

Criterion C: Development

Techniques

1. API Integration and Searching
2. Abstract Data Type: Map
3. Constants and Reusable UI
4. Recursion

Technique #1

One important technique that I made use of was searching and implementing multiple APIs. My client Mr. Howard had requested that I create an application that helps him with shopping at online retail stores; finding the best deals and comparing them across multiple retailers (See Appendix B). For this reason, I decided to make use of two separate REST APIs, Rainforest API and the BlueCart API. One provided realtime data on Amazon shopping prices while the other provided information about Walmart's online marketplace, respectively. The data was primarily stored as a multidimensional JSON file.

```
List<String> parseInput(String input) {
    return input.split(" ");
}

//Append Search is a method that contains the API url with the key. This appends
//the API key and the search term which is the user input together.
String appendAmazonSearch(List<String> searchInput) {
    String appendedURL = constants.UserStack.AmazonProductAPI + "&search_term=";
    appendedURL = "$appendedURL${searchInput[0]}";

    if (searchInput.length > 1) {
        for (int i = 1; i < searchInput.length; i++) {
            appendedURL = "$appendedURL+${searchInput[i]}";
        }
    }
    appendedURL = "$appendedURL&sort_by=price_low_to_high";
    print(appendedURL); // remove this
    return appendedURL;
}
```

To retrieve the data from these APIs I made use of http calls. But in order to get the data based on the user input, I first parsed the user input, splitting each word into it's own String. From there, I took the API key and appended the search parameters that the API provided followed by the search term. Since the API delimited the URL using the & symbol, I had to traverse through the searchInput array returned by the parseInput method and then append that and the & symbol to the pre-existing URL.

```

void fetchFromWalmartApi(Uri url) async {
  setState(() {
    MainMenu._isLoadingData = true;
  });
  var response = await http.get(url);
  String data = response.body;
  MainMenu.json2.addAll(jsonDecode(data));
  setState(() {
    Product.getWalmartProductsList(MainMenu.json2);
    MainMenu._isLoadingData = false;
  });
}

```

```

onPressed: () {
  String url = appendAmazonSearch(parseInput(searchMenu.text));
  String url2 = appendWalmartSearch(parseInput(searchMenu.text));

  Product.currentProductList.clear();

  fetchFromAmazonApi(Uri.parse(url));
  fetchFromWalmartApi(Uri.parse(url2));
},

```

I created two fetchFromAPI methods.

How it works:

Firstly, an HTTP request is made provided the appended URL. A response is returned which is then turned into a JSON String. From there I used the jsonDecode method provided by the dart:convert library to get the JSON structure. I referenced the HTTP docs provided by the dart team (See Appendix A).

When data is fetched from the API, it is converted from a Map<String, dynamic> to a list of products using the following method:

```

static getAmazonProductsList(Map<String, dynamic> json) async {
  Product temp;
  //Forloop which traverses through the JSON object search_results
  List searchResult = json['search_results'];
  currentProductList.clear();

  //populates a list with all the products retrieved from the search
  for (int i = 0; i < searchResult.length; i++) {
    temp = Product(
      "Amazon",
      getLinkFromMap(searchResult[i]),
      getPriceFromMap(searchResult[i]),
      getImageFromMap(searchResult[i]),
    );
    // Prevents irrelevant results from being displayed
    if (double.parse(temp.rawPrice) > 4.00) {
      currentProductList.add(temp);
    }
  }
  productsOrderTraversal(currentProductList.length, currentProductList);
}

```

The `getAmazonProductsList` is a method in the `Product` class. This method is responsible for storing the information regarding each product listing as an object. Since `PriceMatch` is a tool to find the best online deals, having relevant information such as the price and website link are important to have. Thusly in this method, I searched the JSON structure by first converting it to a `Map<String, dynamic>` and then storing all the products stored under the `search_results` key.

Technique #2

One important technique that I employed was **creating my own ADT, which is comprised of a list of nodes**. When I was tasked with creating custom ADTs for complexity, I recalled what I had learned while I was learning about Nodes and binary search trees. I created this abstract data type to future proof. My plan was originally to create a `priceWatchList` for each user which would store the items that a user desires to receive alerts for when they reach an affordable price. But unfortunately due to time constraints and many issues with API integration that was not ready. However, with this `Map` class and connection to Google's NoSQL database `Firebase`, in the future I will be able to create the functionality outlined in success criteria 4 and 5.

```
import 'node.dart';

class Map<K, V> {
  late List<Node> _bucket;
  late int length;
```

I implemented my own map and made use of the Dart language's support for "Templates".

This enabled me to create a `Map` class that I can use for multiple different class variables.

```
class Node<K, V> {
  late V _value;
  late K _key;

  Node(K key, V value) {
    this._key = key;
    this._value = value;
  }

  void setNode(K key, V value) {
    this._value = value;
  }
}
```

In order to implement my own map, I created my own `Node` class. When creating this class, I referenced the PowerPoint lecture that we had on Binary Search Trees to come up with this solution. Having a bucket of nodes to represent the `HashMap` was appropriate because in the future I can easily traverse through the nodes.

```

void replace(K key, V value) {
    int i = 0;

    while (_bucket.elementAt(i) != key) {
        i++;
        if (i > length) {
            return;
        }
    }
    _bucket.elementAt(i).setNode(key, value);
}

```

The replace method I created for the Map class looks at whether a key-value pair already exists. Here I used a while loop to traverse through the bucket of Nodes. This sentinel check looks to see whether a key is already existing, and if it doesn't exist, it will create a new node with that assigned key value pair and add it to the bucket of nodes.

My intent with the Map class was to create the PriceWatchList that every single User would have. Thus, I made a user object hold said information. When registering and logging in, I referenced the userStack and created a new User for each newly registered user.

```

class User {
    late String _name;
    late String _email;
    late Map<String, Double> _PriceWatchList;

    User(String name, String email) {
        _name = name;
        _email = email;
        _PriceWatchList = Map<String, Double>();
    }
}

```

Multiple users are likely to use the application I create. Also, I added the variable PriceWatchList which is a Map with the Key-Value pair being String and double value. When a user adds an item to a price watch list, the key will represent the item name and the value will be the expected price the user is currently looking out for.

The class pictured is responsible for communicating with Firebase Auth and the app's main security functions. In this class I also created the dialogue box which handled the errors.

```

class Auth {
  final FirebaseAuth authService = FirebaseAuth.instance;

  late BuildContext context;
  //Registers User in both Firebase and locally.
  Future registerUser(
    email, password, String firstName, String lastName) async {
    await authService.createUserWithEmailAndPassword(
      email: email, password: password);
    userClass.User tempUser =
      userClass.User(firstName, lastName, authService.currentUser!.uid);
    constants.UserStack.user = tempUser;
  }

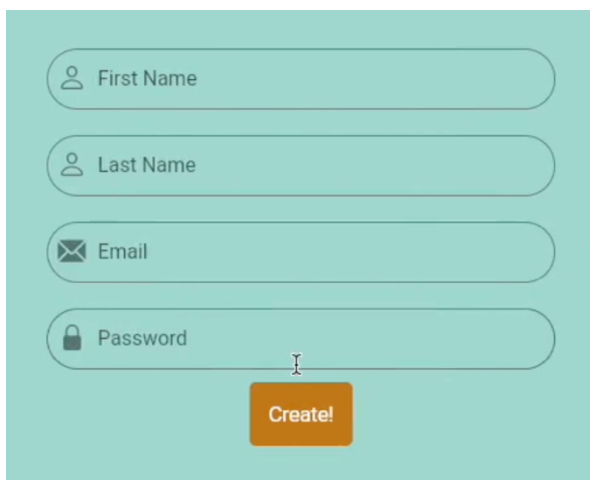
  //Method that is responsible for verifying User Logins.
  Future loginUser(String email, String password, BuildContext context) async {
    try {
      await authService.signInWithEmailAndPassword(
        email: email, password: password);
      // ignore: use_build_context_synchronously
      Navigator.pushNamed(context, '/mainMenu');
    } on FirebaseAuthException catch (e) {
      if (e.code == 'user-not-found') {
        return _showErrorDialog(
          context: context,
          titleText: "Error",
          messageText: 'Whoops user doesn\'t exist');
      }
      if (email.isEmpty || password.isEmpty) {
        return _showErrorDialog(
          context: context,
          titleText: "Error",
          messageText: "Please enter credentials");
      }
      if (!email.contains('@')) {
        return _showErrorDialog(
          context: context,
          titleText: "Error",
          messageText: "Please enter an email");
      } else if (e.code == 'wrong-password') {
        return _showErrorDialog(
          context: context,
          titleText: "Error",
          messageText: 'Incorrect Password. Please try again.');
```

Technique #3

The third technique I employed was making use of global constants. To have appropriate procedural decomposition and maintain clean coding conventions, the most common functions that I needed to call when creating flutter UI elements were instead put in the constants class.

```
class customTextFormField extends StatelessWidget {  
  final TextEditingController controller;  
  final String hintText;  
  final String errorMessage;  
  final IconData name;  
  
  customTextFormField({  
    required this.controller,  
    required this.hintText,  
    required this.errorMessage,  
    required this.name,  
  });  
}
```

Since Flutter makes use of stateful and stateless widgets that have a wide variety of parameters, I created my own customTextFormField that will enable me to create text fields with ease. Instead of having to write lots of code, I can simply put in the parameters of what I want. This is what I used when I created my login page.



```
class MyApp extends StatelessWidget {  
  const MyApp({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      initialRoute: '/',  
      routes: {  
        // When navigating to the "/" route, build the FirstScreen widget.  
        '/': (context) => Login(),  
        // When navigating to the "/second" route, build the SecondScreen widget.  
        '/signup': (context) => SignUp(),  
        '/mainMenu': (context) => const MainMenu(),  
      },  
    ); // MaterialApp  
  }  
}
```

The PriceMatch application has multiple UI screens to navigate. To make it easier to route to different pages, I added my main application class to my global constants file, making it accessible and eliminating redundancy of creating new routing methods.

```
class UserStack {
    static late User user;
    static String AmazonProductAPI =
        "https://api.rainforestapi.com/
    static String WalmartProductAPI =
        "https://api.bluecartapi.com/re
}
```

Likewise, I made use of the UserStack class to store the current User and for referring to the product APIs. By making these variable static, I avoided having to pass “values” through the UI screens and maintained an organized development space.

Technique #4

My application’s main function is to make online shopping easier for customers. Hence, having a UI that is both seamless and informs the user about things like price are vital. The success criteria outlined that the cheapest product listings would be displayed at the top of the listview, therefore I made use of the bubble sort as it is the simplest algorithm that works by swapping adjacent elements multiple times (GeeksForGeeks.org).

```
//Bubble sort recursive method
//This method was inspired by the presentation on recursion and also GeeksForGeeks'
// iterative bubble sort method
static void productsOrderTraversal(int n, List<Product> list) {
    //base case
    if (n == 1) {
        return;
    }
    int count = 0;
    for (int i = 0; i < n - 1; i++) {
        if (double.parse(list[i].rawPrice) > double.parse(list[i + 1].rawPrice)) {
            //when the product at index position i is greater than the next position's price,
            //the two products swap indices.
            Product temp = list[i];
            list[i] = list[i + 1];
            list[i + 1] = temp;
            count = count + 1;
        }
    }
    if (count == 0) {
        return;
    }
    //recursive call
    productsOrderTraversal(n, list);
}
```

Since the data provided by the JSON was Map<String, dynamic>, the rawPrice was a String value. Hence I made use of the double.parse() method when comparing current and next element in the currentProductList list.

Word Count: 1088 Words