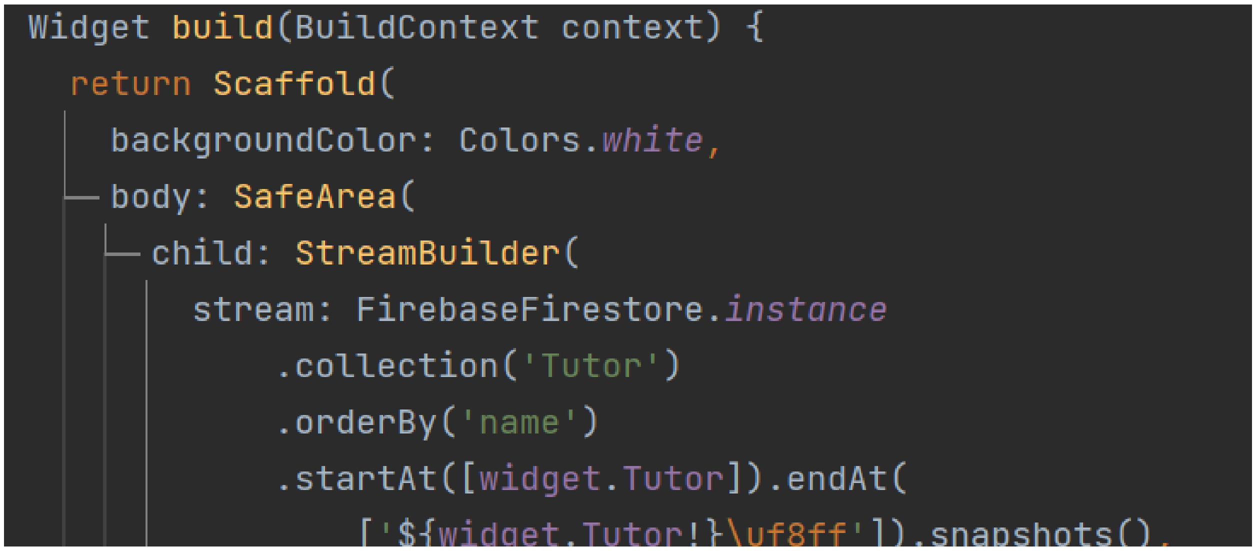
The ingenuity of this is that it creates an easy way to sort tutors into categories that makes it easier for students to search for what they want while also meeting the requirement for merging data structures (Source #2)!



```
void initState() {
  super.initState();
  mergeData();
}

void mergeData() {
  for (var dataList in widget.data) {
    mergedData.addAll(dataList);
  }
  mergedData.sort((a, b) => a['category'].compareTo(b['category']));
}
```

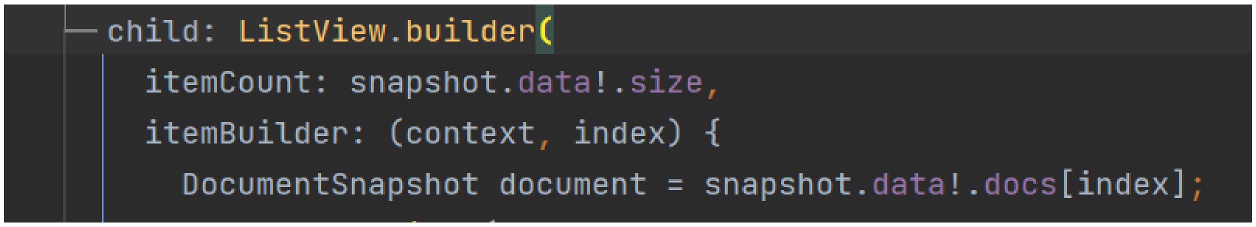
2. Searching For Specified Data Within a File

Since my app is connected to database, and involves a lot of adding new data and changing data, it is obvious that there would be a lot of searching for specified data within a file. Some of the most important examples would be in the TutorProfile Class. The class takes in a Tutor parameter, which is used as a query to search for a tutor in the collection. It then displays the tutor's profile information, such as their name, specialization, specification, and bio. The build method returns a Scaffold widget with a StreamBuilder as its body. The StreamBuilder listens to the Tutor query in the Firestore collection using the FirebaseFirestore package (Source #3).



```
Widget build(BuildContext context) {
  return Scaffold(
    backgroundColor: Colors.white,
    body: SafeArea(
      child: StreamBuilder(
        stream: FirebaseFirestore.instance
            .collection('Tutor')
            .orderBy('name')
            .startAt([widget.Tutor]).endAt(
                ['${widget.Tutor!}\uf8ff']).snapshots(),
```

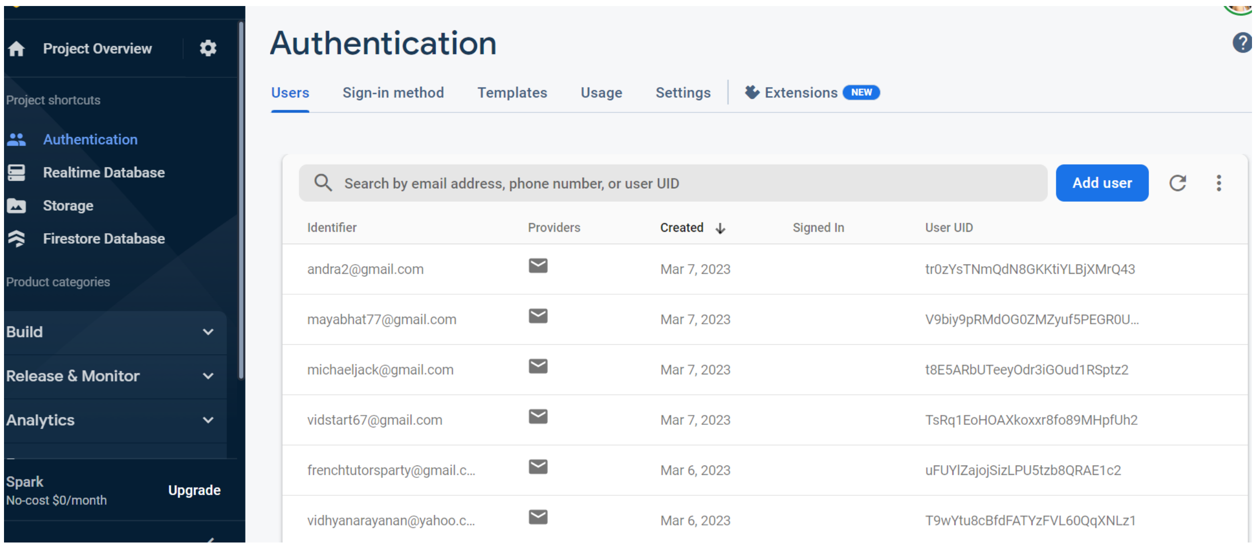
If there is data inside the FirestoreInstance, the ListviewBuilder is activated and then data is pulled from the document inside Firebase that holds Tutor Data:



```
      child: ListView.builder(
        itemCount: snapshot.data!.size,
        itemBuilder: (context, index) {
          DocumentSnapshot document = snapshot.data!.docs[index];
```

Another Significant instance of a class intaking data from a file would be in TutorOrStudent Class. When a user first registers for the app, one of the decisions they make is whether they will have a student or a tutor account. Therefore, once their information is stored, its important that the TutorOrStudent can use the documents from firestore to identify the user type to send the user to the right screen. This ingenuity of this to my app is that My app involves a different layout for students and tutors. Therefore it's important that student and tutor accounts are differentiated right off the bat.

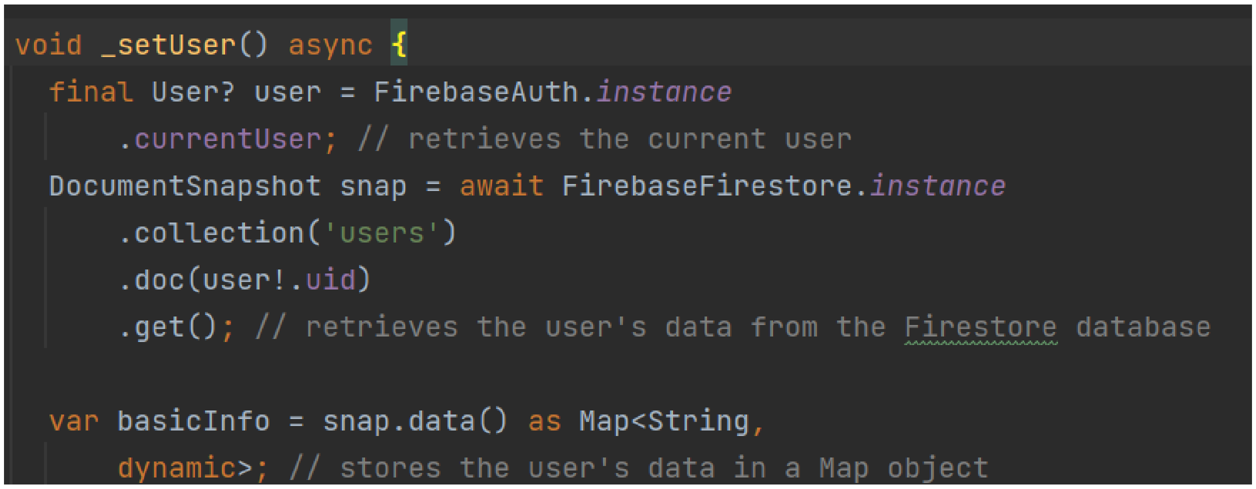
F



Authentication

Identifier	Providers	Created	Signed In	User UID
andra2@gmail.com	✉	Mar 7, 2023		tr0zYsTNmQdN8GKKtiYLBjXMrQ43
mayabhat77@gmail.com	✉	Mar 7, 2023		V9biy9pRMdOG0ZMZyuf5PEGR0U...
michaeljack@gmail.com	✉	Mar 7, 2023		t8E5ARbUTeeyOdr3iGOud1RSptz2
vidstart67@gmail.com	✉	Mar 7, 2023		TsRq1EoHOAXkoxxr8fo89MHpfUh2
frenchtutorsparty@gmail.c...	✉	Mar 6, 2023		uFUYIZajojSizLPU5tzb8QRAE1c2
vidhyanarayanan@yahoo.c...	✉	Mar 6, 2023		T9wYtu8cBfdFATYzFVL60QqXNLz1

Firebase Storage

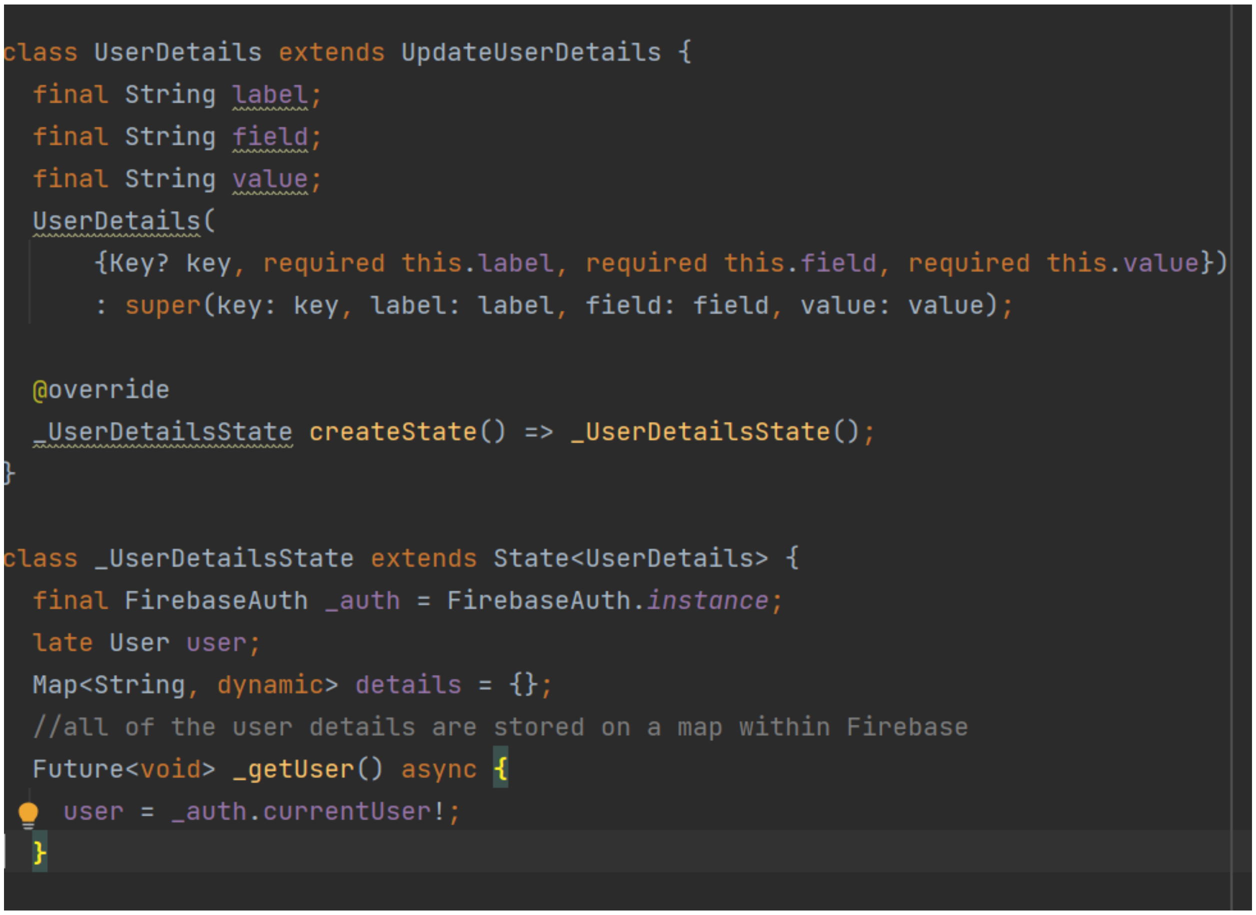


```
void _setUser() async {
  final User? user = FirebaseAuth.instance
      .currentUser; // retrieves the current user
  DocumentSnapshot snap = await FirebaseFirestore.instance
      .collection('users')
      .doc(user!.uid)
      .get(); // retrieves the user's data from the Firestore database

  var basicInfo = snap.data() as Map<String,
      dynamic>; // stores the user's data in a Map object
```

3. Inheritance

Two of the most significant instances of inheritance in this would be that of the settings and the userdetails class, which both extend the updateuserdetails class. The Settings class also exhibits polymorphism with the updateuserdetails class but that will be discussed later. In the userdetails class, inheritance is executed by holding objects that can serve to update user details. Then we can create a list of UserInformation objects and call the updateUserDetails()

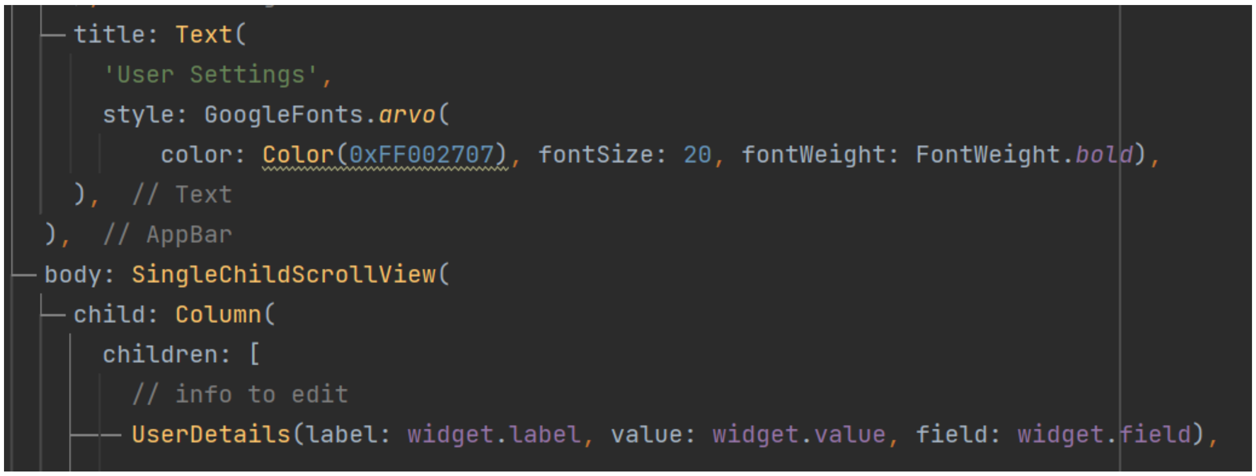
method on each object in the list. The ingenuity of this is that it allows us to easily update user details for multiple types of user information objects. Since tutors and students are high school students bound to change, its important that they can easily change settings and the app is able to properly process it.



```
class UserDetails extends UpdateUserDetails {
  final String label;
  final String field;
  final String value;
  UserDetails(
      {Key? key, required this.label, required this.field, required this.value})
      : super(key: key, label: label, field: field, value: value);

  @override
  _UserDetailsState createState() => _UserDetailsState();
}

class _UserDetailsState extends State<UserDetails> {
  final FirebaseAuth _auth = FirebaseAuth.instance;
  late User user;
  Map<String, dynamic> details = {};
  //all of the user details are stored on a map within Firebase
  Future<void> _getUser() async {
    user = _auth.currentUser!;
  }
```

As you can see, the UserDetails class includes all the fields and parameters from update user details so that all the user details can be updated in a new instance of firebase, which is instantiated in the very top of the _UserDetailsState. This is important to my app because as previously stated, users need to be able to update their data because of the volatile nature of tutoring.

Since the settings class is WHERE the users are able to change their information, it is important that that class inherits from the UpdateUserDetails class especially. This class needs to inherit the same parameters and fields as the UserDetails class, so that screenshot is unnecessary. But one way the settings class is different is through this:



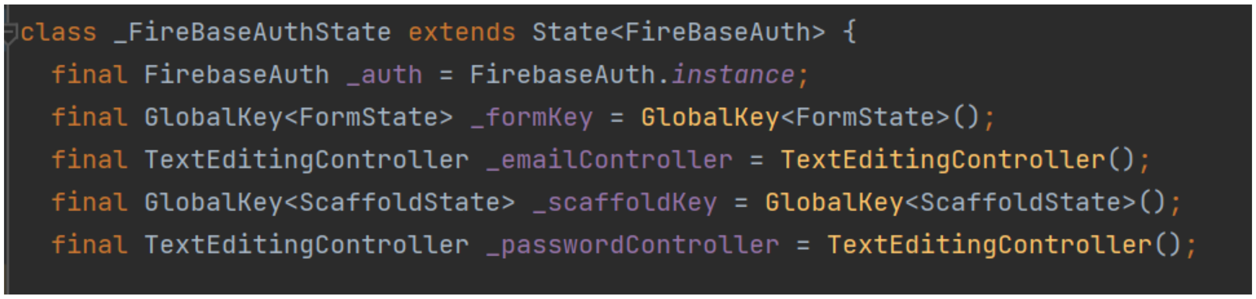
```
    title: Text(
      'User Settings',
      style: GoogleFonts.arvo(
          color: Color(0xFF002707), fontSize: 20, fontWeight: FontWeight.bold),
    ),  // Text
  ),  // AppBar
  body: SingleChildScrollView(
    child: Column(
      children: [
        // info to edit
        UserDetails(label: widget.label, value: widget.value, field: widget.field),
```

As you can see, the top of the screen has a label saying user settings, but the body is compromised of a column that contains the different user details stored in Firebase in the UserDetails class such as the bio, name, email, and for tutors the category and specific subject specialization. Since the settings class inherits the UpdateUserDetails as well, its able to process what happens when the user changes their settings and saves it for the next time they log in / even just use the application.

4. Polymorphism

As previously stated, the Settings class displays characteristic of polymorphism. This is because by extending the UpdateUserDetails abstract class, UserSettings inherits its properties and methods, including the abstract method updateUserDetails(). Since UserSettings is a concrete class that extends UpdateUserDetails, it can provide its own implementation of updateUserDetails(). Additionally, the UserSettings class implements the StatelessWidget interface, which means it can be used as a widget in the Flutter UI tree. This allows UserSettings to be used as a component in building a larger UI, making it more flexible and reusable (Source #4).

5. Encapsulation:

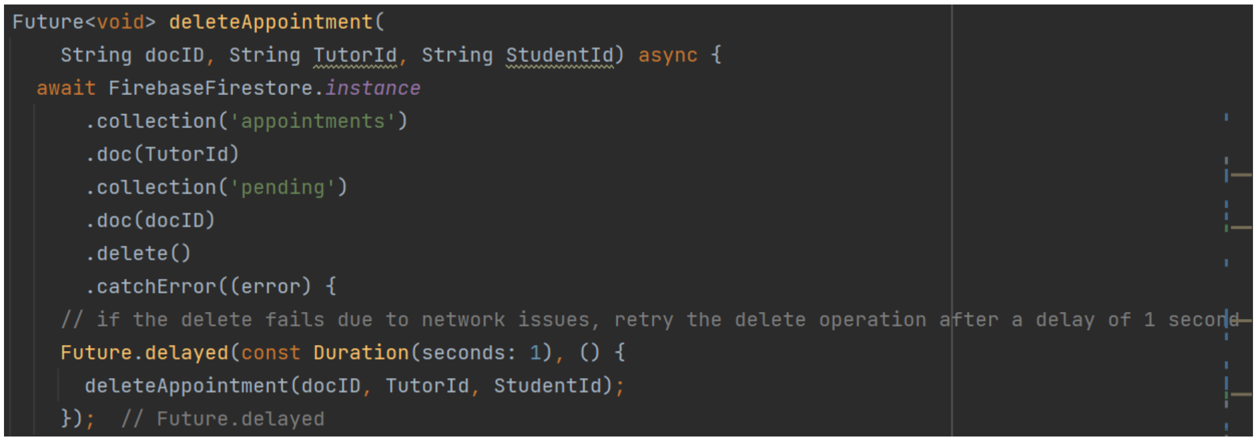


```
class _FireBaseAuthState extends State<FireBaseAuth> {
  final FirebaseAuth _auth = FirebaseAuth.instance;
  final GlobalKey<FormState> _formKey = GlobalKey<FormState>();
  final TextEditingController _emailController = TextEditingController();
  final GlobalKey<ScaffoldState> _scaffoldKey = GlobalKey<ScaffoldState>();
  final TextEditingController _passwordController = TextEditingController();
```

One of the most significant uses of encapsulation in this application would be in the Firebase Authorization class, which is basically inputs all the primary data users provide into firebase for the first time, meaning it is where all the initialization takes place. The class has several private instance variables such as _auth, _formKey, _emailController, _scaffoldKey, _passwordController, as shown above. The ingenuity of this is that it makes the private, so that these variables are only accessible to this class even if it is inherited. Other classes would not have a need for a form key or a password controller, increasing security.

6.) Recursion

There were not too many instances in this app where recursion was needed, but one of the most ingenuine uses of recursion in this app was in the AppointmentList class, specifically in the deleteAppointment() method. When i first coded the app, I struggled with network issues that made it difficult to delete appointments because that involved removing the appointment from firebase too. In order to fix this, I added recursion to the deleteAppointment method so that it will retry the operation from the beginning if it fails after one whole second

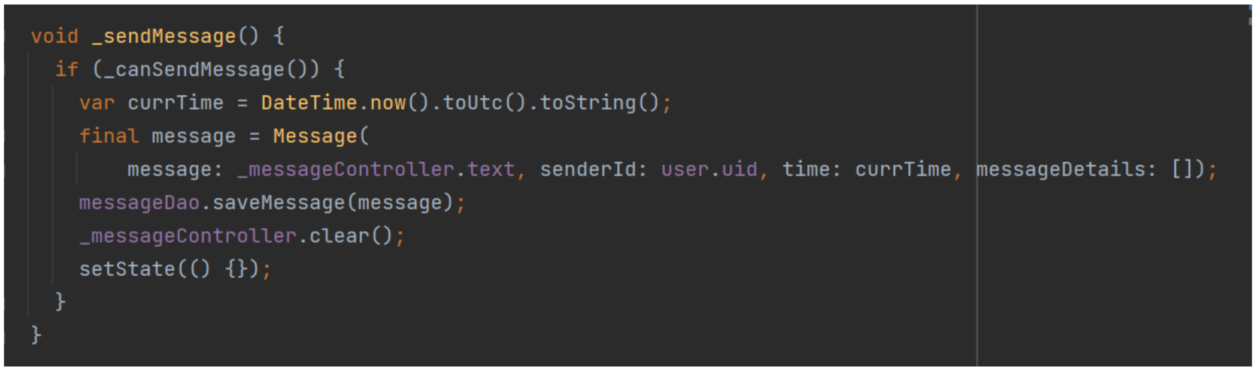


```
Future<void> deleteAppointment(
    String docID, String TutorId, String StudentId) async {
  await FirebaseFirestore.instance
      .collection('appointments')
      .doc(TutorId)
      .collection('pending')
      .doc(docID)
      .delete()
      .catchError((error) {
    // if the delete fails due to network issues, retry the delete operation after a delay of 1 second
    Future.delayed(const Duration(seconds: 1), () {
      deleteAppointment(docID, TutorId, StudentId);
    }); // Future.delayed
```

As shown above, I managed to get the app to time one second by doing a future delay for one second, and then called the deleteAppointment() method within itself.

7.) Parsing a Data Stream:

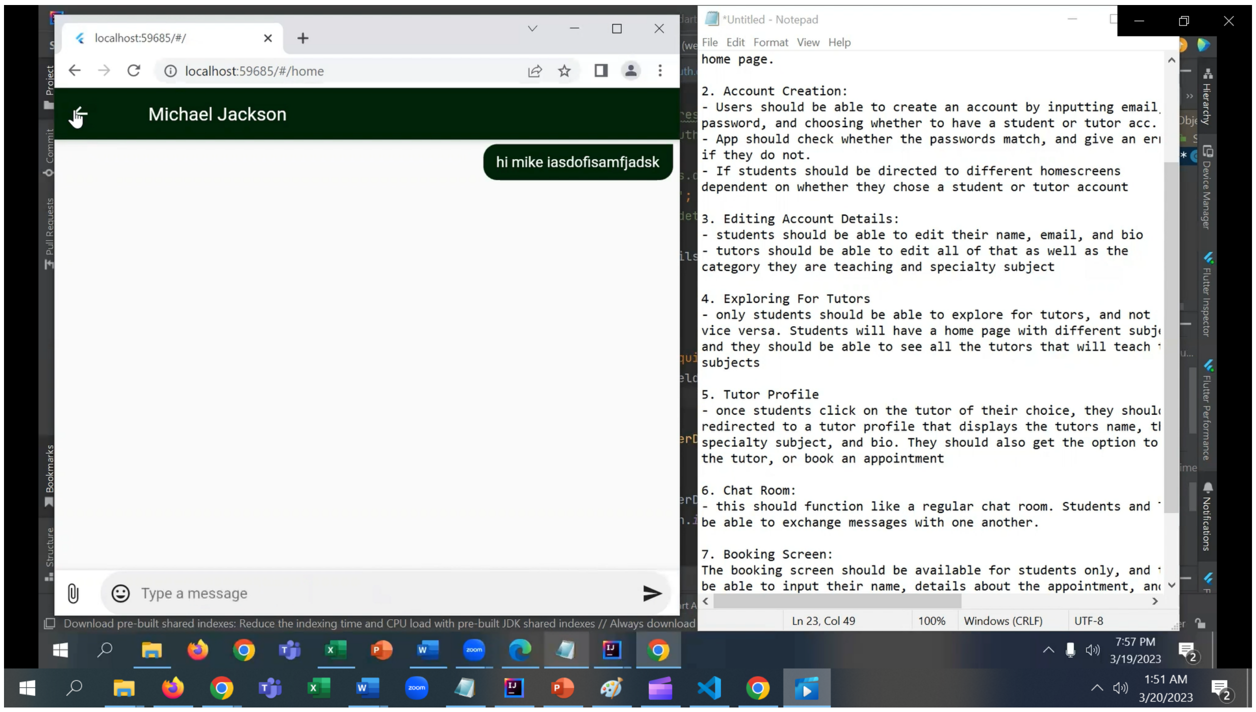
- One notable instance of parsing a data stream within this app is in the chatRoom() class. This class uses the StreamBuilder widget provided by Flutter to listen to a stream and build the UI accordingly.



```
void _sendMessage() {
  if (_canSendMessage()) {
    var currTime = DateTime.now().toUtc().toString();
    final message = Message(
        message: _messageController.text, senderId: user.uid, time: currTime, messageDetails: []);
    messageDao.saveMessage(message);
    _messageController.clear();
    setState(() {});
  }
}
```

  - As shown above, the messages aren't populated manually but rather by the streambuilder that listens to the stream fo data from firebase that is the messages people send.

First, the init_() method Initializes a new ChatRoom object with a name, an empty list of users, and an empty list of message and adds a new user to the chatroom, sends a welcome message to all users in the chatroom, and appends the welcome message to the messages list.



```
home page.

2. Account Creation:
- Users should be able to create an account by inputting email,
password, and choosing whether to have a student or tutor acc.
- App should check whether the passwords match, and give an er
if they do not.
- If students should be directed to different homescreens
dependent on whether they chose a student or tutor account

3. Editing Account Details:
- students should be able to edit their name, email, and bio
- tutors should be able to edit all of that as well as the
category they are teaching and specialty subject

4. Exploring For Tutors
- only students should be able to explore for tutors, and not
vice versa. Students will have a home page with different subj
and they should be able to see all the tutors that will teach
subjects

5. Tutor Profile
- once students click on the tutor of their choice, they shoul
redirected to a tutor profile that displays the tutors name, t
specialty subject, and bio. They should also get the option to
the tutor, or book an appointment

6. Chat Room:
- this should function like a regular chat room. Students and
be able to exchange messages with one another.

7. Booking Screen:
The booking screen should be available for students only, and
be able to input their name, details about the appointment, an
```

Chatroom example:

8.) Additional Libraries.

This Project has a LOT of additional libraries. Here are a few:

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

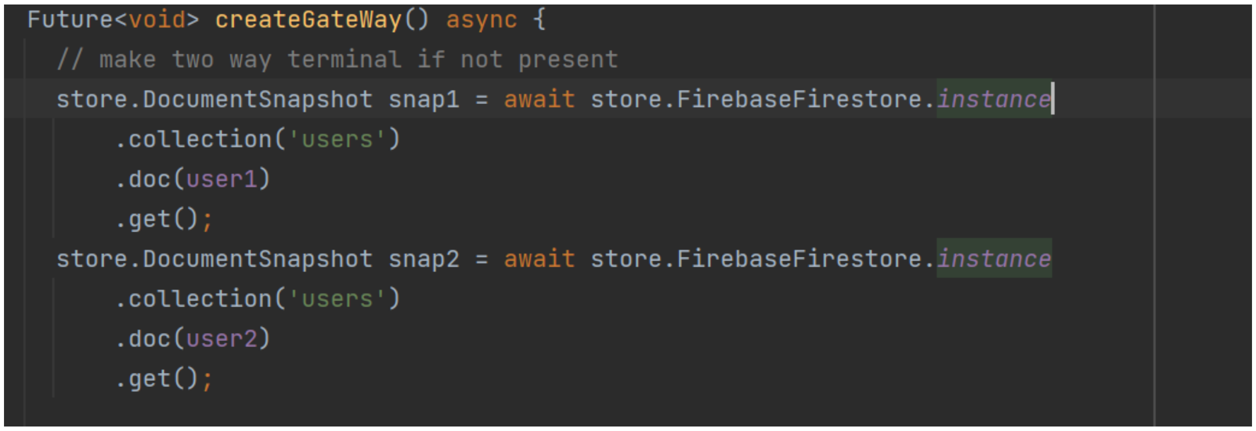
The fonts are needed so the app can have some aesthetic ingenuity. The intl package is important so that firebase knows what times the messages are sent at so that the messages can be sent in the correct order in the chatroom, as emphasized earlier.

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

This import is what allows for UI to even exist at all.

9.) Hierarchical Data Structure

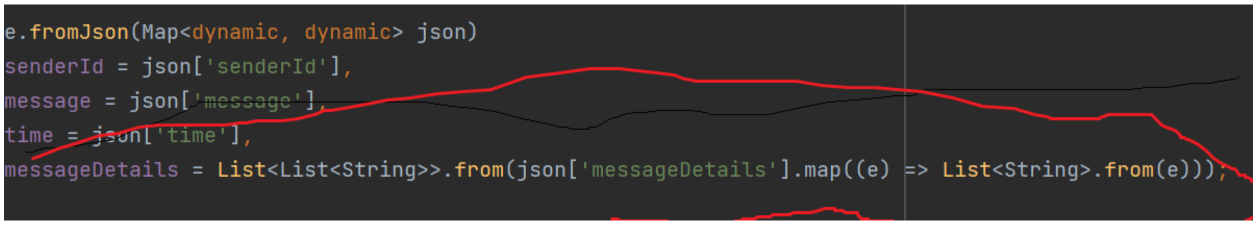
The MessageData class is an example of a hierarchical data structure. Inside this class, the Firebase Realtime Database is in reference to a specific node in the database, which is determined by the chatRoomId string.



```
Future<void> createGateWay() async {
  // make two way terminal if not present
  store.DocumentSnapshot snap1 = await store.FirebaseFirestore.instance
      .collection('users')
      .doc(user1)
      .get();
  store.DocumentSnapshot snap2 = await store.FirebaseFirestore.instance
      .collection('users')
      .doc(user2)
      .get();
```

This node contains a chat child node where messages between two users are stored. The data in this node is organized hierarchically with each message being a child node of the chat node, and each message having its own unique identifier as its key (Source #5)

10.) 2-D or higher Arrays



```
e.fromJson(Map<dynamic, dynamic> json)
senderId = json['senderId'],
message = json['message'],
time = json['time'],
messageDetails = List<List<String>>.from(json['messageDetails'].map((e) => List<String>.from(e)));
```

As shown in the code snippet above from the MessageDetails() class, I added a constructor named messageDetails which is essentially an array that is a list of lists of strings, which gives it the properties of a two dimensional array. This is ingenuine towards my application as it provides a simple storage mechanism to store different conversations: one ‘row’ would represent the strings of text in one conversation between a student and a tutor, and another would represent a conversation between that same student and a different tutor, and there would be a different instance of messageDetails for each student.

WORD COUNT : 1,153