Criterion C: Development

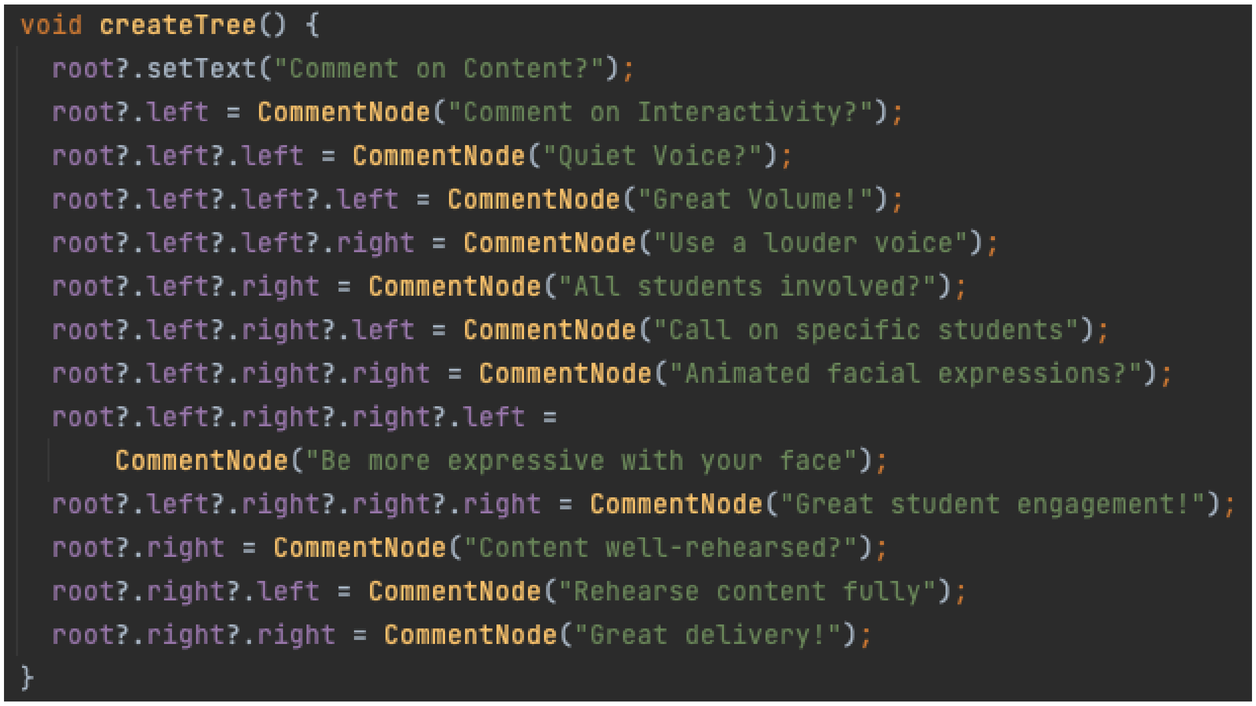
Summary of Techniques:

- Binary Trees
    - Show overseer comment suggestions based on intern experience
- Searching for and Displaying Student/Intern Data
- Editing, Adding, and Deleting Firestore Data
- User Authentication, Account Creation

Log-out not listed as a major technique: user is just redirected to home screen.

Suggestions Binary Trees:

I created a class called CommentNode.dart, which stored a String? _text, a CommentNode? left, and a CommentNode? right. I utilized encapsulation, making _text private and implementing getters and setters for cleaner code. The fields of CommentTree.dart were CommentNode? root, List? path, and bool? searchedFound. Utilizing a method called createTree, I utilized node branches and hard-coded the tree for an intern with one year of experience.

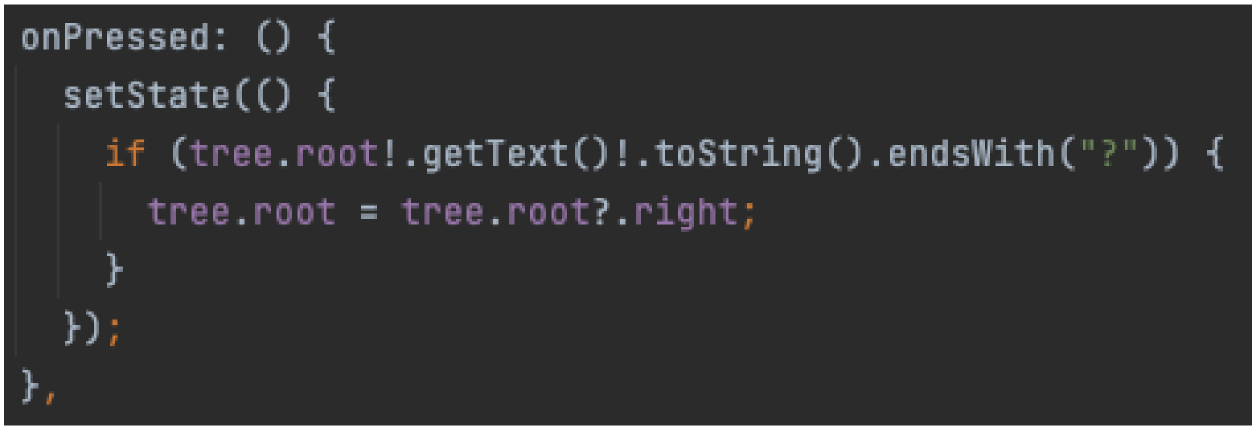


```
void createTree() {
  root?.setText("Comment on Content?");
  root?.left = CommentNode("Comment on Interactivity?");
  root?.left?.left = CommentNode("Quiet Voice?");
  root?.left?.left?.left = CommentNode("Great Volume!");
  root?.left?.left?.right = CommentNode("Use a louder voice");
  root?.left?.right = CommentNode("All students involved?");
  root?.left?.right?.left = CommentNode("Call on specific students");
  root?.left?.right?.right = CommentNode("Animated facial expressions?");
  root?.left?.right?.right?.left =
      CommentNode("Be more expressive with your face");
  root?.left?.right?.right?.right = CommentNode("Great student engagement!");
  root?.right = CommentNode("Content well-rehearsed?");
  root?.right?.left = CommentNode("Rehearse content fully");
  root?.right?.right = CommentNode("Great delivery!");
}
```

Figure 1: createTree() in CommentTree.dart

In OneYearSuggestion.dart, I displayed the tree by using “Yes” and “No” TextButtons along with Text widget. I created an instance of CommentTree, and assigned the root the value of root.left or root.right for “Yes” and “No,” respectively, allowing me to access branches.

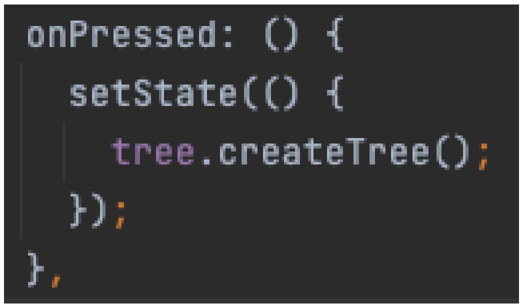
Traversal only occured if the text was a question (not final suggestion).



```
onPressed: () {
  setState(() {
    if (tree.root!.getText()!.toString().endsWith("?")) {
      tree.root = tree.root?.right;
    }
  });
},
```

Figure 2: onPressed() in _yesButton()

At any point, the user can click the “Start/Reset” button, which calls createTree and hence resets the root node (and tree).



```
onPressed: () {
  setState(() {
    tree.createTree();
  });
},
```

Figure 3: onPressed in _startButton()

The implementation of insert and remove was incredibly challenging, as I had been accustomed to using binary search trees with sorted branches. First, I created a method which found the path to text within the tree.

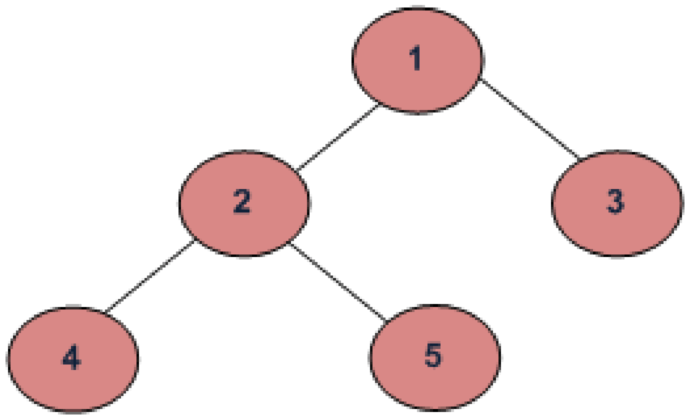
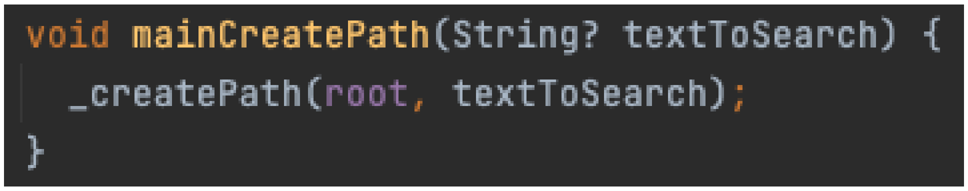


Figure 4: Binary Tree example from GeeksforGeeks

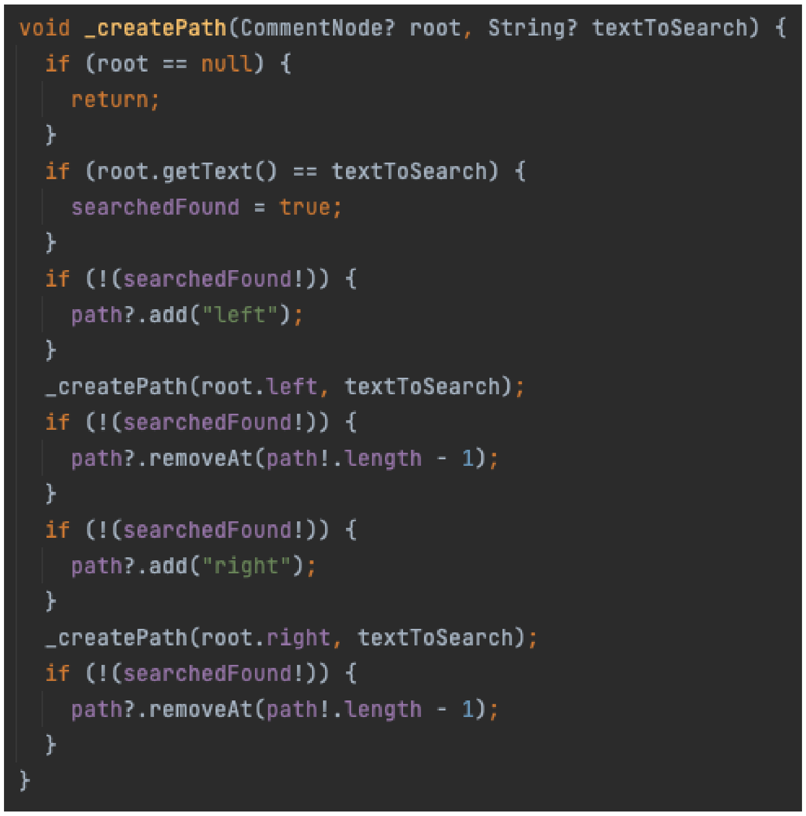
I utilized pre-order traversal (“Tree Traversals (Inorder, Preorder, and Postorder)”) to sift through my tree (example order: 12543). To find the path, I created a public function which called my private recursive function.



```
void mainCreatePath(String? textToSearch) {
  _createPath(root, textToSearch);
}
```

Figure 5: mainCreatePath()

In_createPath(), before progressing, I checked for a null node to avoid a NullPointerException. Then, if the current node held the text, I set searchedFound to true to prevent further path modification. Next, I added "left" to path, recursively called _createPath() with root.left substituting root, and delete the path's last String ("left"). I replicated the process with "right." The algorithm works as pre-order traversal will keep adding to the list, and then removing when rising back from lower tree levels. When the text is found, the path is correct and will not change due to searchedFound.



```
void _createPath(CommentNode? root, String? textToSearch) {
  if (root == null) {
    return;
  }
  if (root.getText() == textToSearch) {
    searchedFound = true;
  }
  if (!(searchedFound!)) {
    path?.add("left");
  }
  _createPath(root.left, textToSearch);
  if (!(searchedFound!)) {
    path?.removeAt(path!.length - 1);
  }
  if (!(searchedFound!)) {
    path?.add("right");
  }
  _createPath(root.right, textToSearch);
  if (!(searchedFound!)) {
    path?.removeAt(path!.length - 1);
  }
}
```

Figure 6: _createPath()

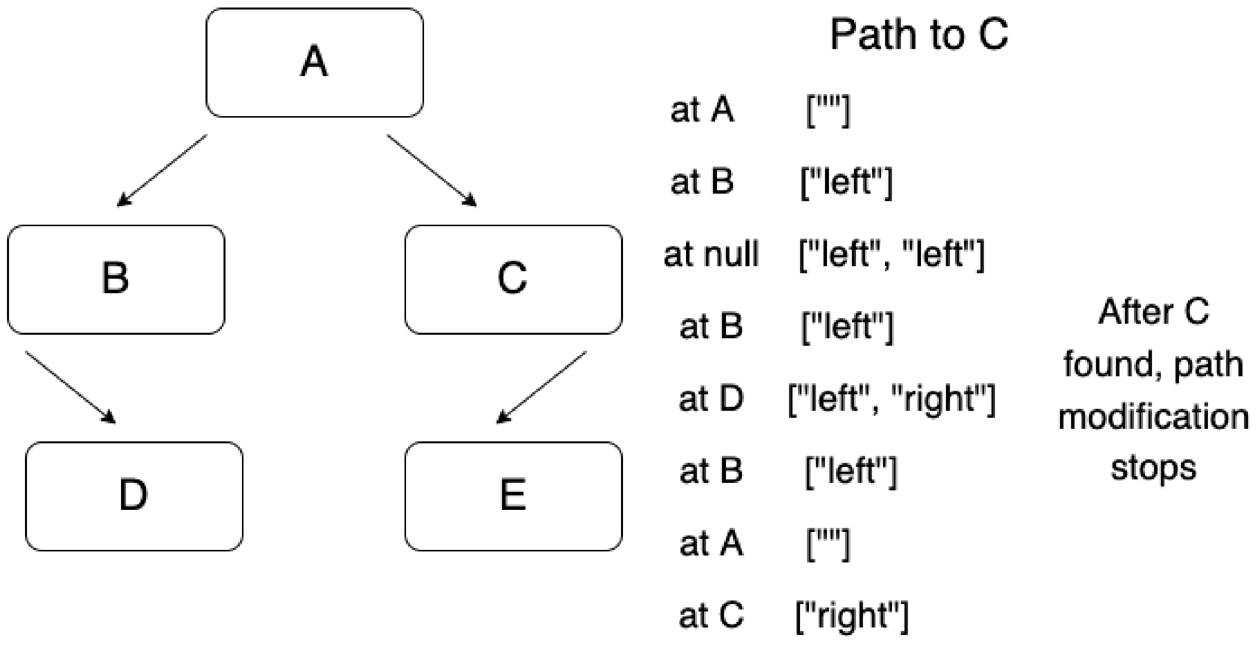
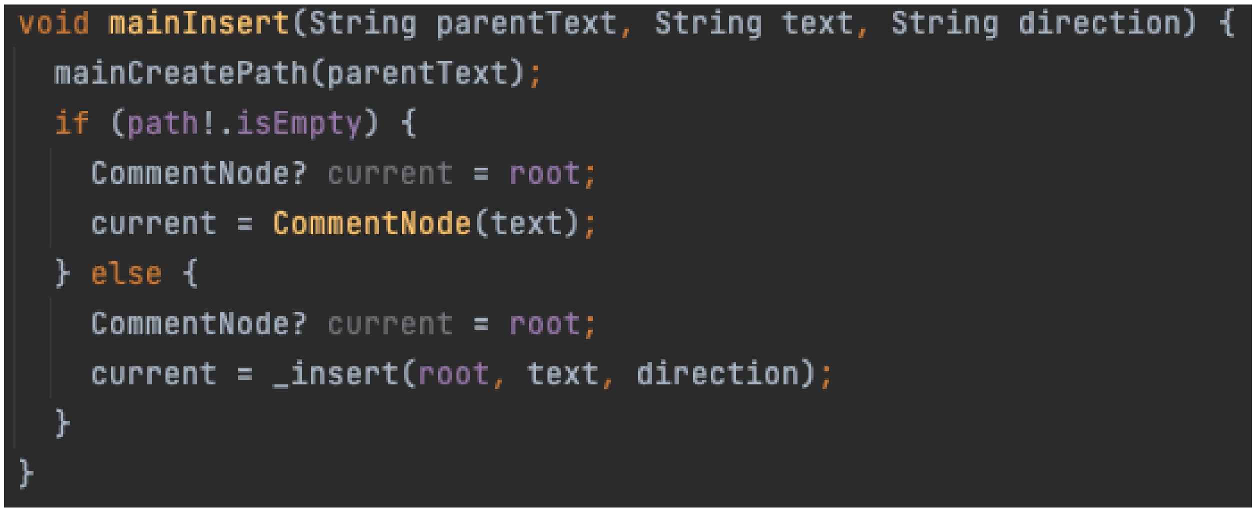


Figure 7: Creating Path

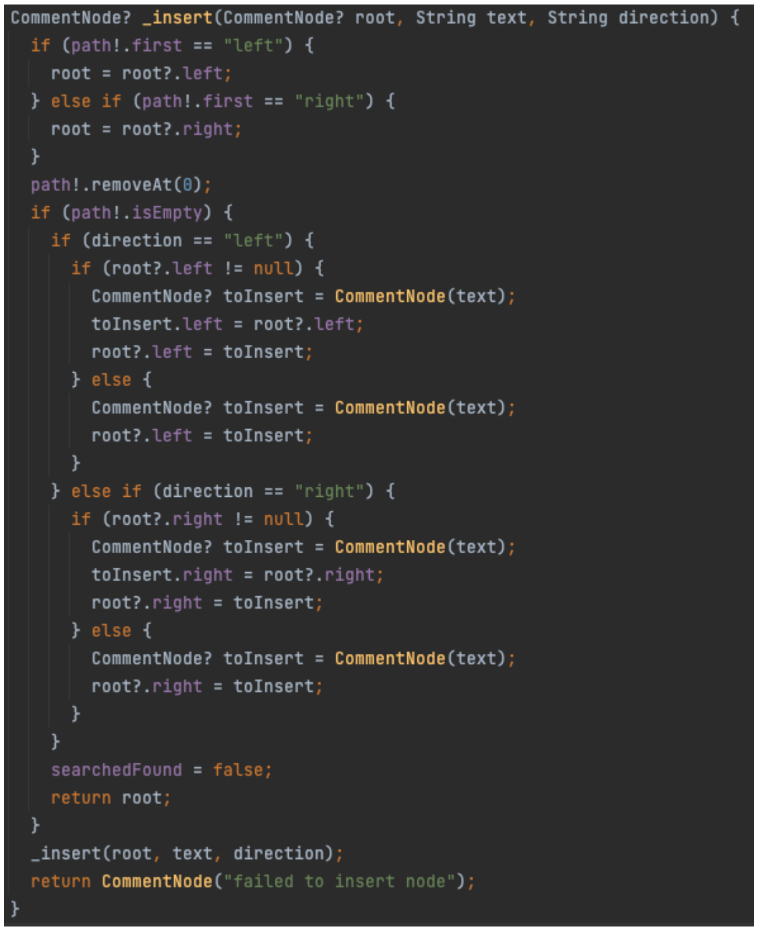
I utilized path in mainInsert(). I error-checked, assigning the root the text for an empty path. I assigned the value of the call of the recursive _insert() to CommentNode? current, which already points to the root. As the root undergoes modification through traversal (parents erased), current enables a functional insertion on the tree's main node.



```
void mainInsert(String parentText, String text, String direction) {
  mainCreatePath(parentText);
  if (path!.isEmpty) {
    CommentNode? current = root;
    current = CommentNode(text);
  } else {
    CommentNode? current = root;
    current = _insert(root, text, direction);
  }
}
```

Figure 8: mainInsert()

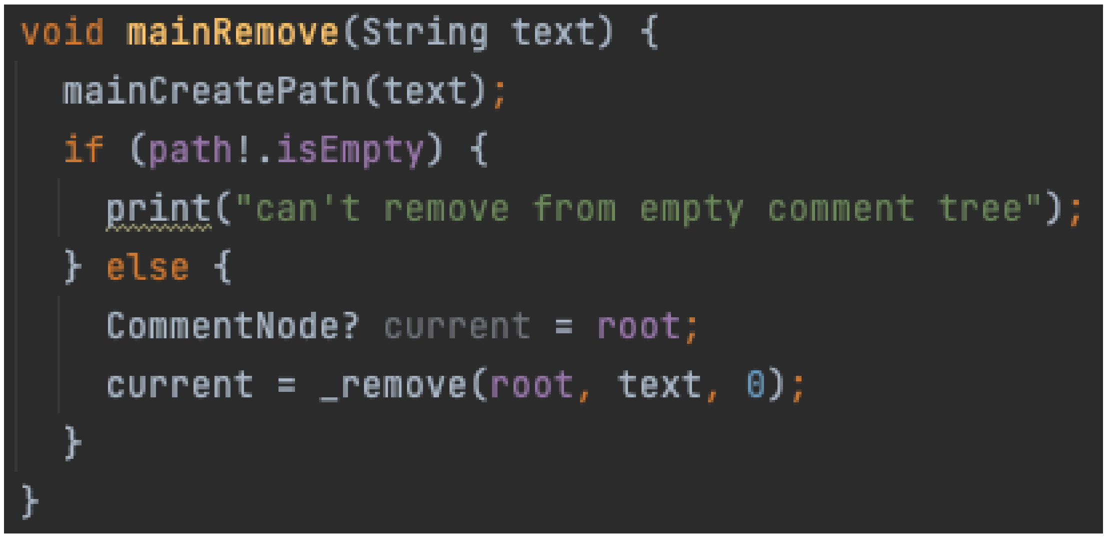
The function iterates through path by traversing based on the list's first element, which is subsequently removed. Once the path is empty, a new node with specified text is added as a branch in specified direction, with any branches rerouted to branch off this new node. The function is recursively called on branches until the insertion when searchedFound is set to false to enable new paths.



```
CommentNode? _insert(CommentNode? root, String text, String direction) {
  if (path!.first == "left") {
    root = root?.left;
  } else if (path!.first == "right") {
    root = root?.right;
  }
  path!.removeAt(0);
  if (path!.isEmpty) {
    if (direction == "left") {
      if (root?.left != null) {
        CommentNode? toInsert = CommentNode(text);
        toInsert.left = root?.left;
        root?.left = toInsert;
      } else {
        CommentNode? toInsert = CommentNode(text);
        root?.left = toInsert;
      }
    } else if (direction == "right") {
      if (root?.right != null) {
        CommentNode? toInsert = CommentNode(text);
        toInsert.right = root?.right;
        root?.right = toInsert;
      } else {
        CommentNode? toInsert = CommentNode(text);
        root?.right = toInsert;
      }
    }
    searchedFound = false;
    return root;
  }
  _insert(root, text, direction);
  return CommentNode("failed to insert node");
}
```

Figure 9: _insert()

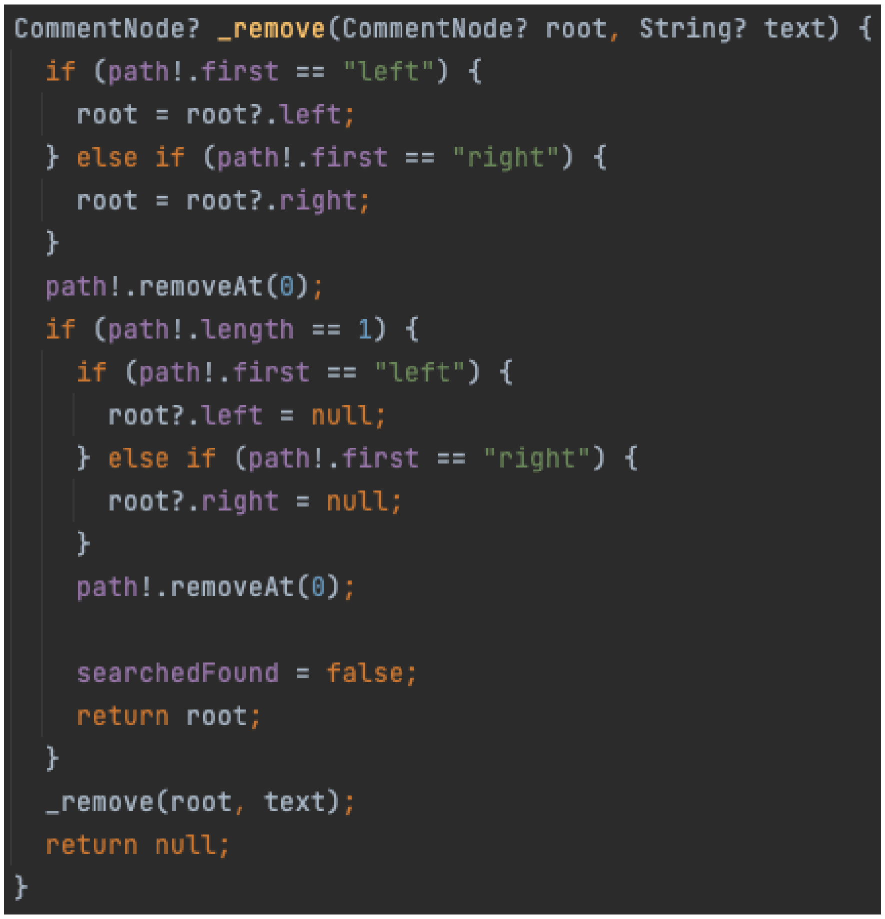
The path was similarly used with removal, with mainRemove() only calling _remove for an existing path.



```
void mainRemove(String text) {
  mainCreatePath(text);
  if (path!.isEmpty) {
    print("can't remove from empty comment tree");
  } else {
    CommentNode? current = root;
    current = _remove(root, text, 0);
  }
}
```

Figure 10: mainRemove()

In the recursive _remove(), traversal was identical to _insert(). To remove, the node was set to null, making its branches null too as it does not make sense for these branches to work under other tree questions.

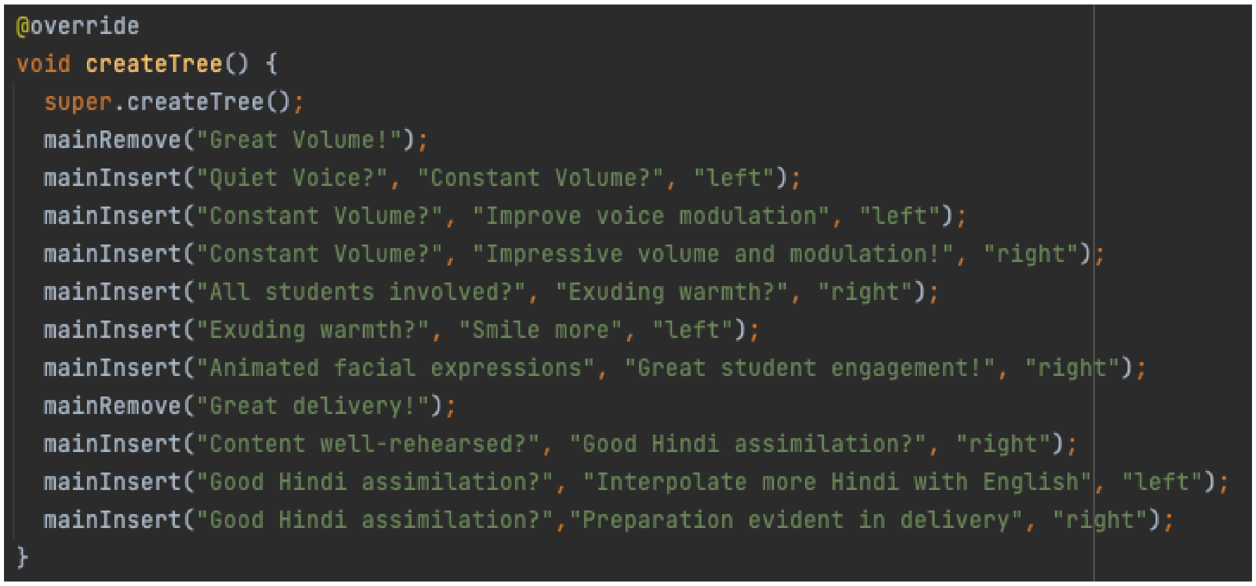


```
CommentNode? _remove(CommentNode? root, String? text) {
  if (path!.first == "left") {
    root = root?.left;
  } else if (path!.first == "right") {
    root = root?.right;
  }
  path!.removeAt(0);
  if (path!.length == 1) {
    if (path!.first == "left") {
      root?.left = null;
    } else if (path!.first == "right") {
      root?.right = null;
    }
    path!.removeAt(0);

    searchedFound = false;
    return root;
  }
  _remove(root, text);
  return null;
}
```

Figure 11: _remove()

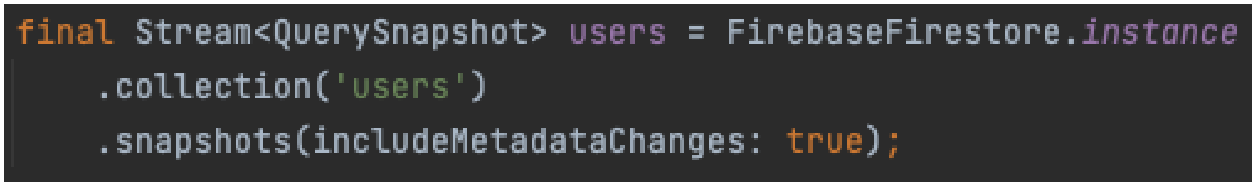
For interns with multiple years of experience, I displayed a different BinaryTree, implemented through SecondCommentTree.dart which extended CommentTree.dart. I used inheritance as it enables functional creation and editing of new CommentTree classes (Rasmussen). I implemented polymorphism, overriding the createTree() method of CommentTree. SecondCommentTree.dart was instantiated in ExperiencedSuggestion.dart like CommentTree's instantiation in OneYearSuggestion.dart, and the user was redirected to the respective page from AdminSearch based on years of experience. The Binary Tree proved the most effective and appropriate method for suggestions due to code reusability, complexity, and its dynamic nature which allowed multiple classes.



```
@override
void createTree() {
  super.createTree();
  mainRemove("Great Volume!");
  mainInsert("Quiet Voice?", "Constant Volume?", "left");
  mainInsert("Constant Volume?", "Improve voice modulation", "left");
  mainInsert("Constant Volume?", "Impressive volume and modulation!", "right");
  mainInsert("All students involved?", "Exuding warmth?", "right");
  mainInsert("Exuding warmth?", "Smile more", "left");
  mainInsert("Animated facial expressions", "Great student engagement!", "right");
  mainRemove("Great delivery!");
  mainInsert("Content well-rehearsed?", "Good Hindi assimilation?", "right");
  mainInsert("Good Hindi assimilation?", "Interpolate more Hindi with English", "left");
  mainInsert("Good Hindi assimilation?","Preparation evident in delivery", "right");
}
```

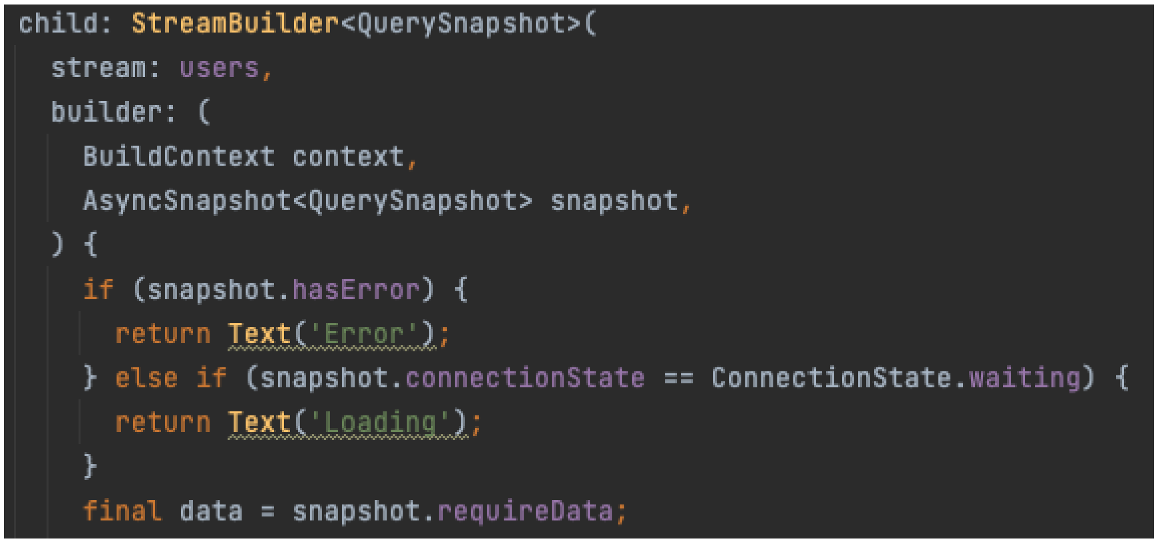
Figure 12: createTree() in SecondCommentTree.dart

Searching and Displaying Data:
I utilized a StreamBuilder: my stream was the snapshots of the users or students collection, depending on whether I searched for interns or students. The builder's AsyncSnapshot snapshot was utilized to gather QuerySnapshot data, utilized by ListView.builder (FlutterFire). I read all data from these indexed documents in the collection, and I queried values by referring to specific keys (Max on Flutter).



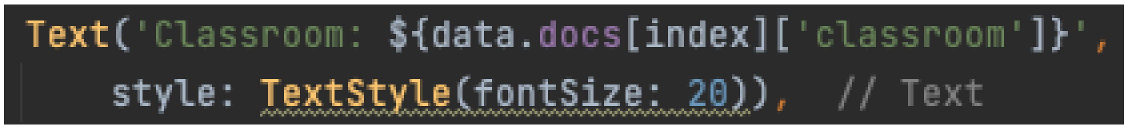
```
final Stream<QuerySnapshot> users = FirebaseFirestore.instance
    .collection('users')
    .snapshots(includeMetadataChanges: true);
```

Figure 13: intern Stream



```
child: StreamBuilder<QuerySnapshot>(
  stream: users,
  builder: (
    BuildContext context,
    AsyncSnapshot<QuerySnapshot> snapshot,
  ) {
    if (snapshot.hasError) {
      return Text('Error');
    } else if (snapshot.connectionState == ConnectionState.waiting) {
      return Text('Loading');
    }
    final data = snapshot.requireData;
```

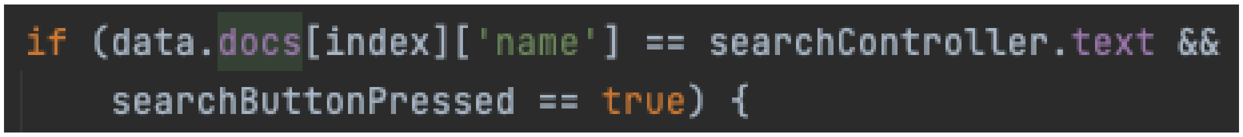
Figure 14: StreamBuilder set-up



```
Text('Classroom: ${data.docs[index]['classroom']}',
    style: TextStyle(fontSize: 20)),  // Text
```

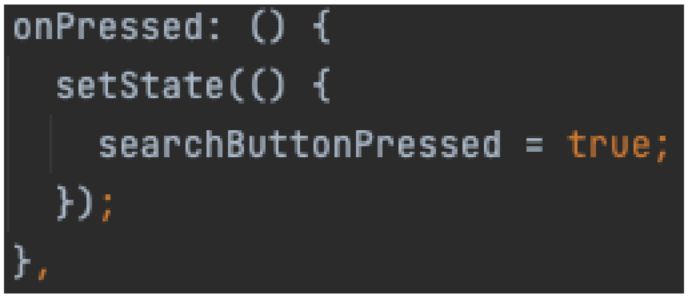
Figure 15: Accessing 'classroom' value in document

To access only the specific document of the intern/student searched for, I accessed the text of TextEditingController searchController and checked its congruity with the current document's value for "name" (itemBuilder loops through collection documents). If the text matches and searchButtonPressed is true, I display data through Text widgets and relegate searchButtonPressed to false. This technique effectively utilizes iteration and indexing to display the correct data and is therefore appropriate for the application.



```
if (data.docs[index]['name'] == searchController.text &&
    searchButtonPressed == true) {
```

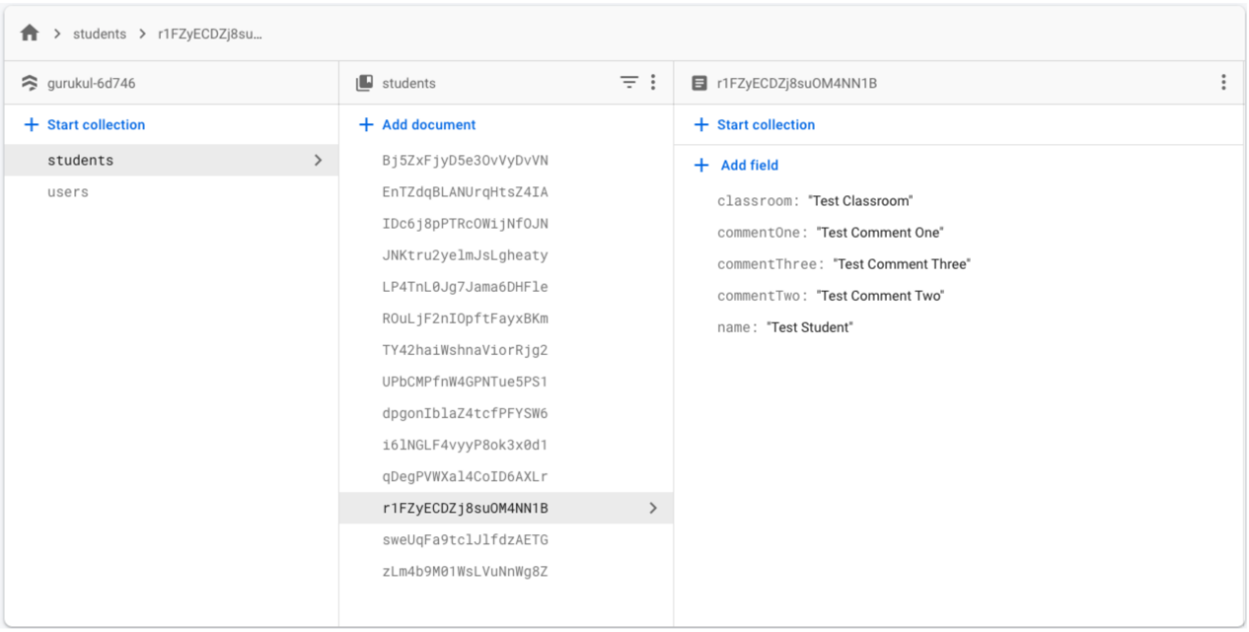
Figure 16: Condition for Data Display



```
onPressed: () {
  setState(() {
    searchButtonPressed = true;
  });
},
```

Figure 17: onPressed() in searchButton()

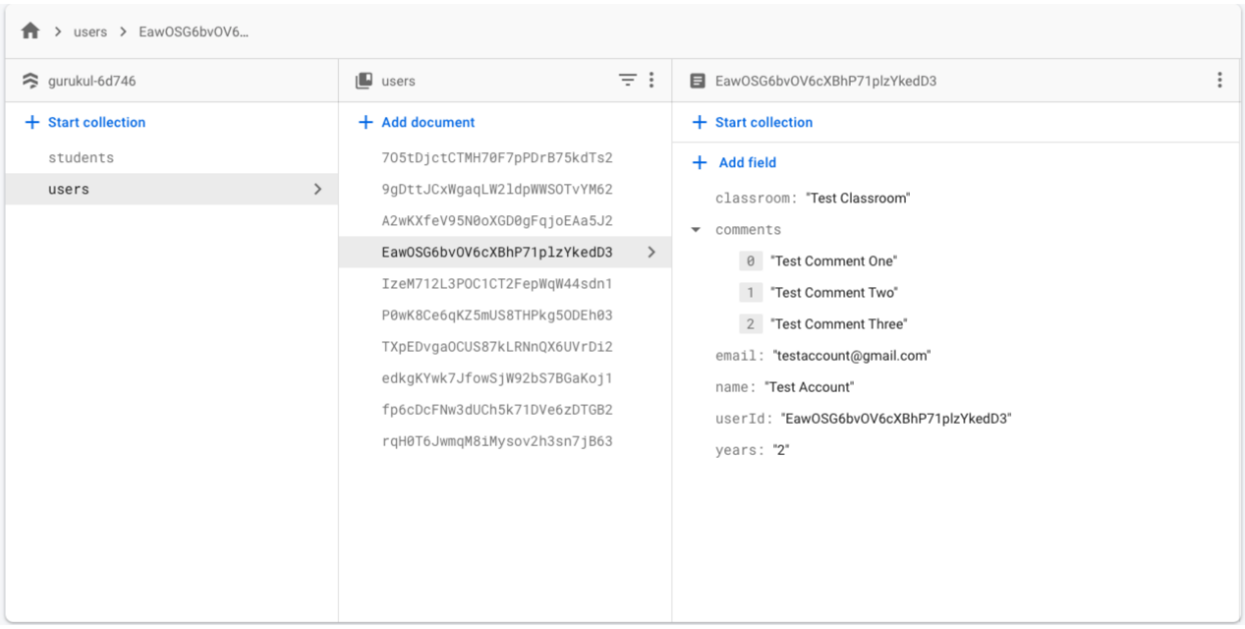
Manipulating Firestore Data:



```
students > r1FZyECDZj8su...

gurukul-6d746          students                     r1FZyECDZj8suOM4NN1B
+ Start collection     + Add document               + Start collection
students >             Bj5ZxFjyD5e3OvVyDvVN         + Add field
users                  EnTZdqBLANUrqHtsZ4IA         classroom: "Test Classroom"
                       IDc6j8pPTRcOWijNfOJN         commentOne: "Test Comment One"
                       JNKtru2yelmJsLgheaty         commentThree: "Test Comment Three"
                       LP4TnL0Jg7Jama6DHFle         commentTwo: "Test Comment Two"
                       ROuLjF2nIOpftFayxBKm         name: "Test Student"
                       TY42haiWshnaViorRjg2
                       UPbCMPfnW4GPNTue5PS1
                       dpgonIblaZ4tcfPFYSW6
                       i6lNGLF4vyyP8ok3x0d1
                       qDegPVWXal4CoID6AXLr
                       r1FZyECDZj8suOM4NN1B >
                       sweUqFa9tclJlfdzAETG
                       zLm4b9M01WsLVuNnWg8Z
```

Figure 18: Sample Student Document

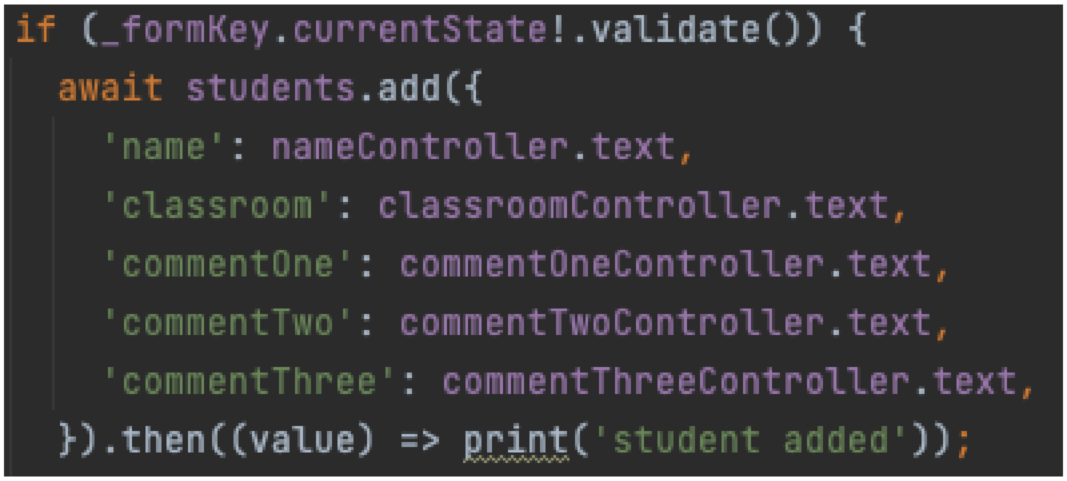


```
users > EawOSG6bvOV6...

gurukul-6d746          users                            EawOSG6bvOV6cXBhP71plzYkedD3
+ Start collection     + Add document                   + Start collection
students               7O5tDjctCTMH70F7pPDrB75kdTs2     + Add field
users >                9gDttJCxWgaqLW2ldpWWSOTvYM62     classroom: "Test Classroom"
                       A2wKXfeV95N0oXGD0gFqjoEAa5J2     comments
                       EawOSG6bvOV6cXBhP71plzYkedD3 >       0  "Test Comment One"
                       IzeM712L3POC1CT2FepWqW44sdn1         1  "Test Comment Two"
                       P0wK8Ce6qKZ5mUS8THPkg5ODEh03         2  "Test Comment Three"
                       TXpEDvgaOCUS87kLRNnQX6UVrDi2     email: "testaccount@gmail.com"
                       edkgKYwk7JfowSjW92bS7BGaKoj1     name: "Test Account"
                       fp6cDcFNw3dUCh5k71DVe6zDTGB2     userId: "EawOSG6bvOV6cXBhP71plzYkedD3"
                       rqH0T6JwmqM8iMysov2h3sn7jB63     years: "2"
```

Figure 19: Sample Intern Document

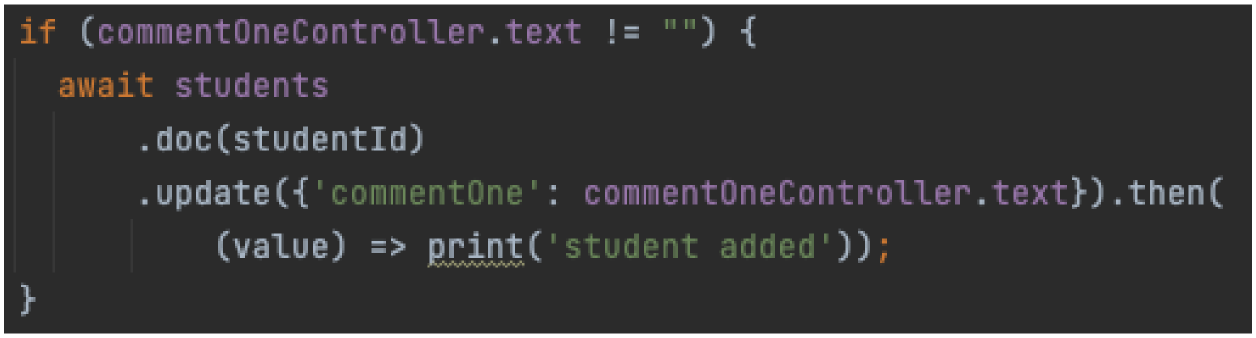
To add a student to Firestore, I utilized TextFields, implementing validator functions for the classroom and name fields to ensure they had values. Once a user clicked "Add Student", a new document was added to Firestore's students collection (variable students) with key-value pairs utilizing text from TextControllers.



```
if (_formKey.currentState!.validate()) {
  await students.add({
    'name': nameController.text,
    'classroom': classroomController.text,
    'commentOne': commentOneController.text,
    'commentTwo': commentTwoController.text,
    'commentThree': commentThreeController.text,
  }).then((value) => print('student added'));
```

Figure 20: Adding Student

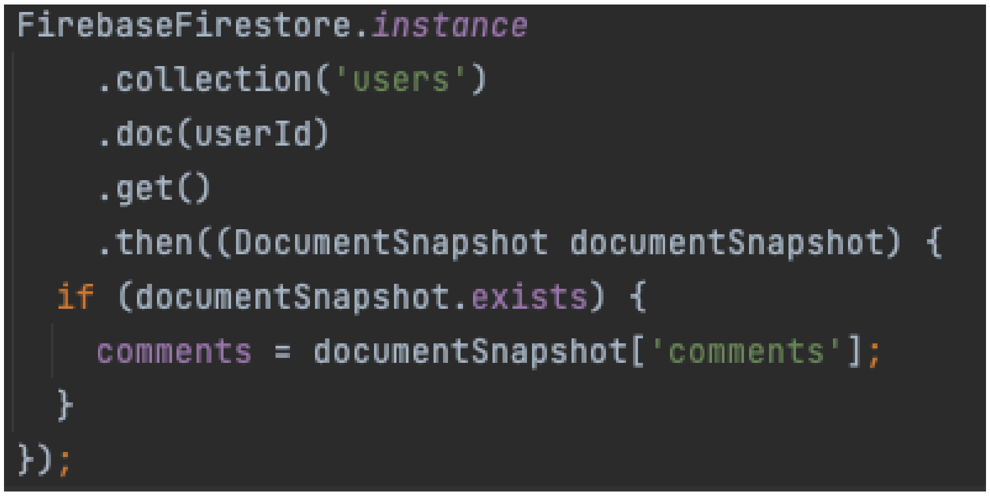
For student/intern editing and student deletion, the page was accessed in a similar manner to the if statement to display searched data. For editing or deleting, the document ID was accessed from the QuerySnapshot data on the respective search screen and passed to the editing/deleting screen (Suragch). Once "Edit Student" is clicked, any fields with values will update the respective key in Firestore (Thakur).



```
if (commentOneController.text != "") {
  await students
      .doc(studentId)
      .update({'commentOne': commentOneController.text}).then(
          (value) => print('student added'));
}
```

Figure 21: Updating Comment One on student's document if not empty

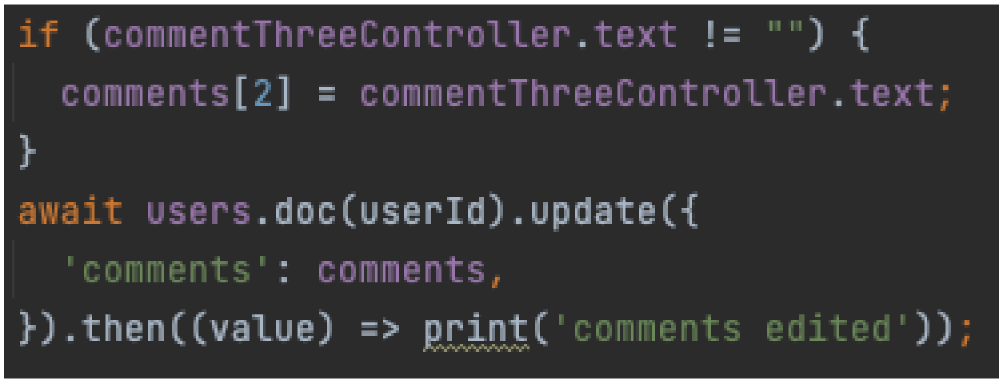
To edit an intern, as the comments were stored in a list, a local variable comments was created with the value of the key "comments" from the user document.



```
FirebaseFirestore.instance
    .collection('users')
    .doc(userId)
    .get()
    .then((DocumentSnapshot documentSnapshot) {
  if (documentSnapshot.exists) {
    comments = documentSnapshot['comments'];
  }
});
```

Figure 22: Pre-existing comments

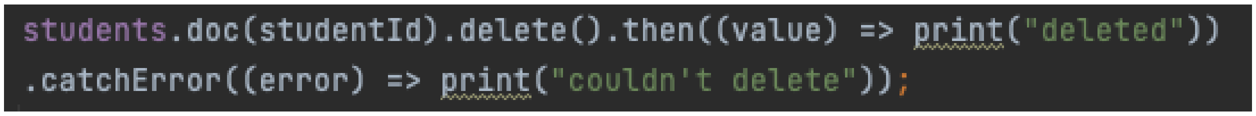
If a comment field had text, its respective index in the list was updated, and then the key "comments" in the intern document was fully updated.



```
if (commentThreeController.text != "") {
  comments[2] = commentThreeController.text;
}
await users.doc(userId).update({
  'comments': comments,
}).then((value) => print('comments edited'));
```

Figure 23: Updating comment three if non-empty and updating comments

To delete a student, the delete functionality was called on the specific student document once they clicked "Confirm Deletion" (FlutterFire). All manipulating functionality was appropriate as it allowed users to access collections or documents from the front-end and only manipulate what they needed to.

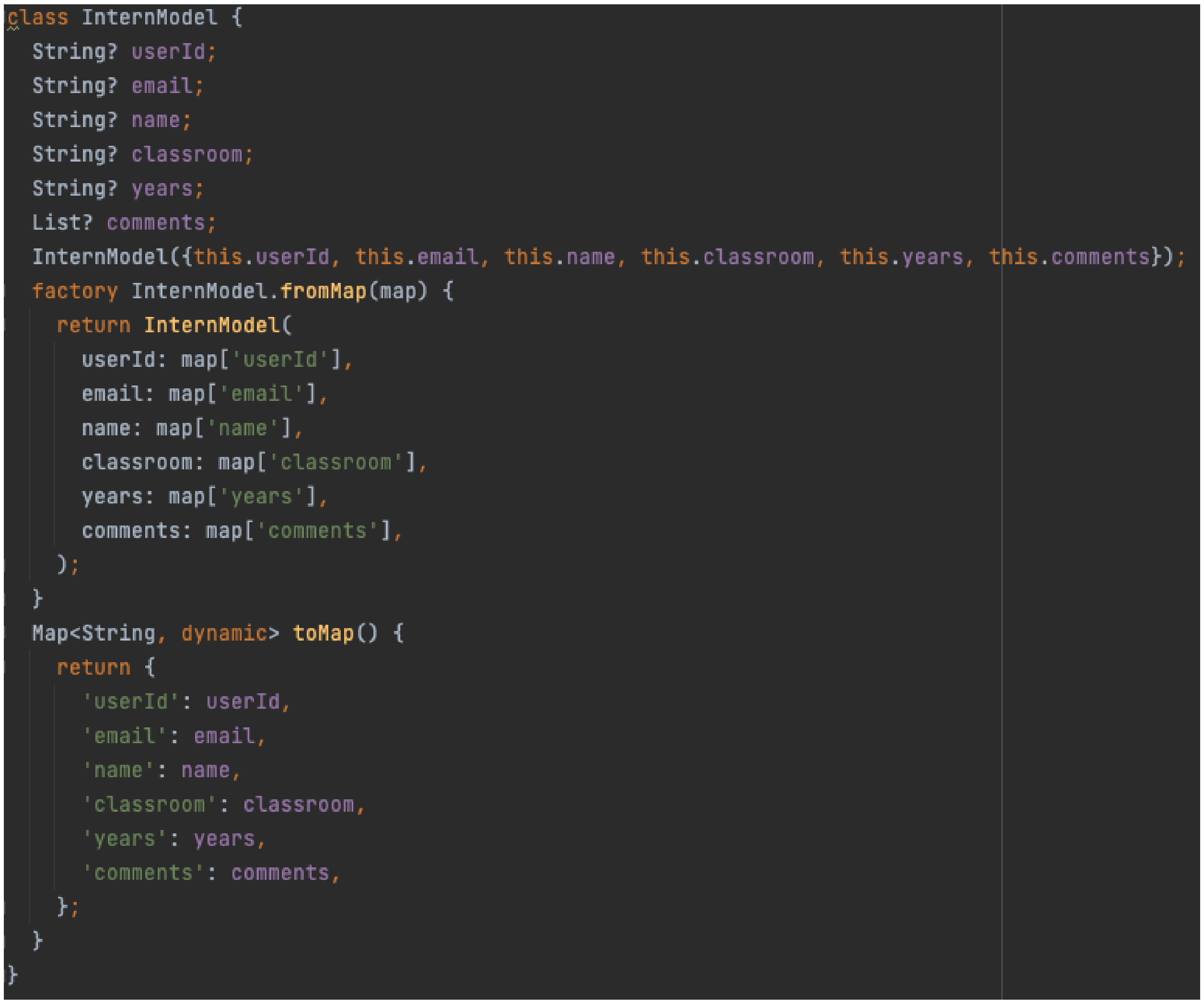


```
students.doc(studentId).delete().then((value) => print("deleted"))
.catchError((error) => print("couldn't delete"));
```

Figure 24: Student Deletion

Authentication and Account Creation:

I mainly adhered to Backslash Flutter's videos for these algorithms. Using Firebase's authentication and database features, a user can create an account and sign in ("Flutter & Firebase Auth | Implementing Cloud Firestore"). I added admin authentication through an extra text field and a hard-coded answer, which allows my application to have much more depth and functionality. I also implemented a more complex InternModel class which acted as a hierarchical record-style data structure, referenced to read intern data for a logged-in intern through its factory methods. This class has a list which acts as the store for all comments on the intern, appropriate as all comments should be linked together.



```dart
class InternModel {
  String? userId;
  String? email;
  String? name;
  String? classroom;
  String? years;
  List? comments;
  InternModel({this.userId, this.email, this.name, this.classroom, this.years, this.comments});
  factory InternModel.fromMap(map) {
    return InternModel(
      userId: map['userId'],
      email: map['email'],
      name: map['name'],
      classroom: map['classroom'],
      years: map['years'],
      comments: map['comments'],
    );
  }
  Map<String, dynamic> toMap() {
    return {
      'userId': userId,
      'email': email,
      'name': name,
      'classroom': classroom,
      'years': years,
      'comments': comments,
    };
  }
}
```

Figure 25: InternModel.dart

Word Count: 1071