GROUP 10 SUMMATIVE PROJECT 9JACLEAN MOBILE APPLICATION IN FLUTTER

SOFTWARE ENGINEERING GROUP 10 SUMMATIVE
AFRICAN LEADERSHIP UNIVERSITY KIGALI, RWANDA.

NAME OF FACILITATOR
SAMIRATU NTHOSI

March, 2025

GROUP ACTIVITIES

DEMO VIDEO LINK:

https://youtu.be/lNI8TIl0it8?feature=shared

GITHUB LINK: https://github.com/Afsaumutoniwase/jaclean.git

S/N	GROUP MEMBERS	ROLE	ATTENDANCE	CONTRIBUTION
1	Lydia Ojoawo	Member	Feb 1-March 30, 2025	Coordinated team meetings and scheduling, worked on documentation, implemented the Market tab and Review tab functionality.
2	Simeon Azeh Kongnyuy	Member	Feb 1-March 30, 2025	Implemented the Authentication page and handled Firebase backend connection for login, registration, and user verification
3	Omar Keita	Member	Feb 1-March 30, 2025	Focused on app testing (unit/widget tests), worked on the Profile tab, and prepared the **slide presentation** for final delivery.
4	Afsa Umutoniwase	Member	Feb 1-March 30, 2025	Designed the Home Page interface and was responsible for setting up the GitHub repository and collaboration structure.
5	Nickitta Umuganwa Asimwe	Member	Feb 1-March 30, 2025	Designed the Service tab interface and worked on documentation

ABSTRACT

The aim of the thesis was to develop a mobile application called 9jaclean, built using the Flutter framework developed by Google. Flutter is a cross-platform development toolkit that enables the creation of applications that run on Android, iOS, Web, and Desktop using a single codebase.

The tools and methodologies applied in this project include Flutter, Google Maps API, Firebase, Agile methodology for project management, and the BLoC (Business Logic Component) pattern for efficient state management within the application.

9jaclean is designed to tackle Nigeria's waste management challenges by offering a digital platform that facilitates responsible waste disposal, recycling, second-hand trading, and donations. The application enables users to buy, sell, donate, and recycle items, as well as schedule waste pickups all through a userfriendly mobile interface. This solution encourages sustainable practices while also promoting economic empowerment through a circular economy model.

CONTENTS

LIST OF FIGURES

INTRODUCTION

1.1 Objectives

The objective of this project is to develop 9jaclean, a mobile application designed to improve waste management and promote sustainability in Nigeria. The app will provide users with a platform to recycle, trade second-hand goods, donate items, and schedule waste pickups, making waste disposal more efficient and environmentally responsible.

To achieve this, the project focuses on creating a cross-platform mobile application using Flutter, ensuring accessibility for both Android and iOS users. Firebase Authentication and Cloud Firestore will be implemented to manage user accounts securely and store real-time data for transactions and waste collection services. Google Maps API will enhance the user experience by helping individuals locate recycling centers and schedule waste pickups based on their location.

Beyond waste management, 9jaclean will serve as a digital marketplace where users can buy, sell, and donate second-hand items, reducing landfill waste while benefiting individuals and communities. The app will also promote environmental awareness by offering educational resources, incentives for responsible disposal, and collaborations with sustainability-focused organizations.

At the end of the project, 9jaclean is expected to provide a practical, scalable, and user-friendly solution to Nigeria's waste management challenges. The long-term goal is to launch the app on major digital platforms, ensuring widespread adoption and ongoing improvements based on user feedback and technological advancements.

1.2 Contributions

Waste management is a major challenge in Nigeria, with millions of tonnes of waste improperly disposed of each year. Many communities lack access to structured waste collection and recycling services, leading to environmental pollution and public health risks. 9jaclean was developed to address this issue by providing a digital platform where individuals can conveniently recycle, trade second-hand goods, donate usable items, and schedule waste pickups.

This app contributes to society by making waste management more accessible and organized. It allows users to find nearby recycling centers, request waste collection services, and participate in a marketplace that promotes the reuse of second-hand goods. By doing so, 9jaclean helps reduce landfill waste, encourage responsible consumption, and support low-income communities through item donations.

Beyond environmental benefits, the project also creates economic opportunities. Independent waste collectors, small businesses, and local recycling centers can use the platform to connect with customers, increasing their income while contributing to sustainability efforts. Additionally, the app raises awareness about responsible waste disposal by providing educational content and incentives for participation.

APPLICATION DESIGN

2.1 Application Features

The 9jaclean mobile application is a multifunctional platform developed to address Nigeria’s waste management challenges through digital innovation. It empowers users to take actionable steps toward sustainability by offering services such as recycling, second-hand trading, and donations, all within a single mobile interface.

To ensure secure access and personalized user experiences, the app includes a complete authentication system**.** New users can easily register an account, and returning users can log in with their email and password. After registration, users are prompted to verify their email addresses to ensure account authenticity. The app also offers a password reset feature, allowing users to recover access quickly through a secure email-based process.

Once authenticated, users arrive at a personalized home dashboard that displays contributions such as the total kilograms of waste recycled and estimated CO_2 emissions saved giving them immediate insight into their environmental impact. From here, users can navigate to core functionalities using an intuitive bottom navigation bar.

The Services section enables users to schedule waste pickups, donate reusable items, and discover nearby certified recycling centers. Users can initiate these services with minimal input, ensuring convenience and promoting consistent engagement with the app’s sustainability goals.

Through the Marketplace, users can buy, sell, or donate second-hand items such as electronics, clothing, and household goods. This feature encourages circular economy practices while providing economic opportunities for users.

The Wallet allows users to track their earnings from recycled goods or marketplace sales, manage withdrawal methods, and view their transaction history. This financial feature reinforces the app’s goal of combining sustainability with economic empowerment.

In the Profile Management area, users can edit personal information, manage linked bank accounts, activate biometric login (Face or Touch ID), and change their password securely.

To maintain service quality and trust, the Review System allows users to rate services and products, write reviews, and view others' feedback. This fosters community engagement and accountability.

All features are accessible with a few taps and designed to work seamlessly on both Android and iOS devices. The app's interface supports smooth user experiences and ensures accessibility regardless of technical background. The primary use cases for the 9jaclean mobile application is shown in Figure 1 below.

Figure 1 shows the main ways users can interact with the 9jaclean mobile app. Each action shown in the diagram represents a useful feature that helps users take part in eco-friendly activities. These features include viewing their past recycling contributions, scheduling pickups for recyclable materials, donating items, buying or selling secondhand products, finding nearby recycling centers, managing their personal profile and wallet, and rating or reviewing services they've used. The app is built mainly for regular users, making it easy for anyone to use the features without needing any special permissions. Users can simply open the app and start exploring helpful tools to support their recycling journey. However, to keep things safe and organized, some parts of the app like adding articles, creating content, or making changes to public information are only managed by admins. This ensures that the app stays trustworthy and free of false, offensive, or misleading information. Admins are the only ones who can create, edit, or remove articles and other sensitive content.

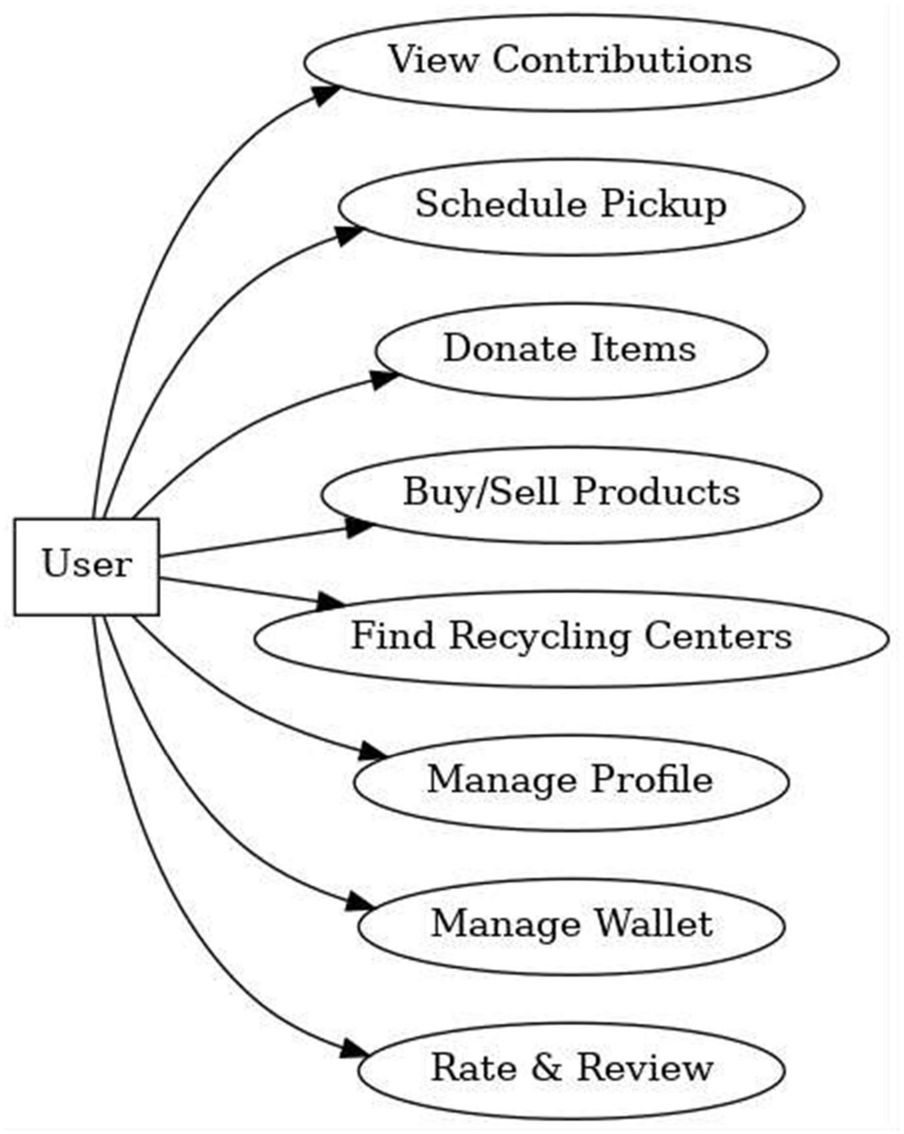


Figure 1: Use Cases in the 9jaclean App

Users do not need to worry about security or data accuracy because the app is designed to protect privacy and promote helpful, real information. In case a user notices anything wrong or has suggestions, they can contact the admin easily using the Contact Screen in the app. This gives everyone a chance to help improve the app while still keeping it safe for everyone.

Another useful feature is the ability to "star" or save articles, so users don’t have to search for them again later. The app is made to be user-friendly, with simple navigation, so that every task from booking a pickup to checking your wallet is just a few taps away. Figure 2 shown below presents the operational logic of the 9jaclean application based on user conditions and navigation steps.

Figure 2 shows the step-by-step flow of how users move through the 9jaclean mobile application. The process begins when the user launches the app. If it is their first time using it, they are taken through an onboarding process. This onboarding helps new users learn about the app, its purpose, and how to use key features like donating items or scheduling waste pickups. If the user is not new, they are sent directly to the home dashboard.

From there, users can choose to log in or register for an account. If a user has forgotten their password, the system gives them the option to reset it. After successfully logging in, the user has full access to the app's main features. These include scheduling pickups for recyclable items, donating or selling goods, finding recycling centers nearby, managing their wallet and payments, editing their personal profile, and giving ratings or reviews for services they’ve used. The flowchart also includes the logout process. After using the app, users can decide whether to stay logged in or log out. If they log out, the app safely ends the session and takes them back to the home screen.

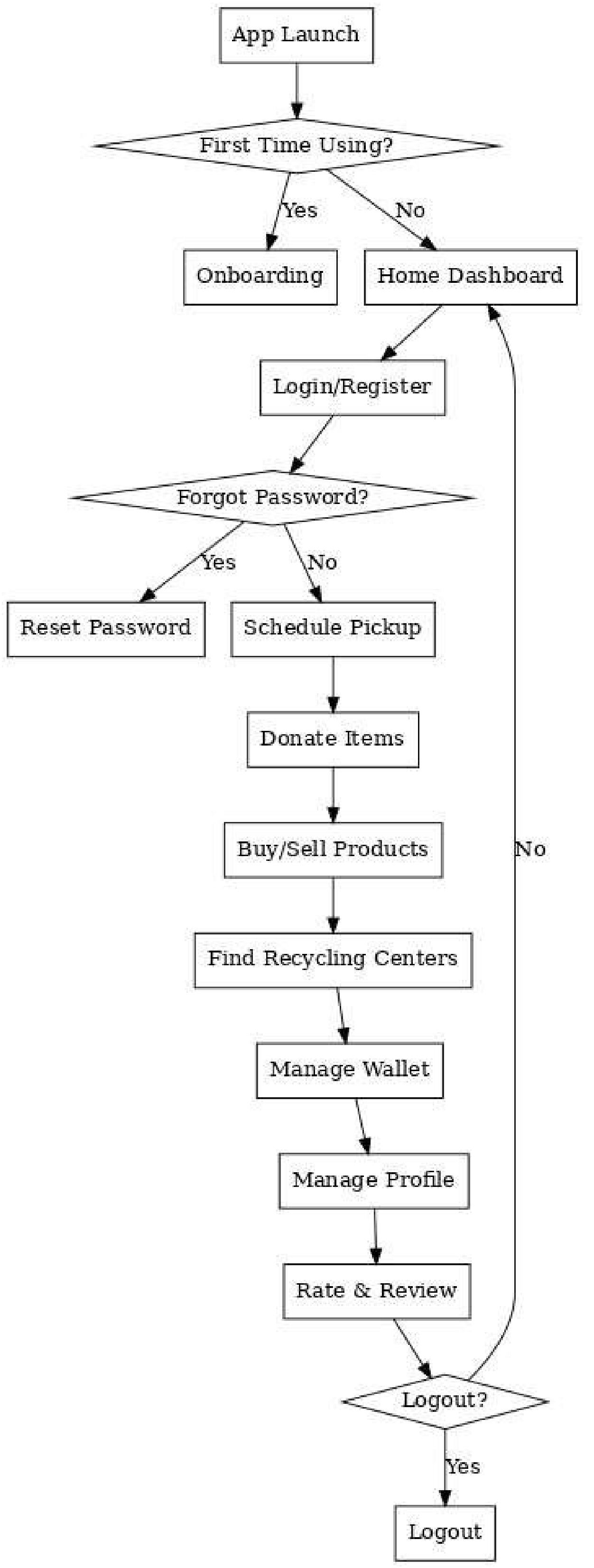


Figure 2: Application Flowchart

This flowchart focuses on user actions and keeps the system easy to use. Every feature is made to be accessible with just a few taps, making sure that even users without technical skills can navigate the app smoothly. Admin actions such as managing articles or handling reports are not shown here, since this version of the app is mainly focused on regular users.

2.2 Overview of UI

The 9jaclean mobile application features a clean, intuitive, and highly user-centered interface designed to improve user engagement and streamline access to its core services. As shown in Figure 3, the app enables convenient interactions such as scheduling waste pickups, locating nearby recycling centers, donating reusable items, and participating in a marketplace for second-hand goods.

Figure 3 below, presents actual screenshots taken from the working 9jaclean prototype, showcasing the user onboarding process, core dashboard, service options, marketplace, profile, wallet, and review functionalities. These visuals demonstrate the natural flow of the application and reflect its current production-ready state. The UI was implemented directly within the application using Flutter, with a focus on delivering a smooth, responsive experience. A fixed bottom navigation bar allows users to effortlessly switch between the main sections of the app: Home**,** Services**,** Marketplace**,** Profile**,** and Reviews**.**

The Home screen serves as a dashboard, highlighting the user's contribution. The Services section provides access to essential features like scheduling pickups, donating items, and locating certified recycling centers using integrated map support. The Marketplace enables users to browse, list, and purchase second-hand items or donate to charity. The Profile tab includes personal information management, and wallet. The Review section allow user to review the products.

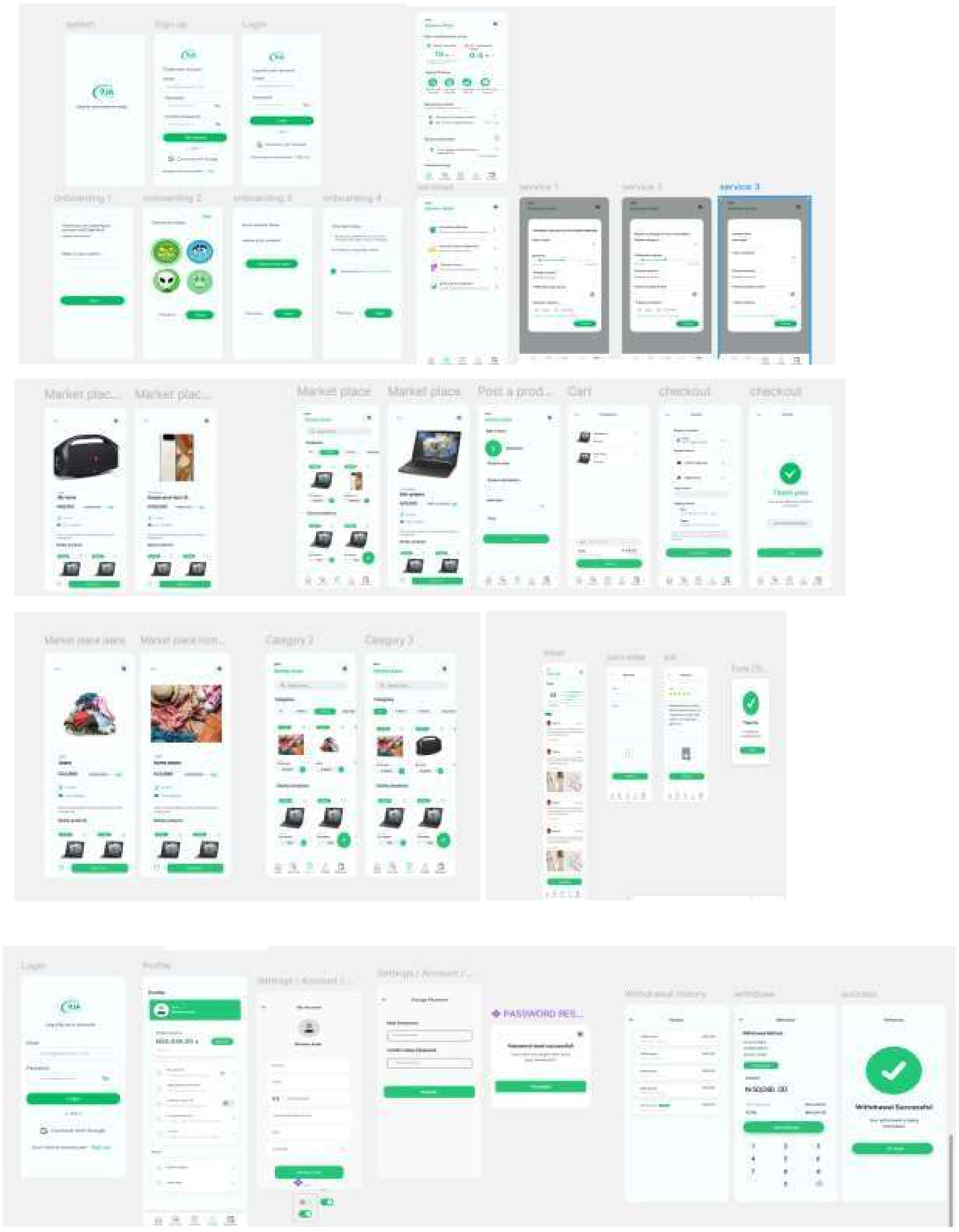


Figure 3. UI Screens of 9jaclean Application

RELEVANT TECHNOLOGIES

3.1 Flutter Framework

Flutter is an open-source UI toolkit developed by Google for building high-performance applications for Android, iOS, web, and desktop all from a single codebase. It uses the Dart programming language and provides a rich collection of customizable widgets that make it possible to build smooth, modern, and responsive interfaces (Flutter Documentation, 2023).

A key feature that significantly enhanced our development process for the 9jaclean app was Flutter's hot reload capability. This allows developers to instantly view changes in the application as the code is updated, enabling faster debugging and real-time user interface iteration (Martin, 2021). It makes it easy to experiment, build UIs, and fix bugs with a much quicker feedback loop.

Flutter supports both Material (Android) and Cupertino (iOS) design principles, ensuring native-like performance and appearance on different devices. For the 9jaclean project, we built core screens including the home dashboard, marketplace, profile management, and transaction views using reusable Flutter widgets. Community support around Flutter is extensive, with many packages and documentation available. This helped streamline integrations with Firebase and Google Maps.

In addition, Flutter's support for structured architectures like BLoC (Business Logic Component) played a major role in helping us manage the app's state and separate UI logic from business logic (Sannino, 2022). This made the codebase cleaner, easier to maintain, and more testable over time. Community support around Flutter is vast and growing. Numerous packages, plugins, and online resources are available that ease integration with tools like Firebase, which we used for user authentication and cloud services, and Google Maps, which powers the location-based features in our app. This strong ecosystem accelerated our development and allowed us to implement complex features with minimal overhead (Martin, 2021).

3.2 BLoC Design Pattern

To keep the code organized and manageable, using an architectural pattern is very important in Flutter development. One of the most popular and effective patterns for this purpose is BLoC, which stands for

Business Logic Component. This design pattern helps to separate the user interface (UI) from the business logic, making the application more structured, testable, and easy to maintain.

An alternative to full BLoC is Cubit, a simplified version of the pattern. While BLoC uses both events and states, Cubit only deals with states. This makes it easier to implement when the logic is not very complex. In our application, Cubit was used in smaller parts of the project where only direct state changes were required, helping reduce unnecessary code and improve performance (Flutter Documentation, 2023).

In the 9jaclean application, both Bloc and Cubit approaches were implemented to handle different parts of the app's logic. The *AuthBloc* was responsible for managing user authentication processes, including login, registration, and verifying active user sessions. For UI customization, *ThemeCubit* enabled dynamic switching between light and dark modes. Additionally, the business logic related to recycling operations was handled by *PickupCubit*, which managed waste pickup requests, while *MarketplaceCubit* controlled the logic behind listing, buying, and selling items in the marketplace section of the app.

By using BLoC and Cubit, we were able to build a clean, scalable structure that made the app more efficient and easier to work on as a team. Figure 4 shows, BLoC and Cubit have the same responsibility which is emitting a new state to change the UI once a response is received or a request is sent.

As shown in Figure 4, both BLoC and Cubit are used to manage app state and update the user interface based on new data or actions. The main difference between them is how they are triggered. Cubit responds when a function is called, while BLoC responds when an event is added.

In the 9jaclean project, we used BLoC to manage the application's theme and authentication state throughout the app's lifecycle. To implement the BLoC pattern, we first added the required dependencies equatable, bloc, and flutter_bloc in the pubspec.yaml file. With these packages, we were able to easily create the main BLoC classes: AuthBloc, AuthEvent, and AuthState, helping to keep our code organized and scalable.

On the other hand, Cubit is more lightweight and requires only a function call to emit a state. For this reason, we used Cubit to manage simpler features like the app's theme mode. This only needed two classes: ThemeCubit and ThemeState.

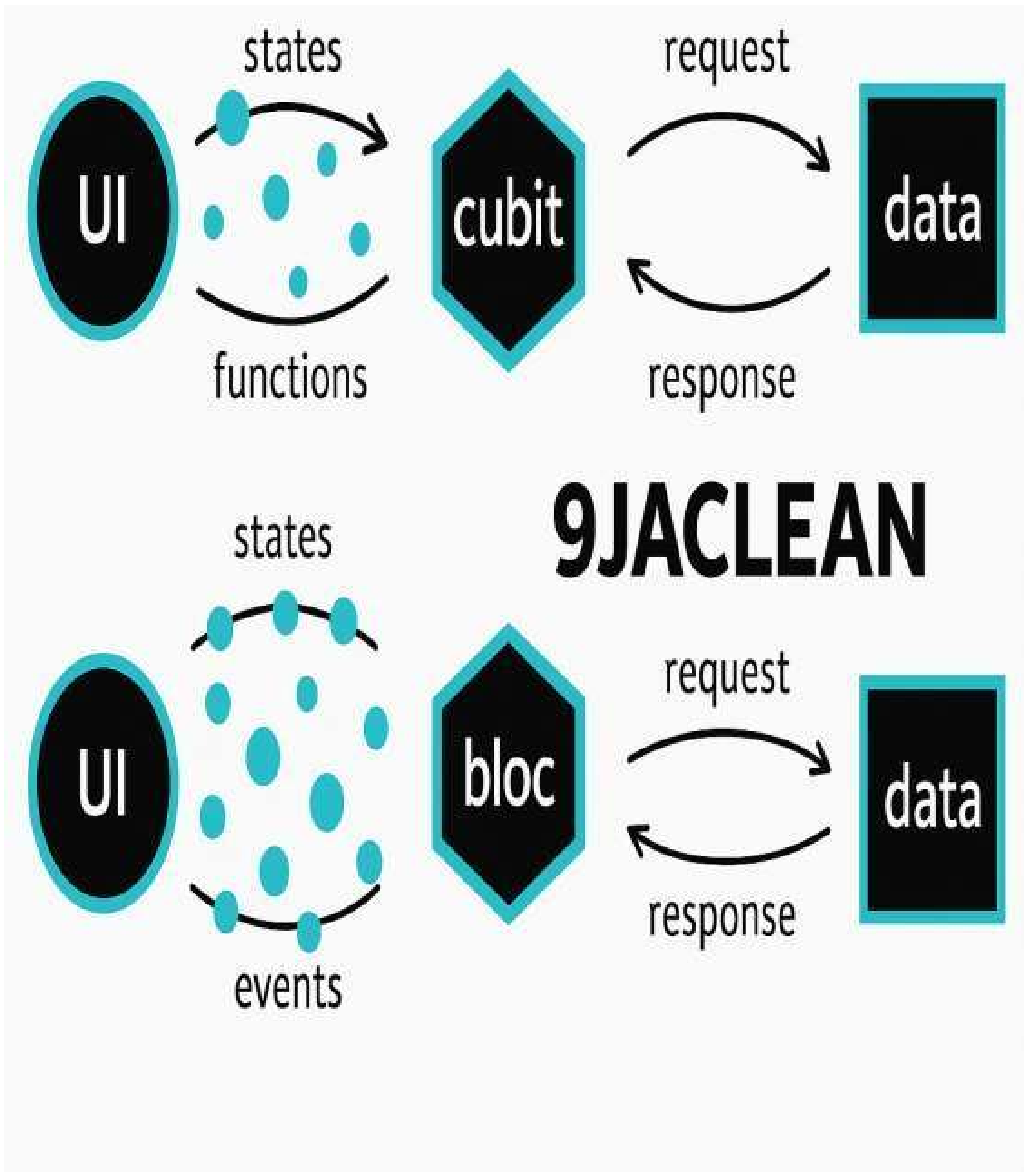


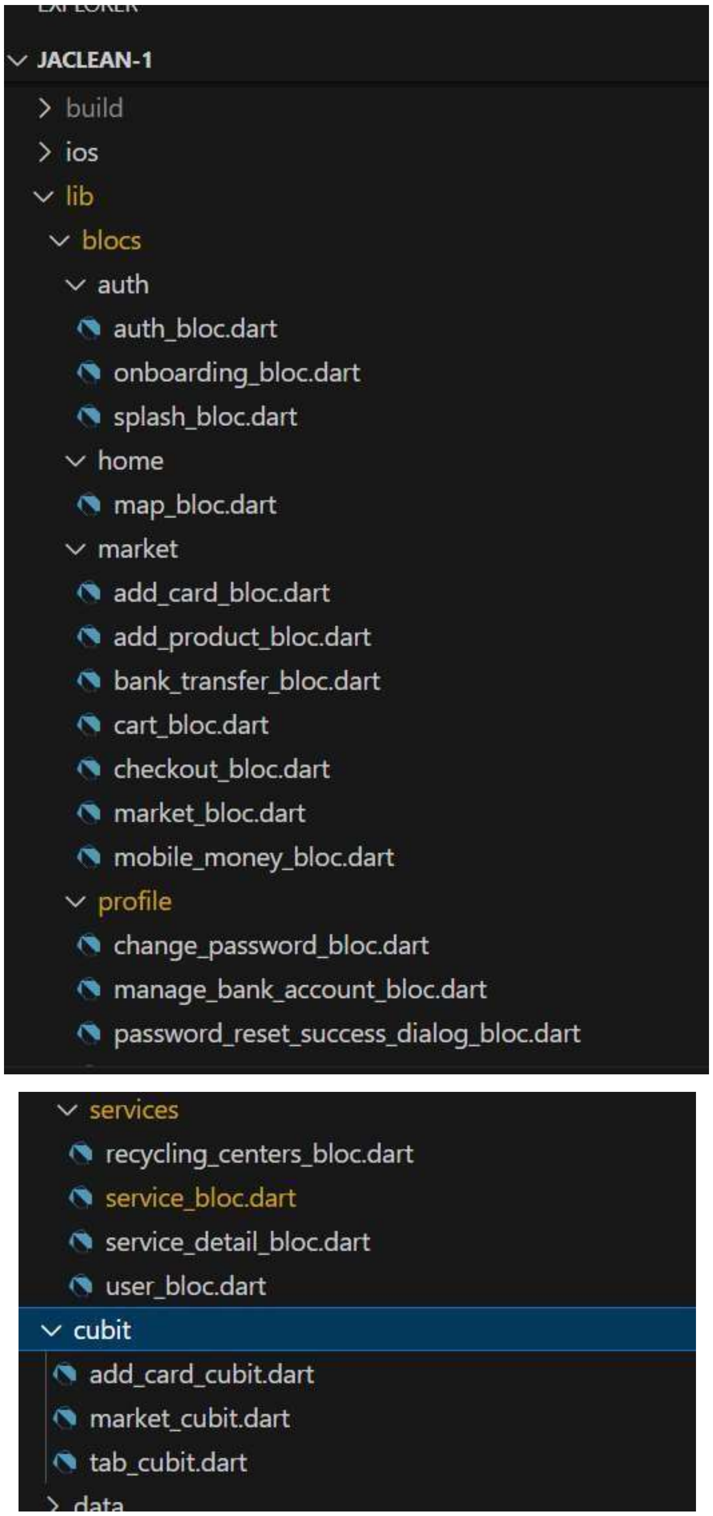
Figure 4. Illustration of how BLoC and Cubit work

Figure 5 illustrates the file organization used to implement state management in the 9jaclean application using both Cubit and BLoC patterns. Cubit was used in simpler cases to manage lightweight state transitions for example, managing tab navigation, market filters, or payment card form logic. These are handled in files like tab_cubit.dart, market_cubit.dart, and add_card_cubit.dart, which reside under the services/cubit directory.

For more complex features such as authentication, onboarding, and marketplace transactions, the BLoC pattern was applied. Unlike traditional BLoC structures that separate logic into three distinct files (bloc, event, and state), 9jaclean combines all three components within a single file for each feature. For instance, in auth_bloc.dart, the BLoC class, its associated events (like login or password reset), and its various states (such as loading, success, or error) are defined together in one file. This approach keeps the code compact and easier to navigate for each domain-specific logic.

The MultiBlocProvider widget is used in the main.dart file to provide multiple BLoC and Cubit instances throughout the app's widget tree. This approach is necessary because the 9jaclean application uses a wide range of BLoCs and Cubits to manage different functionalities such as authentication, cart handling, market operations, password changes, and more. Rather than wrapping each widget separately with its own BlocProvider, the MultiBlocProvider ensures all these blocs are globally available in one place, improving code organization and scalability.

As seen in the main.dart file, each BLoC (like AuthBloc, CartBloc, MarketBloc) and Cubit (like MarketCubit, TabCubit) is created using the BlocProvider(create: (context) => ...) syntax and added as a provider in the list. This enables any part of the app to listen to state changes and react accordingly. For example, when accessing a Cubit's current state such as the selected tab or theme mode, it can be retrieved using BlocProvider.of<TabCubit>(context).state.



```
EXPLORER
∨ JACLEAN-1
  > build
  > ios
  ∨ lib
    ∨ blocs
      ∨ auth
          auth_bloc.dart
          onboarding_bloc.dart
          splash_bloc.dart
      ∨ home
          map_bloc.dart
      ∨ market
          add_card_bloc.dart
          add_product_bloc.dart
          bank_transfer_bloc.dart
          cart_bloc.dart
          checkout_bloc.dart
          market_bloc.dart
          mobile_money_bloc.dart
      ∨ profile
          change_password_bloc.dart
          manage_bank_account_bloc.dart
          password_reset_success_dialog_bloc.dart
      ∨ services
          recycling_centers_bloc.dart
          service_bloc.dart
          service_detail_bloc.dart
          user_bloc.dart
    ∨ cubit
        add_card_cubit.dart
        market_cubit.dart
        tab_cubit.dart
    > data
```

Figure 5. File Structure when using Bloc and Cubit

Figure 6, shows the detailed on how the MultiBlocProvider was used for the 9jaclean project. This same behavior applies to other Cubits and Blocs throughout the application. In the case of AddCardCubit, as seen in the code snippet above, the cubit emits a boolean value representing a loading state. When setLoading(true) is called, it triggers an update, and any UI listening through BlocBuilder can show a loading spinner, disable buttons, or indicate progress to the user. This creates a more interactive and responsive experience, as the interface visually reacts to background operations in real time.

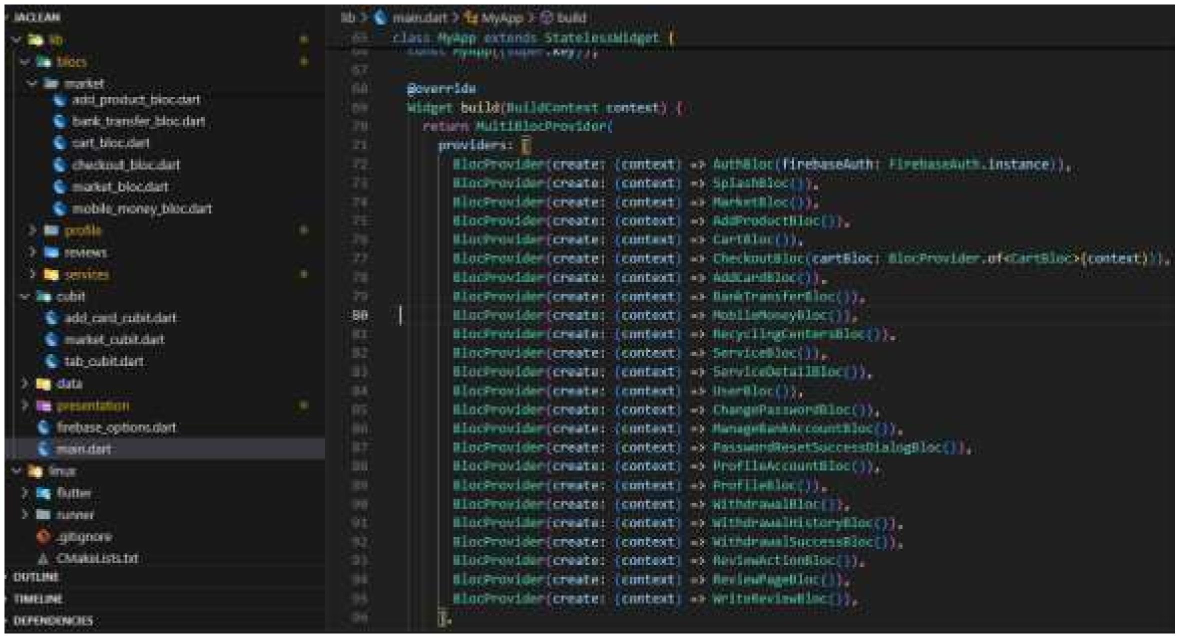
Once the process, as shown in Figure 7, such as adding a payment card is completed, setLoading(false) is called to revert the state. The UI automatically reflects this change by re-enabling any previously disabled elements and removing loading indicators. This mechanism helps reduce boilerplate logic in the view layer and ensures consistent behavior across the app.

As illustrated in Figure 8, the BlocBuilder widget plays a key role in dynamically responding to changes in the authentication state managed by the AuthBloc. In this example, the UI listens for the AuthError state. If an error message related to the email field is detected (i.e., it contains the word "email"), a Text widget is displayed in red below the input field, showing the relevant error message. If no such error is present, the UI displays an empty SizedBox, effectively hiding the message area. This approach ensures that users receive clear and immediate feedback during login or registration, helping them correct form input errors in real-time and improving the overall usability of the app.

Additionally, when a user launches the application, it begins in the AuthStateUninitialized state. During this phase, the BLoC triggers the AuthRepository to check whether the user is already authenticated, particularly as an admin. If authentication is confirmed, the app transitions to the AuthStateSignedIn state, which enables access to admin-specific functionalities within the app, such as options in the navigation drawer. If the user is not authenticated, the state changes accordingly, and the user is redirected to the login screen, ensuring proper access control and flow.

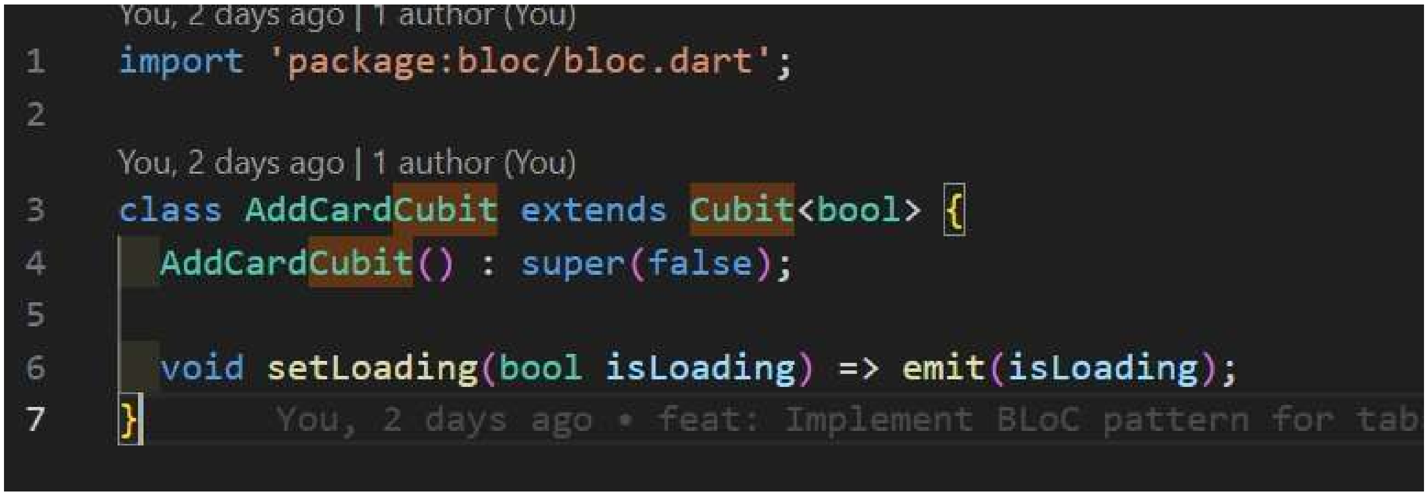
As shown in Figure 9, the AuthBloc class extends the Bloc base class and defines the relationship between AuthEvent and AuthState. It initializes with a required FirebaseAuth instance, which handles the underlying authentication logic via Firebase.

The bloc starts in the AuthInitial state, indicating that no action has yet occurred. Then, it registers event handlers for four key user actions: login, registration, logout, and password reset.



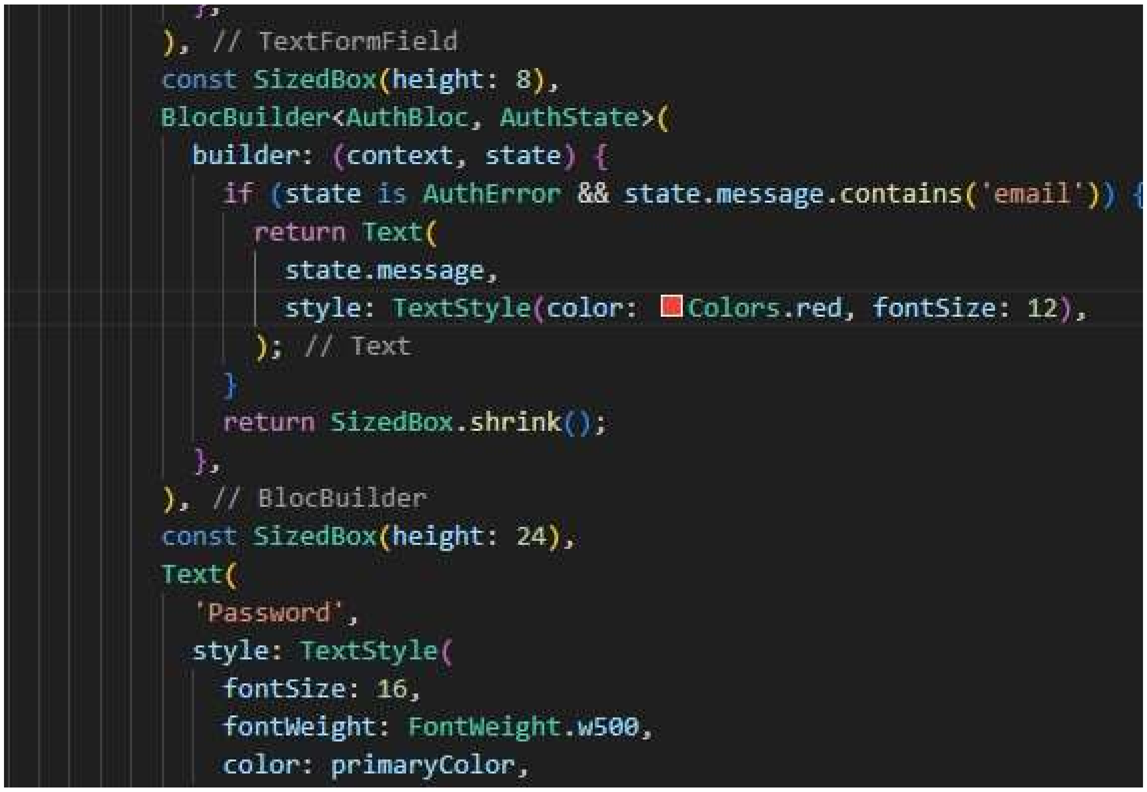
```dart
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MultiBlocProvider(
      providers: [
        BlocProvider(create: (context) => AuthBloc(firebaseAuth: FirebaseAuth.instance)),
        BlocProvider(create: (context) => SplashBloc()),
        BlocProvider(create: (context) => MarketBloc()),
        BlocProvider(create: (context) => AddProductBloc()),
        BlocProvider(create: (context) => CartBloc()),
        BlocProvider(create: (context) => CheckoutBloc(cartBloc: BlocProvider.of<CartBloc>(context))),
        BlocProvider(create: (context) => AddCardBloc()),
        BlocProvider(create: (context) => BankTransferBloc()),
        BlocProvider(create: (context) => MobileMoneyBloc()),
        BlocProvider(create: (context) => RecyclingCentersBloc()),
        BlocProvider(create: (context) => ServiceBloc()),
        BlocProvider(create: (context) => ServiceDetailBloc()),
        BlocProvider(create: (context) => UserBloc()),
        BlocProvider(create: (context) => ChangePasswordBloc()),
        BlocProvider(create: (context) => ManageBankAccountBloc()),
        BlocProvider(create: (context) => PasswordResetSuccessDialogBloc()),
        BlocProvider(create: (context) => ProfileAccountBloc()),
        BlocProvider(create: (context) => ProfileBloc()),
        BlocProvider(create: (context) => WithdrawalBloc()),
        BlocProvider(create: (context) => WithdrawalHistoryBloc()),
        BlocProvider(create: (context) => WithdrawalSuccessBloc()),
        BlocProvider(create: (context) => ReviewActionBloc()),
        BlocProvider(create: (context) => ReviewPageBloc()),
        BlocProvider(create: (context) => WriteReviewBloc()),
      ],
```

Figure 6. MultiBlocProvider in the main.dart file



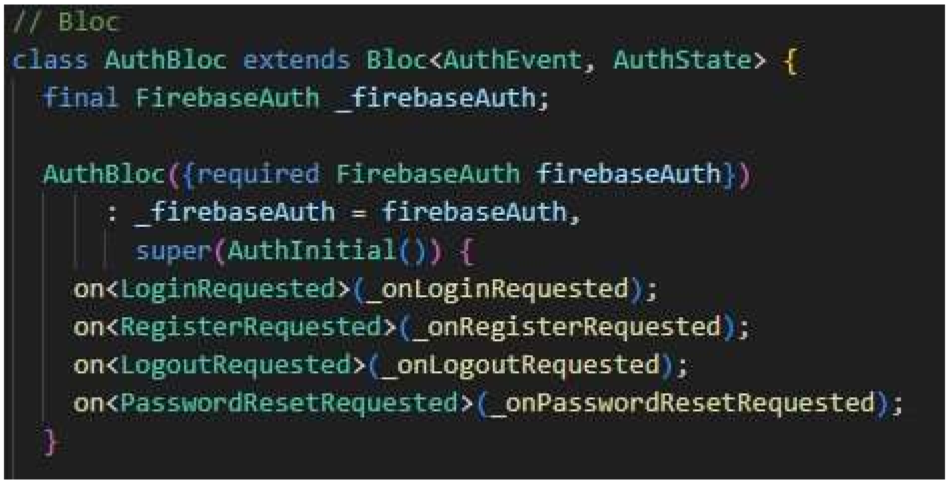
```dart
You, 2 days ago | 1 author (You)
import 'package:bloc/bloc.dart';

You, 2 days ago | 1 author (You)
class AddCardCubit extends Cubit<bool> {
  AddCardCubit() : super(false);

  void setLoading(bool isLoading) => emit(isLoading);
}        You, 2 days ago • feat: Implement BLoC pattern for tab
```

Figure 7. Functions of AddCardCubit



```
    }),
    ), // TextFormField
    const SizedBox(height: 8),
    BlocBuilder<AuthBloc, AuthState>(
      builder: (context, state) {
        if (state is AuthError && state.message.contains('email')) {
          return Text(
            state.message,
            style: TextStyle(color: Colors.red, fontSize: 12),
          ); // Text
        }
        return SizedBox.shrink();
      },
    ), // BlocBuilder
    const SizedBox(height: 24),
    Text(
      'Password',
      style: TextStyle(
        fontSize: 16,
        fontWeight: FontWeight.w500,
        color: primaryColor,
```

Figure 8. BlocBuilder using AuthBloc



```
// Bloc
class AuthBloc extends Bloc<AuthEvent, AuthState> {
  final FirebaseAuth _firebaseAuth;

  AuthBloc({required FirebaseAuth firebaseAuth})
      : _firebaseAuth = firebaseAuth,
        super(AuthInitial()) {
    on<LoginRequested>(_onLoginRequested);
    on<RegisterRequested>(_onRegisterRequested);
    on<LogoutRequested>(_onLogoutRequested);
    on<PasswordResetRequested>(_onPasswordResetRequested);
  }
```

Figure 9. AuthBloc class

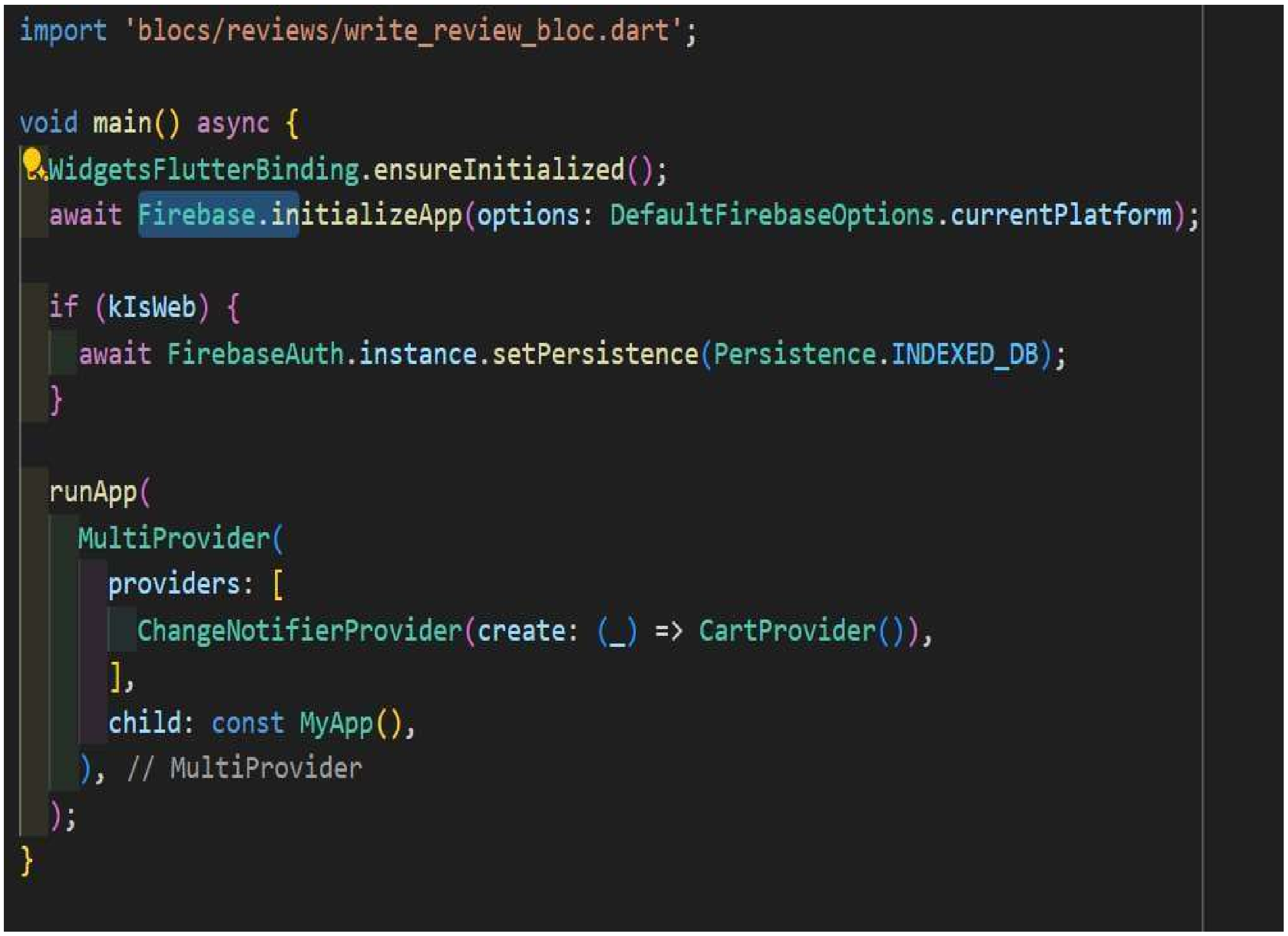
3.3 Google Firebase

Firebase served as the backend infrastructure for the 9jaclean mobile application, enabling essential features such as user authentication and real-time data management. These capabilities provided a secure and scalable foundation for the platform, allowing seamless management of user sessions, marketplace transactions, and live updates. Firebase is particularly valuable in mobile development due to its ease of integration and support for cross-platform synchronization (Firebase Documentation, 2024).

Firebase Authentication was implemented to handle user login, registration, and password reset functionalities within the app. Users could create accounts using their email and password, while an email verification process was also added to confirm account ownership before granting full access to features like pickups or wallet transactions. Figure 10 shows the firebase authentication implementation. This service streamlined the user management process without requiring developers to build custom back-end logic, making it both secure and efficient (Rosencrance, 2019).

For storing and syncing data, the application leveraged **Cloud Firestore**. Firebase's cloud-based NoSQL database. Firestore was used to store structured information such as pickup requests, marketplace items, wallet transactions, and user reviews. These were organized into document-based collections like *Users*, *Pickups*, *Products*, and *Reviews* that supported advanced querying, sorting, and filtering operations.
Firestore's ability to sync data in real time across devices ensured a smooth and dynamic user experience (Krause, 2022).

One of Firestore's most valuable advantages was its real-time update capability, which made it ideal for a responsive application like 9jaclean. For example, when a user scheduled a pickup or listed an item for sale, the update was instantly reflected on other users' devices. This removed the need for manual data refresh and contributed to a fluid, modern app experience as shown in Figure 10. In addition, Firebase's customizable security rules allowed



```dart
import 'blocs/reviews/write_review_bloc.dart';

void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp(options: DefaultFirebaseOptions.currentPlatform);

  if (kIsWeb) {
    await FirebaseAuth.instance.setPersistence(Persistence.INDEXED_DB);
  }

  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (_) => CartProvider()),
      ],
      child: const MyApp(),
    ), // MultiProvider
  );
}
```

Figure 10. Firebase Authentication implementation

developers to define granular permissions, restricting database access based on user roles and authentication status (Patel, 2023).

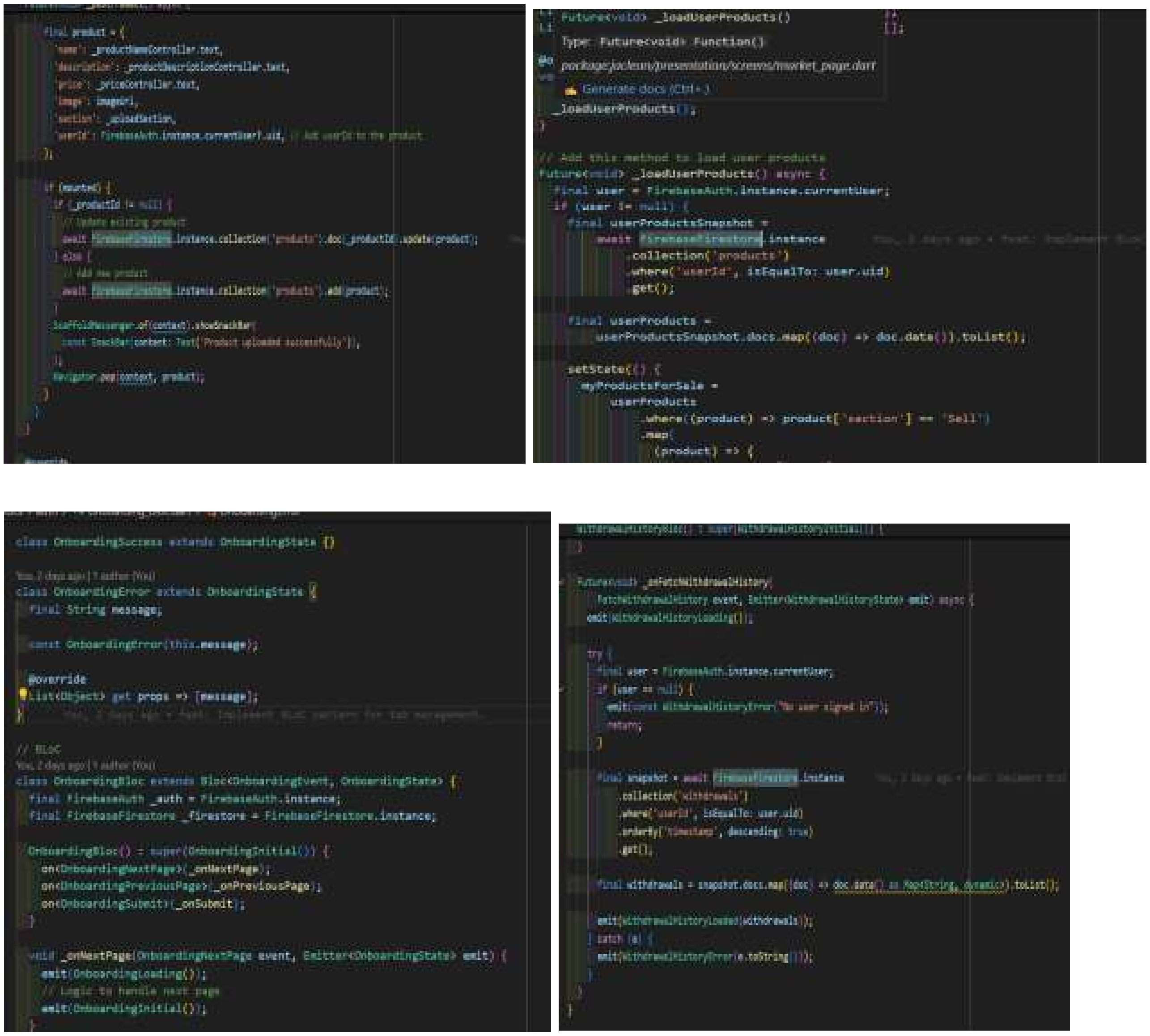
The development team also used the **Firebase Command Line Interface (CLI)** to set up and connect the Flutter application with Firebase. Firebase CLI automated the initialization of services and generated required configuration files for both Android and iOS platforms, streamlining deployment and testing. It also simplified project linking, especially when managing multiple environments or working collaboratively (Firebase CLI Docs, 2024).

As demonstrated in Figure 11, Firestore plays a central role in storing and retrieving data across multiple tabs within the 9jaclean application. Whether users are posting new products, managing onboarding submissions, or browsing through market listings, all these actions interact with Firestore to ensure data is stored in real-time and can be accessed seamlessly across devices.

In the Market section, Firestore is used to store product listings uploaded by users. Figure 10 shows how user-submitted product information such as name, description, price, image URL, and section is packaged into a document and saved to the products collection in Firestore. When a user adds a new product, the app either updates an existing document or creates a new one using the add() method. This data is then used to display product listings dynamically in the app's marketplace.

The Add Product tab also relies heavily on Firestore, particularly for fetching user-specific products. As seen in Figure 11, the _loadUserProducts() method queries the products collection using a where clause that filters items based on the logged-in user's ID. This ensures that users only see their own items, and any changes they make (such as edits or deletions) are reflected instantly thanks to Firestore's real-time sync capabilities.

Similarly, the Onboarding tab makes use of Firestore to save the information collected during the onboarding process. This includes user details, preferences, and selected options. The OnboardingBloc interacts



```dart
    final product = {
      'name': _productNameController.text,
      'description': _productDescriptionController.text,
      'price': _priceController.text,
      'image': imageUrl,
      'section': _uploadSection,
      'userId': FirebaseAuth.instance.currentUser?.uid, // Add userId to the product
    };

    if (mounted) {
      if (_productId != null) {
        // Update existing product
        await FirebaseFirestore.instance.collection('products').doc(_productId).update(product);
      } else {
        // Add new product
        await FirebaseFirestore.instance.collection('products').add(product);
      }
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(content: Text('Product uploaded successfully')),
      );
      Navigator.pop(context, product);
    }
  }
}
```

```dart
  Future<void> _loadUserProducts()
  Type: Future<void> Function()
  package:jacleum/presentation/screens/market_page.dart
  Generate docs (Ctrl+.)
  _loadUserProducts();
}

// Add this method to load user products
Future<void> _loadUserProducts() async {
  final user = FirebaseAuth.instance.currentUser;
  if (user != null) {
    final userProductsSnapshot =
        await FirebaseFirestore.instance
            .collection('products')
            .where('userId', isEqualTo: user.uid)
            .get();

    final userProducts =
        userProductsSnapshot.docs.map((doc) => doc.data()).toList();

    setState(() {
      myProductsForSale =
          userProducts
              .where((product) => product['section'] == 'Sell')
              .map(
                (product) => {
```

```dart
class OnboardingSuccess extends OnboardingState {}

class OnboardingError extends OnboardingState {
  final String message;

  const OnboardingError(this.message);

  @override
  List<Object> get props => [message];
}

// BLoC
class OnboardingBloc extends Bloc<OnboardingEvent, OnboardingState> {
  final FirebaseAuth _auth = FirebaseAuth.instance;
  final FirebaseFirestore _firestore = FirebaseFirestore.instance;

  OnboardingBloc() : super(OnboardingInitial()) {
    on<OnboardingNextPage>(_onNextPage);
    on<OnboardingPreviousPage>(_onPreviousPage);
    on<OnboardingSubmit>(_onSubmit);
  }

  void _onNextPage(OnboardingNextPage event, Emitter<OnboardingState> emit) {
    emit(OnboardingLoading());
    // Logic to handle next page
    emit(OnboardingInitial());
  }
```

```dart
  WithdrawalHistoryBloc() : super(WithdrawalHistoryInitial()) {
  }

  Future<void> _onFetchWithdrawalHistory(
      FetchWithdrawalHistory event, Emitter<WithdrawalHistoryState> emit) async {
    emit(WithdrawalHistoryLoading());

    try {
      final user = FirebaseAuth.instance.currentUser;
      if (user == null) {
        emit(const WithdrawalHistoryError('No user signed in'));
        return;
      }

      final snapshot = await FirebaseFirestore.instance
          .collection('withdrawals')
          .where('userId', isEqualTo: user.uid)
          .orderBy('timestamp', descending: true)
          .get();

      final withdrawals = snapshot.docs.map((doc) => doc.data() as Map<String, dynamic>).toList();

      emit(WithdrawalHistoryLoaded(withdrawals));
    } catch (e) {
      emit(WithdrawalHistoryError(e.toString()));
    }
  }
}
```

Figure 11. Cloud Firestore implementation

3.4 Google Maps and OpenRouteServices API

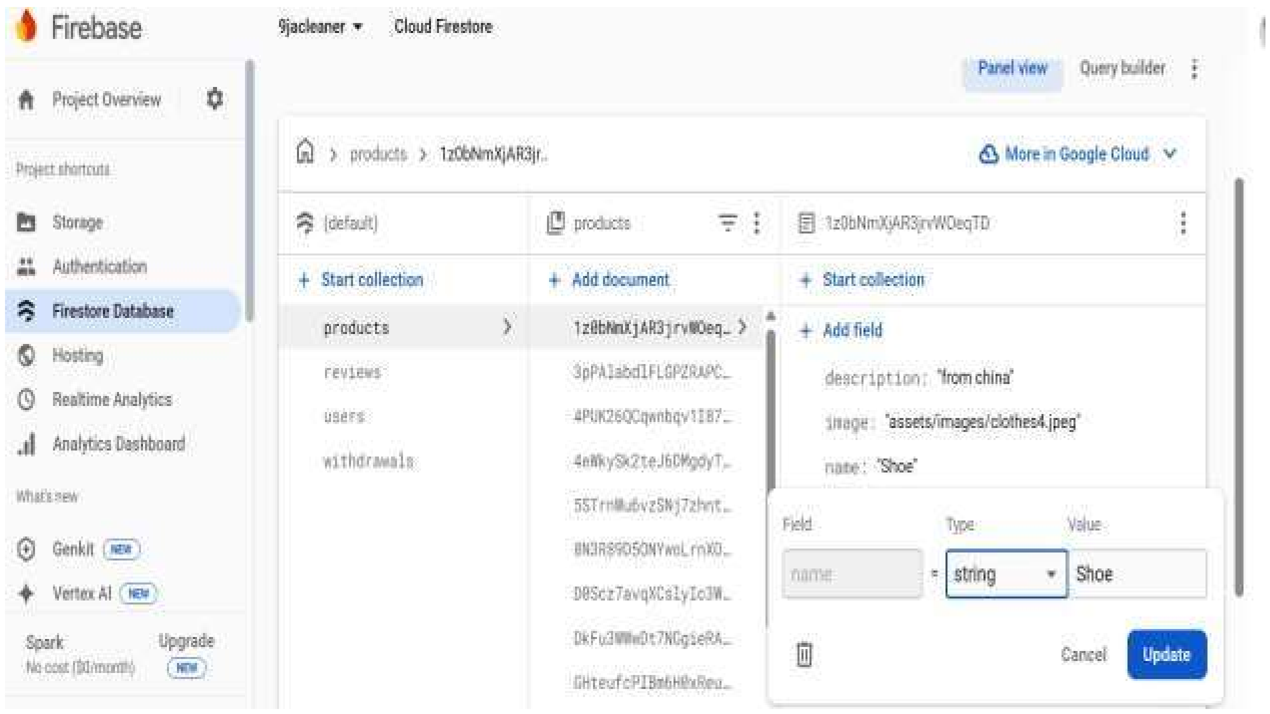
In the 9jaclean application, Google Maps and OpenRouteService API were integrated to allow users to conveniently view the location of recycling centers on an interactive map. This feature helps users identify the nearest drop-off points based on their current location, supporting the app's goal of promoting accessible and eco-friendly waste management.

Google Maps is used to render the map interface within the app. Through the Maps SDK for Android and iOS, we enabled the app to display real-time locations and map markers for recycling centers. The API key, generated from the Google Cloud Console, was configured in AndroidManifest.xml and AppDelegate.swift to ensure proper integration with Flutter (Google Cloud, 2023).

To provide routing functionality, especially in cases where the Google Directions API is restricted due to billing constraints, the project incorporates the OpenRouteService API. This open-source service allows the application to draw dynamic route polylines between the user's current GPS location and a selected recycling center. It enhances the usability of the map by visually guiding users to their selected center with accurate directions (OpenRouteService, 2023).

This map feature is especially useful during onboarding or while exploring services. When a user selects a recycling center, the coordinates are passed to the map view where the location is marked and navigable as shown in Figure 12. This interaction not only improves user orientation but also encourages engagement with nearby centers by making them easily discoverable and reachable (Rohan, 2020).

In the 9jaclean project, map functionality was implemented to help users locate recycling centers near them. After setting up a Google Cloud account, an API key was generated and added to two core configuration files AndroidManifest.xml for Android and AppDelegate.swift for iOS to enable map integration in Flutter. This step is essential for the Google Maps services to function properly across platforms. To create direction paths between the user's current location and a specific recycling center, as shown in Figure 13 the Directions API would normally be used. However, as seen in the implementation, the app instead integrates **OpenRouteServices API**, a free and open-source alternative. This is particularly useful when developers face challenges with Google's billing requirements The map view is powered by the google_maps_flutter and flutter_map packages. Coordinates of the recycling center (latitude and longitude) are passed through MapBloc, which renders the location dynamically.


Firebase 9jacleaner Cloud Firestore

Panel view Query builder

products > 1z0bNmXjAR3jr..

More in Google Cloud

(default) | products | 1z0bNmXjAR3jrvWOeqTD

Start collection | Add document | Start collection

products | 1z0bNmXjAR3jrvWOeq… | Add field

reviews | 3pPA1abdIFLGPZRAPC… | description: "from china"

users | 4PUK26QCqwnbqv1I87… | image: "assets/images/clothes4.jpeg"

withdrawals | 4eWkySk2teJ6DMgdyT… | name: "Shoe"

55TrnWubvzSNj7zhnt…

8N3R89D5ONYwoLrnXO…

D0Scz7avqXCslyIc3W…

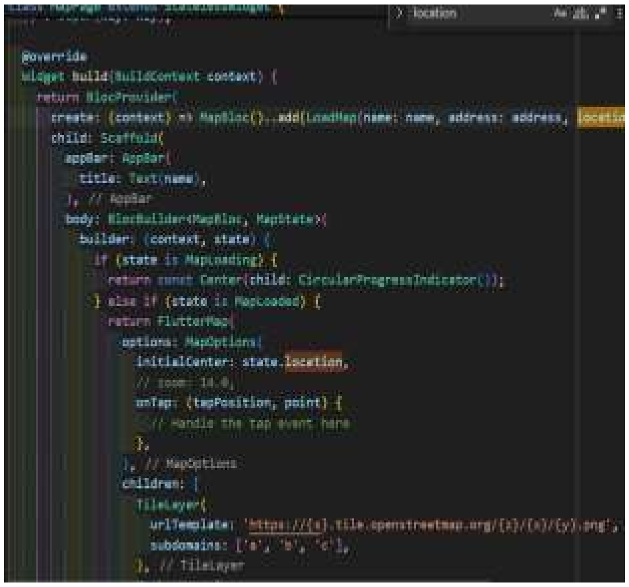
DkFu3WWwDt7NGgieRA…

GHteufcPIBm6H0xReu…

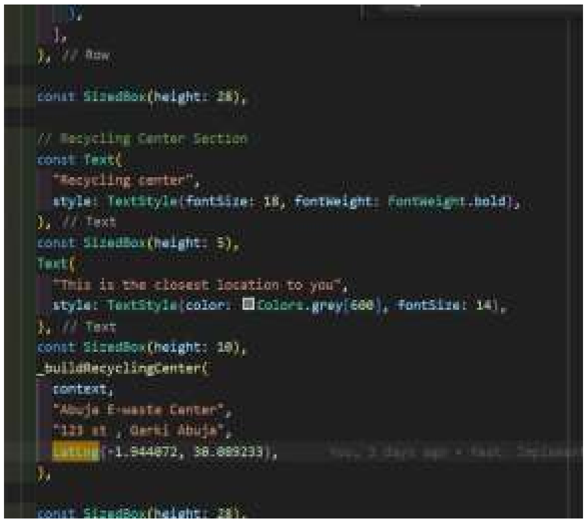
Field: name = Type: string Value: Shoe

Cancel Update

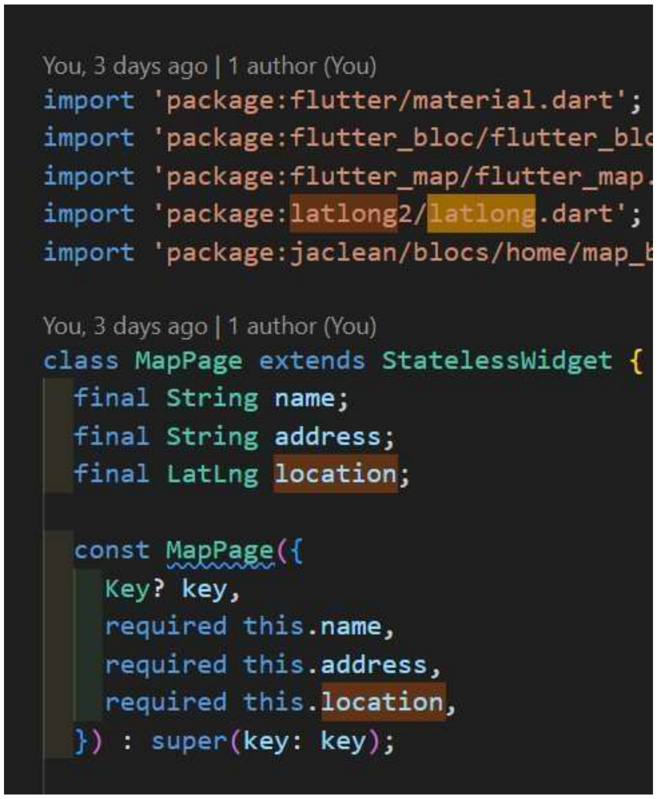
Figure 12. Enable APIs and Services



```
@override
Widget build(BuildContext context) {
  return BlocProvider(
    create: (context) => MapBloc()..add(LoadMap(name: name, address: address, locatio
    child: Scaffold(
      appBar: AppBar(
        title: Text(name),
      ), // AppBar
      body: BlocBuilder<MapBloc, MapState>(
        builder: (context, state) {
          if (state is MapLoading) {
            return const Center(child: CircularProgressIndicator());
          } else if (state is MapLoaded) {
            return FlutterMap(
              options: MapOptions(
                initialCenter: state.location,
                // zoom: 14.0,
                onTap: (tapPosition, point) {
                  // Handle the tap event here
                },
              ), // MapOptions
              children: [
                TileLayer(
                  urlTemplate: 'https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png',
                  subdomains: ['a', 'b', 'c'],
                ), // TileLayer
```



```
      ),
    ],
  ), // Row

  const SizedBox(height: 28),

  // Recycling Center Section
  const Text(
    "Recycling center",
    style: TextStyle(fontSize: 18, fontWeight: FontWeight.bold),
  ), // Text
  const SizedBox(height: 5),
  Text(
    "This is the closest location to you",
    style: TextStyle(color: Colors.grey[600], fontSize: 14),
  ), // Text
  const SizedBox(height: 10),
  _buildRecyclingCenter(
    context,
    "Abuja E-waste Center",
    "123 st , Garki Abuja",
    LatLng(-1.944072, 30.089233),
  ),

  const SizedBox(height: 28),
```



```
You, 3 days ago | 1 author (You)
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_blo
import 'package:flutter_map/flutter_map.
import 'package:latlong2/latlong.dart';
import 'package:jaclean/blocs/home/map_b

You, 3 days ago | 1 author (You)
class MapPage extends StatelessWidget {
  final String name;
  final String address;
  final LatLng location;

  const MapPage({
    Key? key,
    required this.name,
    required this.address,
    required this.location,
  }) : super(key: key);
```

Figure 13. Open Route Services API implementation

METHODOLOGY

4.1 Project Methodology

To ensure that the development of the 9jaclean mobile application met its functional and performance requirements on time, an organized project management approach was essential. For this reason, the Agile methodology, in combination with the Scrum framework, was adopted throughout the project lifecycle. These methodologies were selected because of their iterative nature, flexibility, and ability to adapt to changing user and stakeholder requirements.

Agile software development emphasizes delivering working software in incremental phases, promoting continuous collaboration between team members, developers, and stakeholders. Rather than waiting for a final, large-scale deployment, Agile encourages the release of smaller functional parts that can be tested, reviewed, and refined. This approach fosters frequent feedback, reduces risk, and improves product quality (Highsmith, 2022).

Figure 14 illustrates the Agile lifecycle, which includes several iterative stages: requirement analysis, design, development, testing, deployment, maintenance, and final release. By breaking the development into manageable sprints, the team was able to prioritize features such as user registration, authentication, item donations, recycling center navigation, and wallet transactions based on feedback and evolving priorities. This method helped ensure that the app was built in alignment with user needs and technical feasibility.

Scrum, a popular Agile framework, was used to implement this methodology more effectively. It allowed the team to organize development tasks into sprints short, time-boxed iterations typically lasting one to two weeks. At the beginning of each sprint, user stories and tasks were defined and assigned during planning sessions. Daily

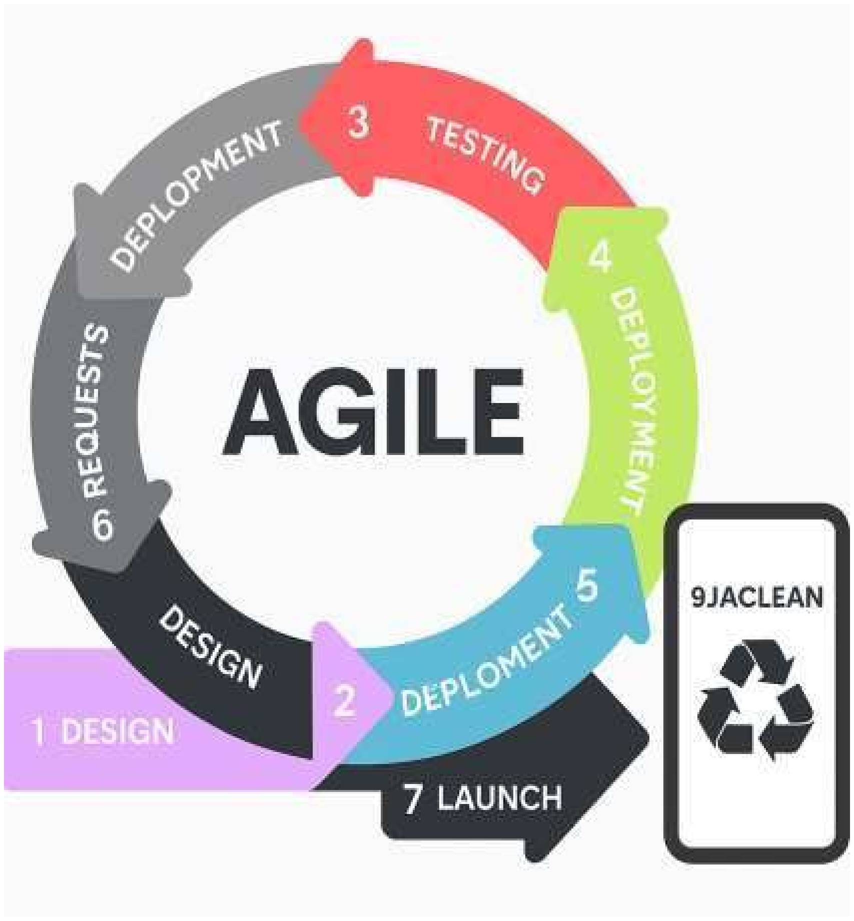


Figure 14. Agile lifecycle methodology

stand-up meetings were held to monitor progress and tackle blockers, while sprint reviews and retrospectives ensured continuous improvement of both the process and the product (Schwaber & Sutherland, 2020).

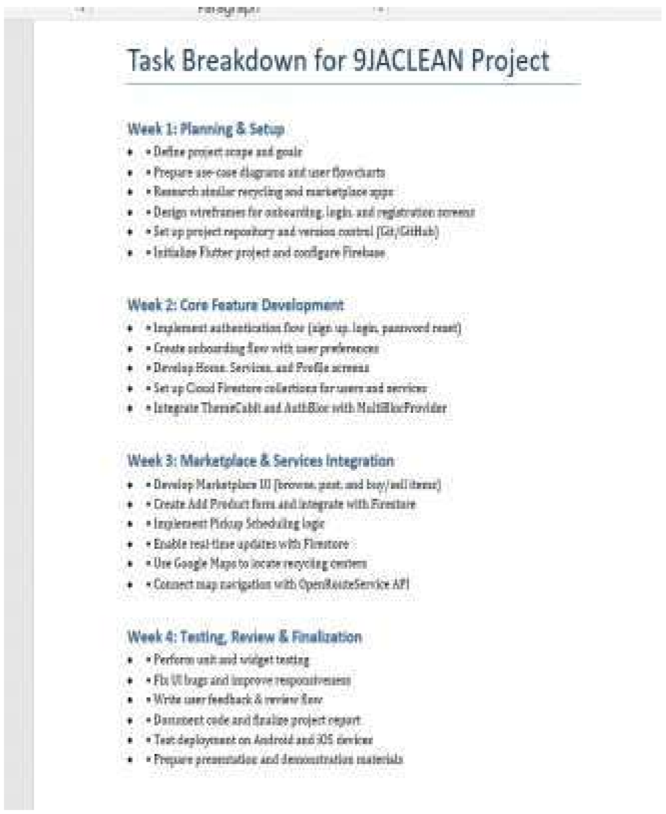
Agile with Scrum proved especially effective for 9jaclean, as the project involved multiple interconnected features marketplace management, recycling logistics, profile handling, and payment systems all requiring close coordination. The methodology enabled the development team to stay on track, deliver functional updates regularly, and integrate new features based on user feedback without deviating from the project timeline.

4.2 Overview of the Development Process and Tools

To manage the 9jaclean project efficiently, the development team adopted a lightweight and collaborative toolset centered around **Google Workspace**, primarily Google Docs and Google Sheets. These tools were used to coordinate tasks, document requirements, track progress, and assign responsibilities. Unlike traditional enterprise project management tools like Azure DevOps, Google Workspace offers real-time collaboration, accessibility, and ease of use qualities especially suitable for small to mid-sized development teams (O'Connell, 2022).

As shown in Figure 15, the team organized work into weekly sprints aligned with Agile methodology. Each sprint focused on a specific feature set such as onboarding, authentication, or the marketplace module. Large tasks were broken down into smaller sub-tasks to facilitate progress tracking and ensure timely delivery of functional components. This iterative and incremental development model promoted flexibility and responsiveness to feedback (Schwaber & Sutherland, 2020).

During the early development stage, flowcharts and use-case diagrams were created to visualize the user journey and define system behavior. These diagrams played a key role in ensuring functional completeness and the core technology used in the development of the 9jaclean application is the Flutter framework**,** created by Google and based on the Dart programming language.



Paragraph

Task Breakdown for 9JACLEAN Project

Week 1: Planning & Setup

- • Define project scope and goals
- • Prepare use-case diagrams and user flowcharts
- • Research similar recycling and marketplace apps
- • Design wireframes for onboarding, login, and registration screens
- • Set up project repository and version control (Git/GitHub)
- • Initialize Flutter project and configure Firebase

Week 2: Core Feature Development

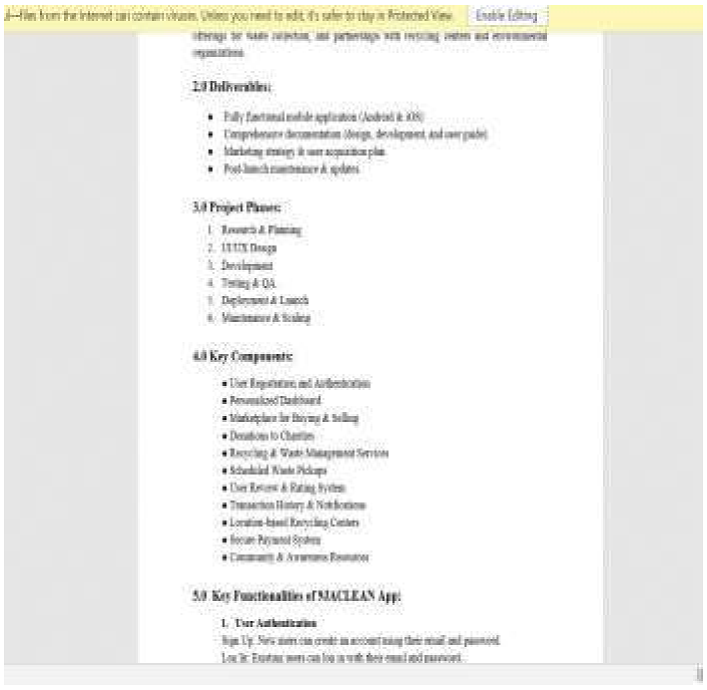
- • Implement authentication flow (sign up, login, password reset)
- • Create onboarding flow with user preferences
- • Develop Home, Services, and Profile screens
- • Set up Cloud Firestore collections for users and services
- • Integrate ThemeCubit and AuthBloc with MultiBlocProvider

Week 3: Marketplace & Services Integration

- • Develop Marketplace UI (browse, post, and buy/sell items)
- • Create Add Product form and integrate with Firestore
- • Implement Pickup Scheduling logic
- • Enable real-time updates with Firestore
- • Use Google Maps to locate recycling centers
- • Connect map navigation with OpenRouteService API

Week 4: Testing, Review & Finalization

- • Perform unit and widget testing
- • Fix UI bugs and improve responsiveness
- • Write user feedback & review flow
- • Document code and finalize project report
- • Test deployment on Android and iOS devices
- • Prepare presentation and demonstration materials



—Files from the internet can contain viruses. Unless you need to edit, it's safer to stay in Protected View. Enable Editing

offerings for waste collection, and partnerships with recycling centers and environmental organizations.

2.0 Deliverables:

- Fully functional mobile application (Android & iOS)
- Comprehensive documentation (design, development, and user guide)
- Marketing strategy & user acquisition plan
- Post-launch maintenance & updates

3.0 Project Phases:

1. Research & Planning
2. UI/UX Design
3. Development
4. Testing & QA
5. Deployment & Launch
6. Maintenance & Scaling

4.0 Key Components:

- ▪ User Registration and Authentication
- ▪ Personalized Dashboard
- ▪ Marketplace for Buying & Selling
- ▪ Donations to Charities
- ▪ Recycling & Waste Management Services
- ▪ Scheduled Waste Pickups
- ▪ User Review & Rating System
- ▪ Transaction History & Notifications
- ▪ Location-based Recycling Centers
- ▪ Secure Payment System
- ▪ Community & Awareness Resources

5.0 Key Functionalities of 9JACLEAN App:

1. **User Authentication**

Sign Up: New users can create an account using their email and password.
Log In: Existing users can log in with their email and password.

Figure 15. Google Workplace

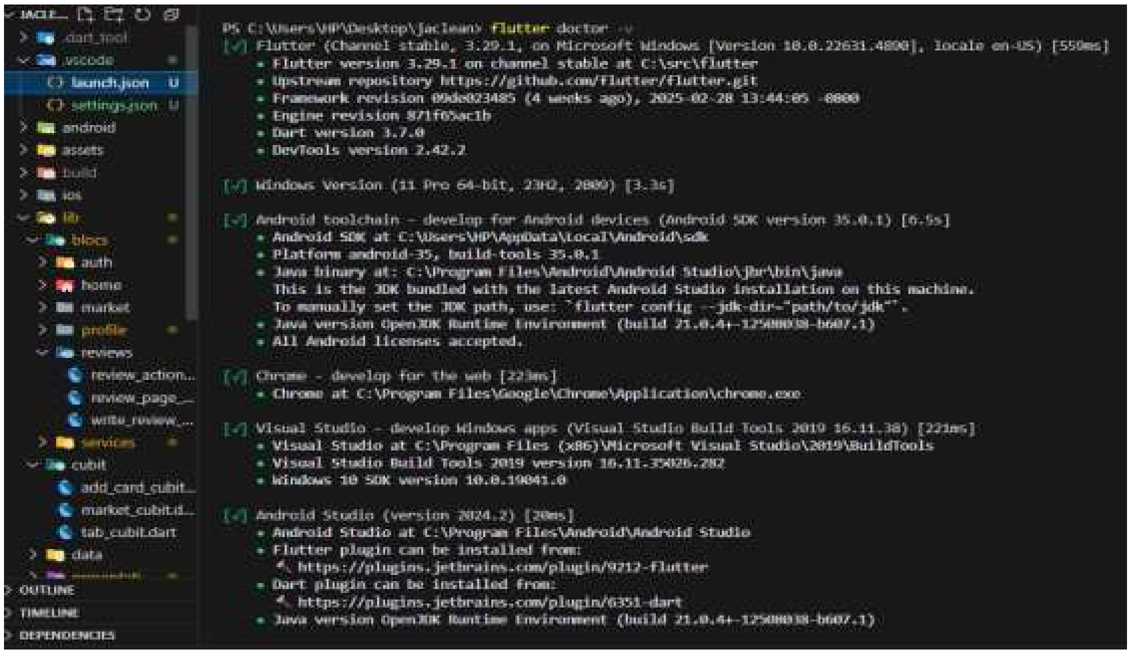
Flutter enables developers to build highperformance applications for Android, iOS, web, and desktop using just one codebase, which improves development speed and consistency across platforms (Google Developers, 2023).

To begin working with Flutter, the Flutter SDK (Software Development Kit) must be installed. Developers can either download it directly from the Flutter documentation website or clone it from GitHub. When downloading the SDK, there are two options: the stable version and the beta version. The stable version is recommended by the developer community because it is thoroughly tested and avoids the risk of encountering errors during development. The beta version, on the other hand, includes early access to features that are still being finalized. For this project, the team used the stable version to ensure a smooth and predictable development experience (Janson, 2022).

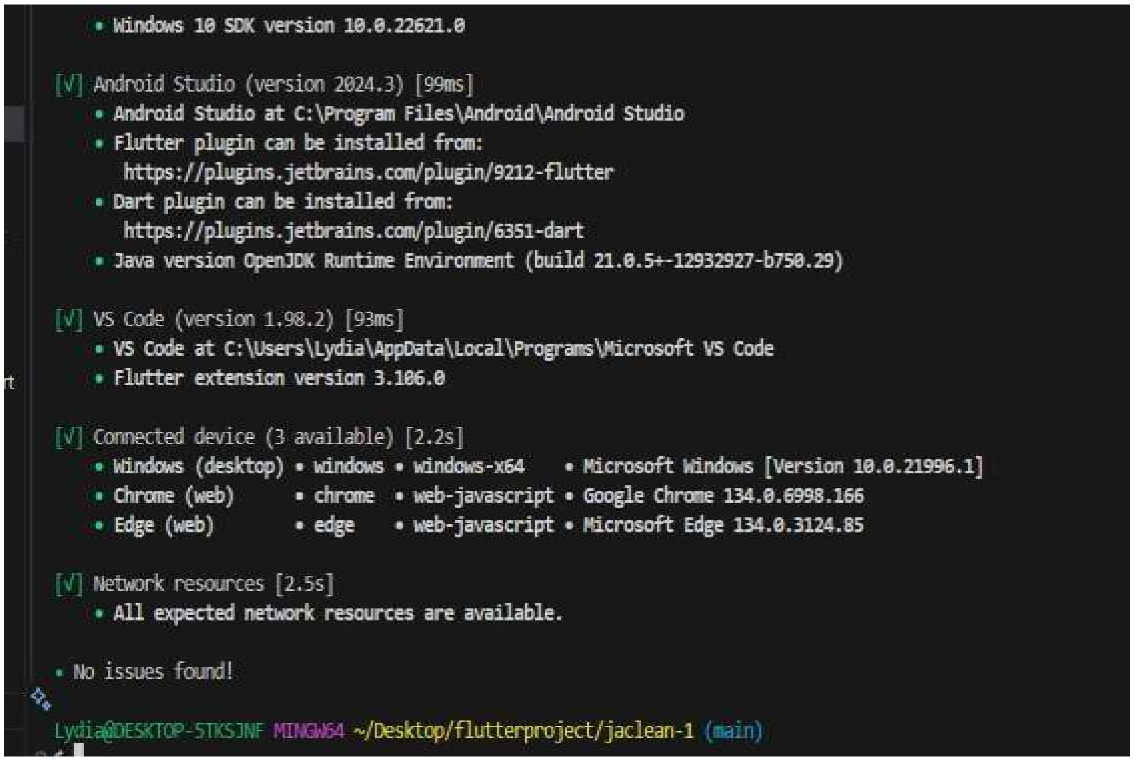
In addition to the SDK, development required proper configuration of supporting tools. Android Studio was used to develop and debug the Android version of the app, while Xcode was necessary for compiling the iOS version. To enable proper communication between Flutter and Android devices, the Android SDK and Java Development Kit (JDK) were added to the system's environment variables.

Flutter includes a command-line tool called flutter doctor, which helps developers verify that their environment is correctly set up as shown in Figure 16. Running this command checks for missing components and confirms whether Flutter has access to the required dependencies. Developers can also include the -v flag to get detailed reports about their configuration, available platforms, and potential issues that need to be fixed.

This setup process ensured that every member of the development team had a consistent and functioning environment, allowing them to work efficiently across platforms and reduce time spent troubleshooting setup



```
PS C:\Users\HP\Desktop\jaclean> flutter doctor -v
[√] Flutter (Channel stable, 3.29.1, on Microsoft Windows [Version 10.0.22631.4890], locale en-US) [550ms]
    • Flutter version 3.29.1 on channel stable at C:\src\flutter
    • Upstream repository https://github.com/flutter/flutter.git
    • Framework revision 09de023485 (4 weeks ago), 2025-02-28 13:44:05 -0800
    • Engine revision 871f65ac1b
    • Dart version 3.7.0
    • DevTools version 2.42.2

[√] Windows Version (11 Pro 64-bit, 23H2, 2009) [3.3s]

[√] Android toolchain - develop for Android devices (Android SDK version 35.0.1) [6.5s]
    • Android SDK at C:\Users\HP\AppData\Local\Android\sdk
    • Platform android-35, build-tools 35.0.1
    • Java binary at: C:\Program Files\Android\Android Studio\jbr\bin\java
      This is the JDK bundled with the latest Android Studio installation on this machine.
      To manually set the JDK path, use: `flutter config --jdk-dir="path/to/jdk"`.
    • Java version OpenJDK Runtime Environment (build 21.0.4+-12508038-b607.1)
    • All Android licenses accepted.

[√] Chrome - develop for the web [223ms]
    • Chrome at C:\Program Files\Google\Chrome\Application\chrome.exe

[√] Visual Studio - develop Windows apps (Visual Studio Build Tools 2019 16.11.38) [223ms]
    • Visual Studio at C:\Program Files (x86)\Microsoft Visual Studio\2019\BuildTools
    • Visual Studio Build Tools 2019 version 16.11.35026.282
    • Windows 10 SDK version 10.0.19041.0

[√] Android Studio (version 2024.2) [20ms]
    • Android Studio at C:\Program Files\Android\Android Studio
    • Flutter plugin can be installed from:
      🔨 https://plugins.jetbrains.com/plugin/9212-flutter
    • Dart plugin can be installed from:
      🔨 https://plugins.jetbrains.com/plugin/6351-dart
    • Java version OpenJDK Runtime Environment (build 21.0.4+-12508038-b607.1)
```



```
    • Windows 10 SDK version 10.0.22621.0

[√] Android Studio (version 2024.3) [99ms]
    • Android Studio at C:\Program Files\Android\Android Studio
    • Flutter plugin can be installed from:
      https://plugins.jetbrains.com/plugin/9212-flutter
    • Dart plugin can be installed from:
      https://plugins.jetbrains.com/plugin/6351-dart
    • Java version OpenJDK Runtime Environment (build 21.0.5+-12932927-b750.29)

[√] VS Code (version 1.98.2) [93ms]
    • VS Code at C:\Users\Lydia\AppData\Local\Programs\Microsoft VS Code
    • Flutter extension version 3.106.0

[√] Connected device (3 available) [2.2s]
    • Windows (desktop) • windows • windows-x64    • Microsoft Windows [Version 10.0.21996.1]
    • Chrome (web)      • chrome  • web-javascript • Google Chrome 134.0.6998.166
    • Edge (web)        • edge    • web-javascript • Microsoft Edge 134.0.3124.85

[√] Network resources [2.5s]
    • All expected network resources are available.

• No issues found!

Lydia@DESKTOP-5TKSJNF MINGW64 ~/Desktop/flutterproject/jaclean-1 (main)
```

Figure 16. Running "flutter doctor -v

issues. Flutter’s simplicity and developer-friendly tooling made it ideal for building a large, scalable application like 9jaclean.

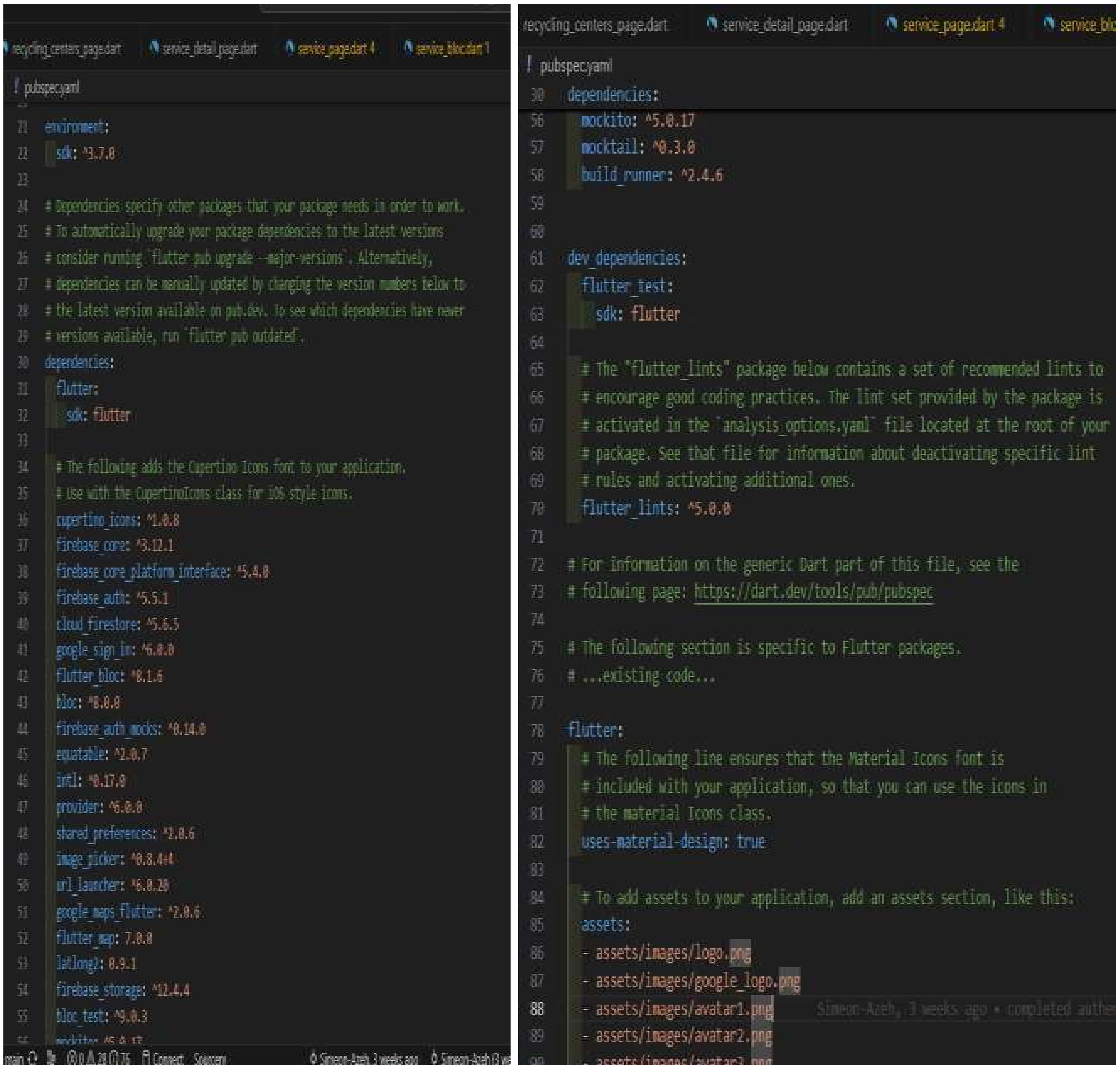
The BLoC design pattern, Firebase Authentication, and other integrated services within the 9jaclean project depend heavily on external libraries known as dependencies. These dependencies are declared inside the pubspec.yaml file, which serves as the central configuration for managing packages in any Flutter project. This file defines all the tools and libraries required for building, testing, and running the application. Without properly defining and managing dependencies, the application’s core features like login, registration, Firestore integration, and real-time state management would not function.

To explore, compare, or better understand these packages, Flutter developers typically refer to pub.dev, which is the official Dart and Flutter package repository. Pub.dev offers in-depth documentation, popularity scores, and example implementations, making it a critical resource for evaluating and selecting the right tools (Dart Dev Team, 2024). From commonly used packages like flutter_bloc, firebase_auth, and cloud_firestore to UI libraries such as google_fonts or flutter_spinkit, the platform ensures that developers can build with confidence using well-supported tools.

In the case of 9jaclean, as shown in Figure 17, the pubspec.yaml file includes a variety of essential packages. These were required to implement core modules such as authentication (via firebase_auth), cloud-based data storage (via cloud_firestore), and state management (via flutter_bloc and equatable). Additional tools were also included to support file uploads, UI consistency, and network handling.

After listing the required packages, the flutter pub get command must be run to fetch and integrate them into the project environment. This process ensures that all declared libraries are downloaded and linked correctly. In **Visual Studio Code**, this operation is often performed automatically when the file is saved (e.g., by pressing **Ctrl + S**), streamlining the setup and keeping the project synchronized.

Using trusted and well-maintained libraries especially those from Google helped speed up development in the 9jaclean project. Instead of building complex features from scratch, packages like url_launcher, permission_handler, and shared_preferences were used to handle tasks such as opening web views, requesting user permissions, and storing local preferences. These tools reduced development time and improved app reliability, as they are optimized for both Android and iOS (Wang et al., 2021).



```yaml
environment:
  sdk: ^3.7.0

# Dependencies specify other packages that your package needs in order to work.
# To automatically upgrade your package dependencies to the latest versions
# consider running `flutter pub upgrade --major-versions`. Alternatively,
# dependencies can be manually updated by changing the version numbers below to
# the latest version available on pub.dev. To see which dependencies have newer
# versions available, run `flutter pub outdated`.
dependencies:
  flutter:
    sdk: flutter

  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^1.0.8
  firebase_core: ^3.12.1
  firebase_core_platform_interface: ^5.4.0
  firebase_auth: ^5.5.1
  cloud_firestore: ^5.6.5
  google_sign_in: ^6.0.0
  flutter_bloc: ^8.1.6
  bloc: ^8.0.0
  firebase_auth_mocks: ^0.14.0
  equatable: ^2.0.7
  intl: ^0.17.0
  provider: ^6.0.0
  shared_preferences: ^2.0.6
  image_picker: ^0.8.4+4
  url_launcher: ^6.0.20
  google_maps_flutter: ^2.0.6
  flutter_map: 7.0.0
  latlong2: 0.9.1
  firebase_storage: ^12.4.4
  bloc_test: ^9.0.3
  mockito: ^5.0.17
```

```yaml
dependencies:
  mockito: ^5.0.17
  mocktail: ^0.3.0
  build_runner: ^2.4.6


dev_dependencies:
  flutter_test:
    sdk: flutter

  # The "flutter_lints" package below contains a set of recommended lints to
  # encourage good coding practices. The lint set provided by the package is
  # activated in the `analysis_options.yaml` file located at the root of your
  # package. See that file for information about deactivating specific lint
  # rules and activating additional ones.
  flutter_lints: ^5.0.0

# For information on the generic Dart part of this file, see the
# following page: https://dart.dev/tools/pub/pubspec

# The following section is specific to Flutter packages.
# ...existing code...

flutter:
  # The following line ensures that the Material Icons font is
  # included with your application, so that you can use the icons in
  # the material Icons class.
  uses-material-design: true

  # To add assets to your application, add an assets section, like this:
  assets:
  - assets/images/logo.png
  - assets/images/google_logo.png
  - assets/images/avatar1.png      Simeon-Azeh, 3 weeks ago • completed auther
  - assets/images/avatar2.png
  - assets/images/avatar3.png
```

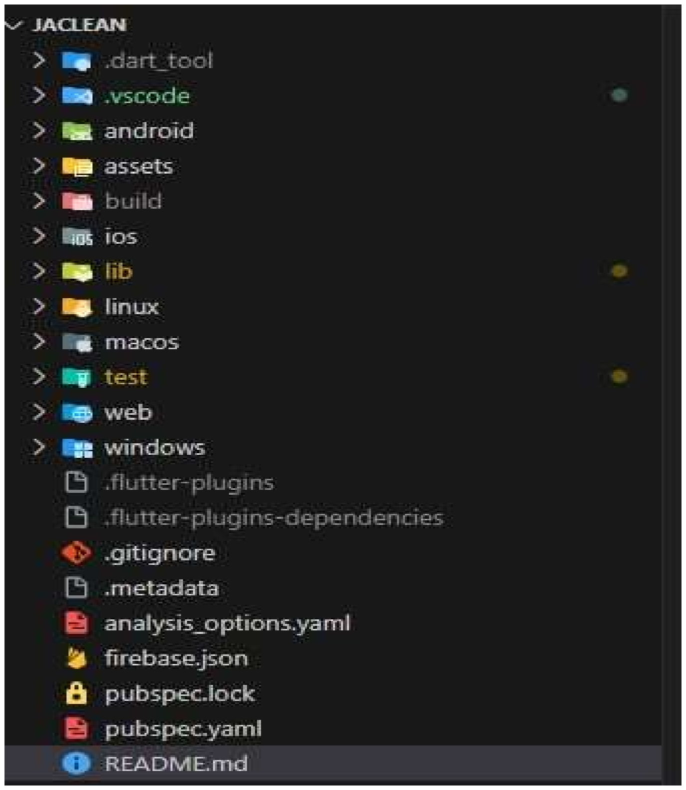
Figure 17. Flutter dependencies for the 9jaclean project

IMPLEMENTATION

After initializing the Flutter project, a clear and well-organized folder structure was established to ensure long-term maintainability and ease of development. As illustrated in Figure 18, the project root contains essential files like pubspec.yaml for dependency management and structured folders such as lib, assets, and test. The assets folder was specifically used to store necessary images and icons used throughout the application, ensuring consistent visual elements and a centralized reference point for media files.

Within the lib directory, the application logic was divided into subfolders, each handling specific functionalities such as business logic, user interface, and data handling. This structure followed the principle of separation of concerns, allowing different parts of the application to be managed and updated independently. To manage and manipulate data effectively, three model classes were created: Admin, Article, and Category. The Admin model is responsible for storing administrator credentials, such as an email address, which is later associated with the articles they create or edit in the Firestore database. The Article class includes factory constructors that enable easy conversion between Dart objects and JSON format, facilitating smooth communication between the local app and Cloud Firestore. The Category model was designed to manage article classifications on the home screen. When an admin views articles under a specific category and proceeds to add a new one, the app automatically references that selected category, eliminating the need for redundant input.

As shown in Figure 18 below, the 9jaclean application is structured using a modular file system to ensure the codebase remains organized, scalable, and easy to maintain. One of the core folders in the project is the services directory, which includes files responsible for implementing the BLoC (Business Logic Component) pattern, managing API calls, and handling local data storage using the SharedPreferences package. SharedPreferences provides a lightweight and efficient solution for saving simple key-value data locally on the device.



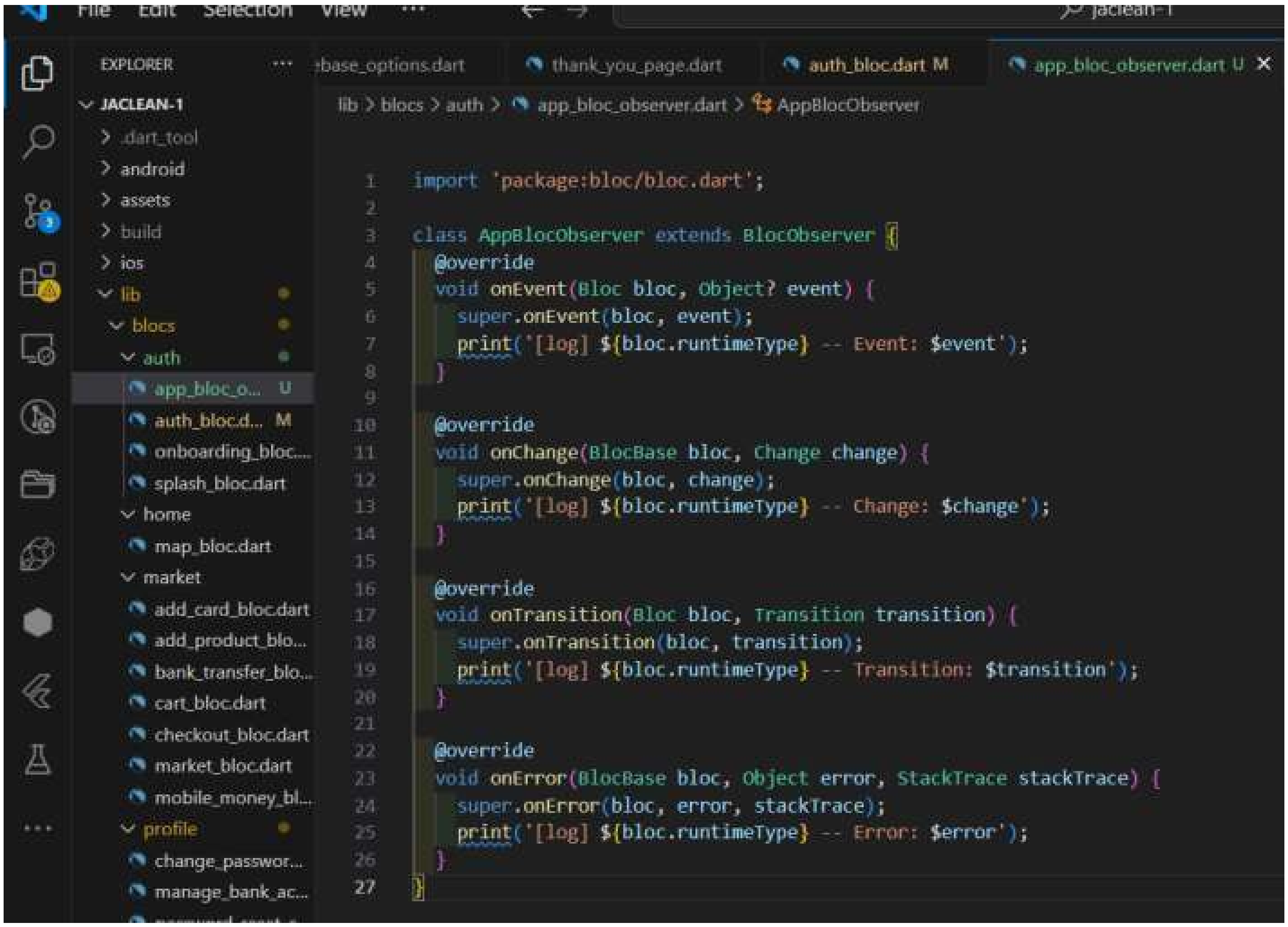
```
JACLEAN
  > .dart_tool
  > .vscode
  > android
  > assets
  > build
  > ios
  > lib
  > linux
  > macos
  > test
  > web
  > windows
    .flutter-plugins
    .flutter-plugins-dependencies
    .gitignore
    .metadata
    analysis_options.yaml
    firebase.json
    pubspec.lock
    pubspec.yaml
    README.md
```

Figure 18. Overall view of the project's file structure.

This is used in the app to store user preferences such as theme mode, allowing the interface to retain user settings between sessions. The utils folder contains helper functions, constant definitions, and custom widgets that are shared across the app, helping to maintain a consistent structure and reduce redundancy. In terms of user interface, the application separates its screens and UI components into a dedicated views directory. Inside this folder, smaller visual elements and reusable components are placed in the widgets subfolder, while the complete screens are kept in the pages folder. This separation of widgets and screens supports clean code principles and makes the application easier to scale and refactor.

To ensure a maintainable and testable architecture, the app follows the BLoC pattern’s three-layer approach. This includes the UI layer, which renders content and receives user input, the BLoC layer, which handles business logic and state transitions, and the data layer, which manages network calls or local storage. The BLoC acts as a bridge between the UI and data layers, listening to events triggered by the UI, performing logic, and emitting new states that the UI can react to (Islomov, 2022). This separation ensures a clean codebase where logic and presentation do not mix.

Additionally, a custom AuthBlocObserver was implemented to help monitor the application's state changes. As demonstrated in Figure 19, this observer provides visibility into each transition made by any bloc or cubit in the app. It is especially useful during debugging and helps verify that only authenticated users can access sensitive features such as managing articles. By observing the flow of state changes, developers can quickly identify and address potential bugs or inconsistencies during development.



```dart
import 'package:bloc/bloc.dart';

class AppBlocObserver extends BlocObserver {
  @override
  void onEvent(Bloc bloc, Object? event) {
    super.onEvent(bloc, event);
    print('[log] ${bloc.runtimeType} -- Event: $event');
  }

  @override
  void onChange(BlocBase bloc, Change change) {
    super.onChange(bloc, change);
    print('[log] ${bloc.runtimeType} -- Change: $change');
  }

  @override
  void onTransition(Bloc bloc, Transition transition) {
    super.onTransition(bloc, transition);
    print('[log] ${bloc.runtimeType} -- Transition: $transition');
  }

  @override
  void onError(BlocBase bloc, Object error, StackTrace stackTrace) {
    super.onError(bloc, error, stackTrace);
    print('[log] ${bloc.runtimeType} -- Error: $error');
  }
}
```

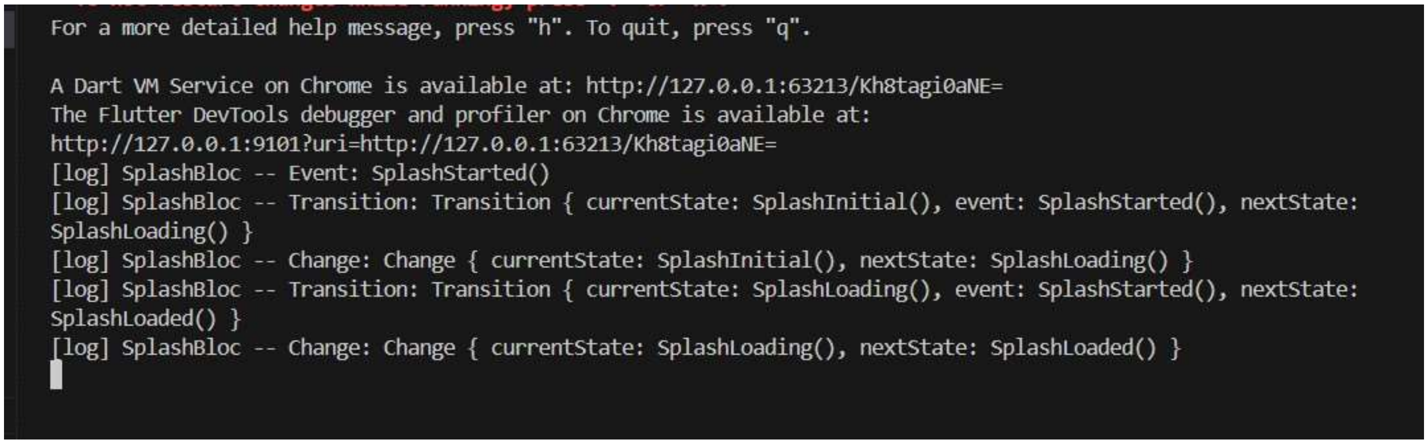
Figure 19. Customized AuthBloc Observer

As shown in **Figure 20**, the printed logs on the debug console display how the BLoC observer monitors and logs each state change throughout the lifecycle of the SplashBloc. The transition from SplashInitial() to SplashLoading() and eventually to SplashLoaded() is recorded step-by-step, providing a transparent view of the BLoC state management in action. This behavior confirms that the authentication and splash logic executes as expected, ensuring the application starts correctly and navigates the user based on initialization status.

This type of console output is made possible by the implementation of a custom BlocObserver in the project, which is especially valuable during debugging and testing. It helps developers trace back the flow of events, transitions, and resulting states, making it easier to identify issues or verify expected behavior.

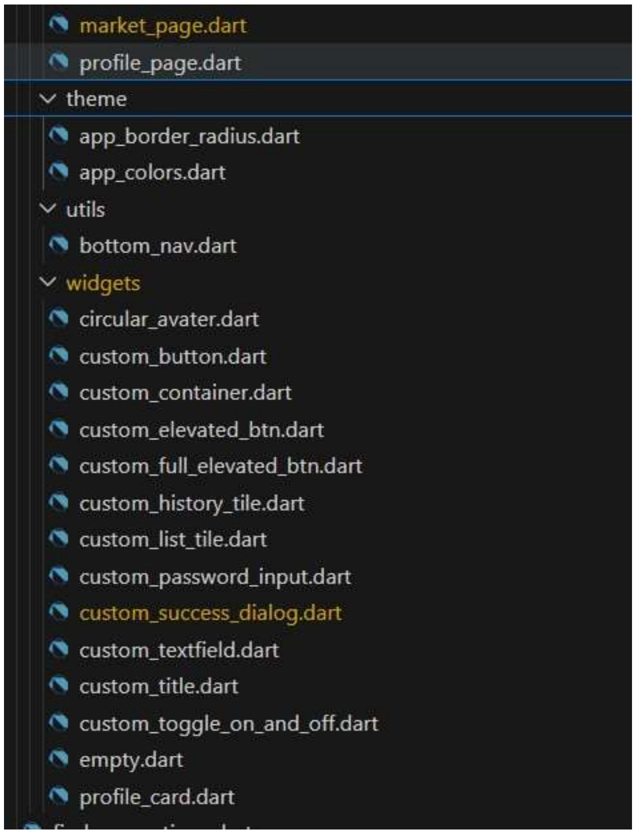
Furthermore, the observer also tracks state changes related to the ThemeCubit, which is responsible for applying the correct visual theme (light, dark, or system default) at the start of the app. The selected theme is saved to local storage using SharedPreferences, so that the user's preference is remembered and restored automatically the next time the app is launched. This enhances user experience by providing continuity in appearance without requiring re-selection.

Widgets are fundamental components in Flutter, forming the backbone of the application's interface and functionality. These widgets can be reused and customized to maintain consistency and efficiency across the app. In the 9jaclean project, all customized widgets and utility tools were organized within a dedicated custom folder, as shown in Figure 21. This folder serves as a central location for theming logic and reusable UI components.



```
For a more detailed help message, press "h". To quit, press "q".

A Dart VM Service on Chrome is available at: http://127.0.0.1:63213/Kh8tagi0aNE=
The Flutter DevTools debugger and profiler on Chrome is available at:
http://127.0.0.1:9101?uri=http://127.0.0.1:63213/Kh8tagi0aNE=
[log] SplashBloc -- Event: SplashStarted()
[log] SplashBloc -- Transition: Transition { currentState: SplashInitial(), event: SplashStarted(), nextState:
SplashLoading() }
[log] SplashBloc -- Change: Change { currentState: SplashInitial(), nextState: SplashLoading() }
[log] SplashBloc -- Transition: Transition { currentState: SplashLoading(), event: SplashStarted(), nextState:
SplashLoaded() }
[log] SplashBloc -- Change: Change { currentState: SplashLoading(), nextState: SplashLoaded() }
```

Figure 20. Customized Bloc observer log



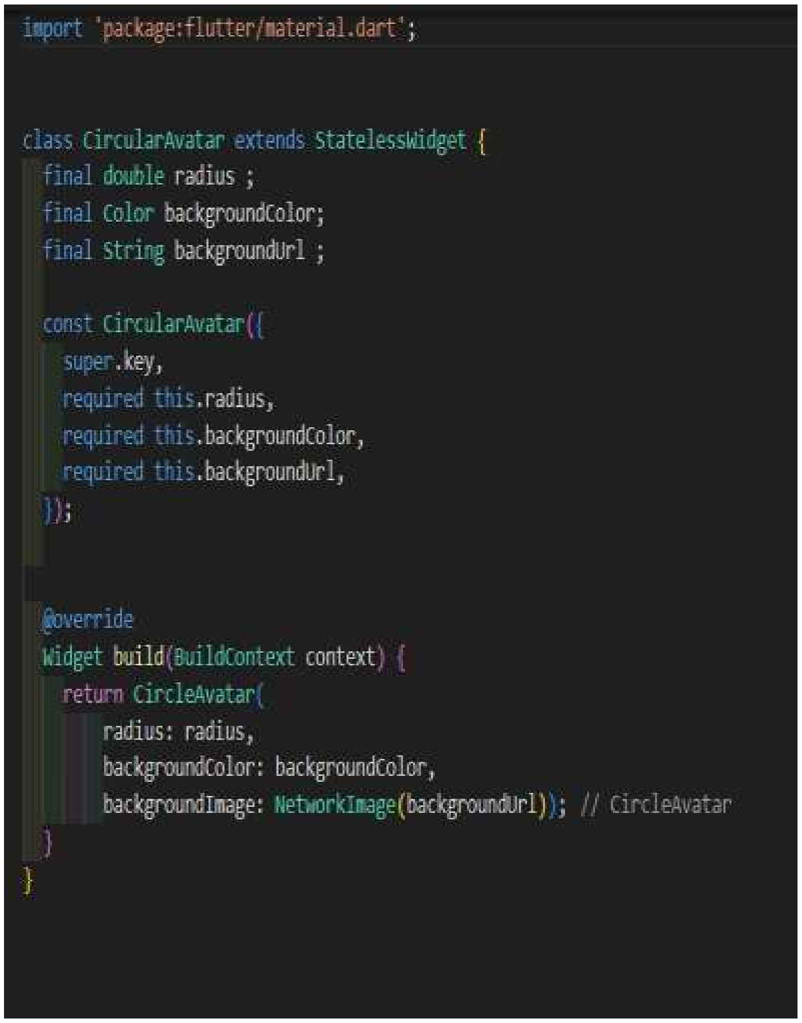
```
market_page.dart
profile_page.dart
theme
  app_border_radius.dart
  app_colors.dart
utils
  bottom_nav.dart
widgets
  circular_avater.dart
  custom_button.dart
  custom_container.dart
  custom_elevated_btn.dart
  custom_full_elevated_btn.dart
  custom_history_tile.dart
  custom_list_tile.dart
  custom_password_input.dart
  custom_success_dialog.dart
  custom_textfield.dart
  custom_title.dart
  custom_toggle_on_and_off.dart
  empty.dart
  profile_card.dart
```

Figure 21. Custom folder

Specifically, it includes theme configuration files for both light and dark modes, ensuring visual adaptability based on user preferences or system settings. Additionally, it houses utility widgets such as a customized word limit formatter, which enforces input restrictions in fields like the contact screen, enhancing both usability and data validation.

In the 9jaclean project, several widgets are used repeatedly across different screens, often with identical properties. Writing these widgets from scratch each time introduces unnecessary boilerplate code and increases the risk of inconsistencies in the user interface. To address this, reusable customized widgets were created with pre-defined styles and behaviors, significantly reducing redundancy and ensuring a consistent design throughout the application. This approach promotes both reusability and maintainability within the codebase as shown in Figure 22. Developers no longer need to manually reassign the same parameters each time a widget is instantiated, improving development efficiency. In the case of the 9jaclean project, one example of a reusable custom widget is the CircularAvatar shown in Figure 22. This widget was created to display a consistent circular avatar across various screens where user profiles or avatars are required. Instead of rewriting the same properties such as radius, background color, and background image each time a profile image is needed, the custom CircularAvatar class encapsulates this logic in a single reusable component.

The constructor of the custom CircularAvatar widget has been implemented with flexibility in mind. It accepts several parameters, some of which are optional and marked with the required keyword to ensure proper usage. These parameters include radius, backgroundColor, and backgroundUrl, allowing the avatar to be styled and populated dynamically based on user data.



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


class CircularAvatar extends StatelessWidget {
  final double radius ;
  final Color backgroundColor;
  final String backgroundUrl ;

  const CircularAvatar({
    super.key,
    required this.radius,
    required this.backgroundColor,
    required this.backgroundUrl,
  });


  @override
  Widget build(BuildContext context) {
    return CircleAvatar(
        radius: radius,
        backgroundColor: backgroundColor,
        backgroundImage: NetworkImage(backgroundUrl)); // CircleAvatar
  }
}
```

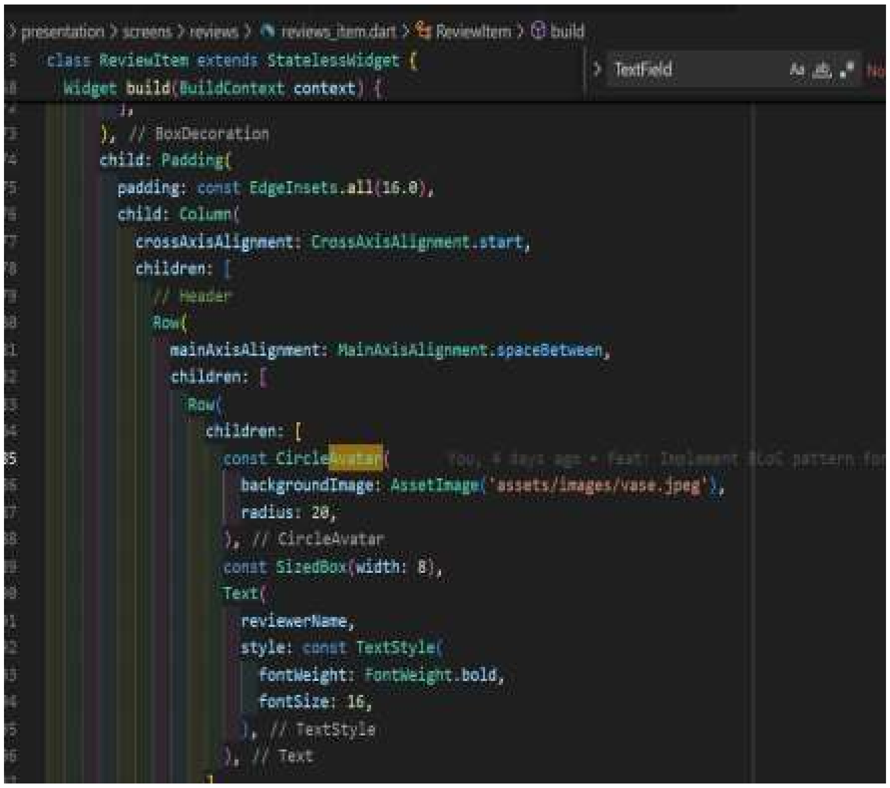
Figure 22. Optional and required parameters of the customized list tile field widget.

This widget is especially useful when displaying user profiles, such as on the dashboard or review screens. By customizing properties like size and color, and dynamically passing image URLs, the same widget can serve multiple purposes throughout the application. This design eliminates the need to repeatedly configure avatar settings manually and instead enables consistent styling and faster development. The use of a stateless widget also ensures that the component remains lightweight and efficient in performance.

As shown in Figure 23, the CircleAvatar widget has been customized and used consistently across different parts of the application, such as the review section. Without a reusable and standardized avatar implementation, the codebase would quickly become cluttered with repetitive configurations for radius, background images, and layout properties. By utilizing a common widget or structure like CircleAvatar, the project avoids unnecessary code duplication and ensures a uniform design experience. This approach improves maintainability, keeps the UI consistent, and reduces the chances of visual mismatches across different screens where user profile images or icons are displayed.

The custom circular avatar is reused multiple times, it can pass distinct arguments, which proves that it not only brings the consistency of the design but also the flexibility of the design so that developers can set and alter the properties as they want. For example, several text form fields only have one line while some have more than one line. Also, there are different validators for the application to check if the administrators enter valid information or not.

The custom circular avatar widget helps keep the design of the app clean and consistent. It is used to display each reviewer’s profile picture in the reviews section. Even though the same widget is reused in different places, it can show different images and sizes depending on what is needed. For example, some avatars might have a different background color or radius, but they all follow the same format, which makes the app look neat and well-designed.



```
> presentation > screens > reviews > reviews_item.dart > ReviewItem > build
class ReviewItem extends StatelessWidget {
  Widget build(BuildContext context) {
          ),
        ), // BoxDecoration
        child: Padding(
          padding: const EdgeInsets.all(16.0),
          child: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
              // Header
              Row(
                mainAxisAlignment: MainAxisAlignment.spaceBetween,
                children: [
                  Row(
                    children: [
                      const CircleAvatar(      You, 4 days ago • feat: Implement BLoC pattern for
                        backgroundImage: AssetImage('assets/images/vase.jpeg'),
                        radius: 20,
                      ), // CircleAvatar
                      const SizedBox(width: 8),
                      Text(
                        reviewerName,
                        style: const TextStyle(
                          fontWeight: FontWeight.bold,
                          fontSize: 16,
                        ), // TextStyle
                      ), // Text
                    ]
```

Figure 23. User Avartar customization

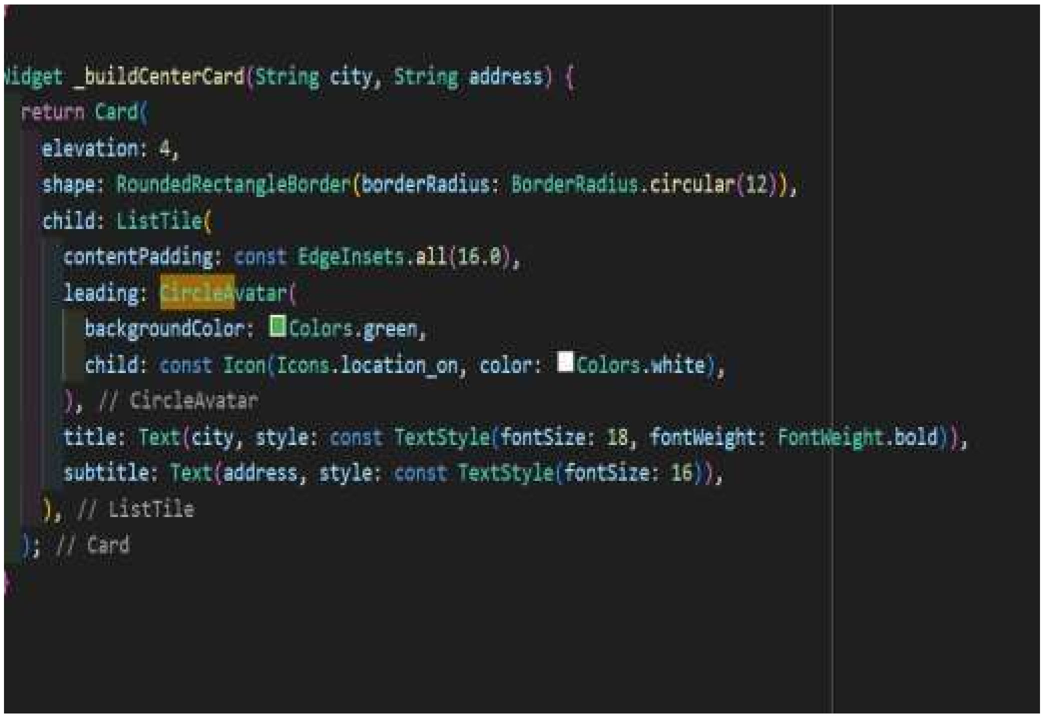
As shown in Figure 24 below, using a custom widget like CircularAvatar made the development process much simpler, faster, and more efficient. Instead of writing out the same CircleAvatar code every single time an avatar was needed, the development team only had to define the logic and styling once inside a reusable widget. After that, they could simply reuse this widget across the entire app whether in the reviews section, profile cards, or any other screen that required a user image. This saved a lot of time, reduced code duplication, and kept the codebase much cleaner and easier to manage.

The biggest advantage of doing it this way is flexibility. For example, if the design team decides later to change the radius of the avatar or add a default image in case one fails to load, it only takes one update in the CircularAvatar file and that change reflects instantly everywhere the widget is used. This not only speeds up future updates, but also helps avoid inconsistencies in the design. Everything stays uniform, giving the app a more polished and professional look.

Figure 25 highlights how this custom widget contributes to consistent UI across the app. You'll notice that the same CircularAvatar is used to display user images in the review cards, creating a unified design. This same idea is carried over to the article features too both the add and edit article screens use the same text fields and layout widgets instead of building separate screens from scratch. This approach saves development time and helps prevent bugs caused by inconsistent behavior between similar features.

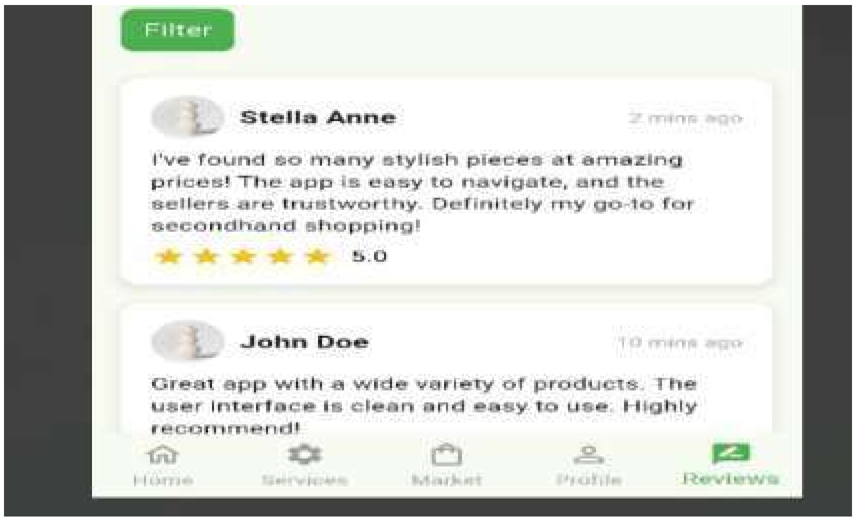
Overall, while the custom CircularAvatar might seem like a small part of the project, it plays a key role in ensuring a smooth and visually consistent user experience. It's one of those little tools that, when done right, makes a big difference in how easy the app is to use and how good it feels to interact with.

BlocConsumer is a combination of two important widgets from the flutter_bloc package: BlocListener and BlocBuilder. It allows developers to both listen for changes in state and rebuild parts of the UI in response to those changes (Wogu, 2022). This dual functionality makes BlocConsumer particularly useful for handling actions like showing snackbars, navigating between screens, and displaying dynamic UI updates such as loading indicators.



```
idget _buildCenterCard(String city, String address) {
  return Card(
    elevation: 4,
    shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(12)),
    child: ListTile(
      contentPadding: const EdgeInsets.all(16.0),
      leading: CircleAvatar(
        backgroundColor: Colors.green,
        child: const Icon(Icons.location_on, color: Colors.white),
      ), // CircleAvatar
      title: Text(city, style: const TextStyle(fontSize: 18, fontWeight: FontWeight.bold)),
      subtitle: Text(address, style: const TextStyle(fontSize: 16)),
    ), // ListTile
  ); // Card
```

Figure 24. Customized widgets implementation



Filter

Stella Anne 2 mins ago

I've found so many stylish pieces at amazing prices! The app is easy to navigate, and the sellers are trustworthy. Definitely my go-to for secondhand shopping!

★★★★★ 5.0

John Doe 10 mins ago

Great app with a wide variety of products. The user interface is clean and easy to use. Highly recommend!

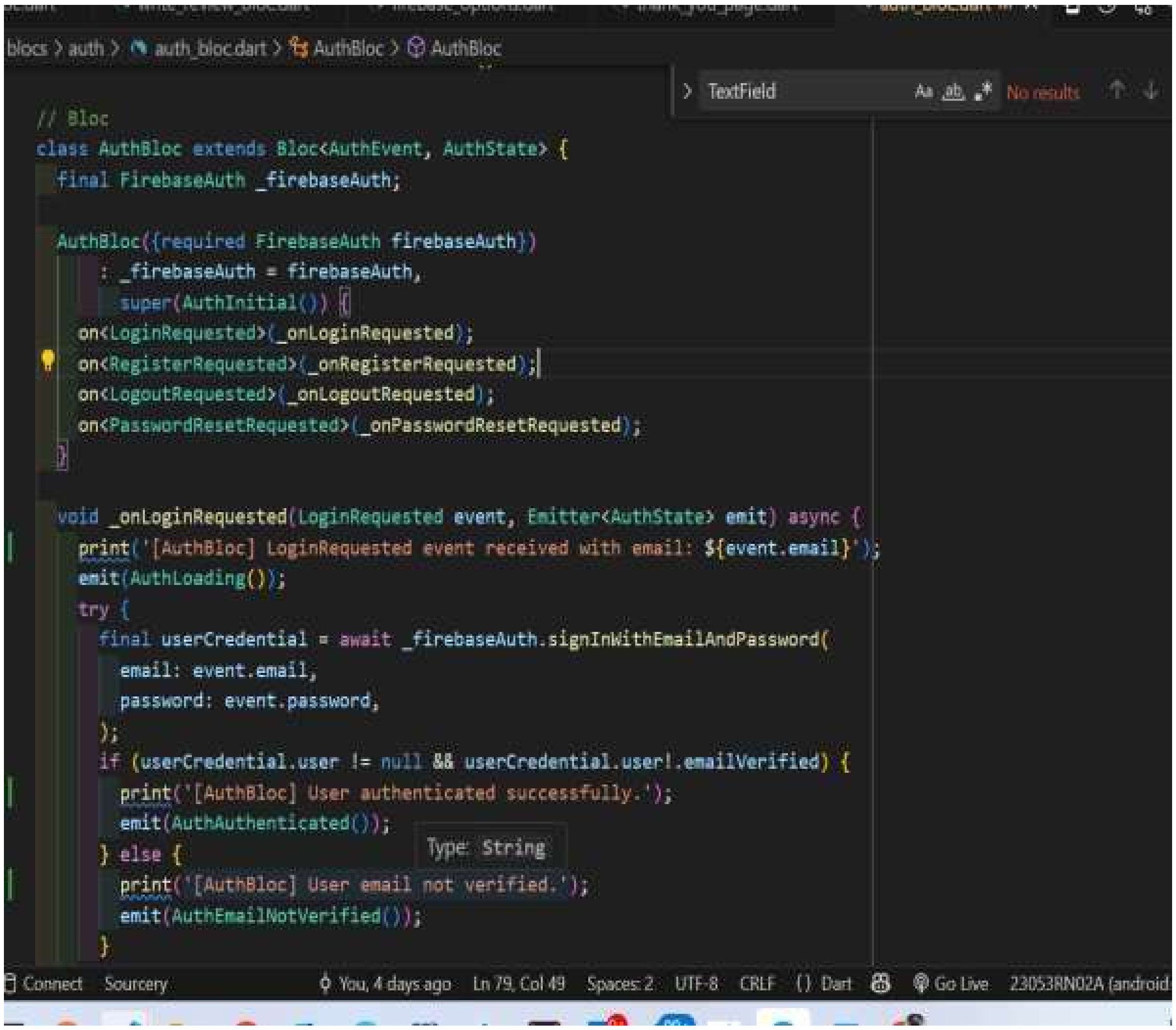
Figure 25. Review Section screen

In the context of the 9jaclean app, BlocConsumer is used extensively during the authentication process, which is managed through a custom AuthBloc class. This Bloc handles several events such as LoginRequested, RegisterRequested, LogoutRequested, and PasswordResetRequested, as seen in Figure 26. Each event is wired to a specific method like _onLoginRequested, which is responsible for verifying credentials using Firebase Authentication.

When a login attempt is made, the LoginRequested event is dispatched to the Bloc. Inside the _onLoginRequested method, Firebase checks the email and password using signInWithEmailAndPassword. If the credentials are correct and the email is verified, the Bloc emits an AuthAuthenticated state. If not, it emits an AuthEmailNotVerified or other error state.

The listener part of the BlocConsumer observes these state changes and performs actions like navigating to the home screen or showing error dialogs. The builder section of the BlocConsumer updates the UI based on the state such as showing a loading spinner when the state is AuthLoading, or re-enabling the login button once authentication is complete. This ensures the user receives immediate feedback during authentication without blocking the UI.

According to Felangel (2023), this separation of logic and UI offered by BlocConsumer promotes clean architecture and improves testability. By organizing state transitions through AuthBloc, the project maintains a clear flow of logic, improves maintainability, and avoids mixing business logic with UI components.



blocs > auth > auth_bloc.dart > AuthBloc > AuthBloc

```
// Bloc
class AuthBloc extends Bloc<AuthEvent, AuthState> {
  final FirebaseAuth _firebaseAuth;

  AuthBloc({required FirebaseAuth firebaseAuth})
      : _firebaseAuth = firebaseAuth,
        super(AuthInitial()) {
    on<LoginRequested>(_onLoginRequested);
    on<RegisterRequested>(_onRegisterRequested);
    on<LogoutRequested>(_onLogoutRequested);
    on<PasswordResetRequested>(_onPasswordResetRequested);
  }

  void _onLoginRequested(LoginRequested event, Emitter<AuthState> emit) async {
    print('[AuthBloc] LoginRequested event received with email: ${event.email}');
    emit(AuthLoading());
    try {
      final userCredential = await _firebaseAuth.signInWithEmailAndPassword(
        email: event.email,
        password: event.password,
      );
      if (userCredential.user != null && userCredential.user!.emailVerified) {
        print('[AuthBloc] User authenticated successfully.');
        emit(AuthAuthenticated());
      } else {
        print('[AuthBloc] User email not verified.');
        emit(AuthEmailNotVerified());
      }
```

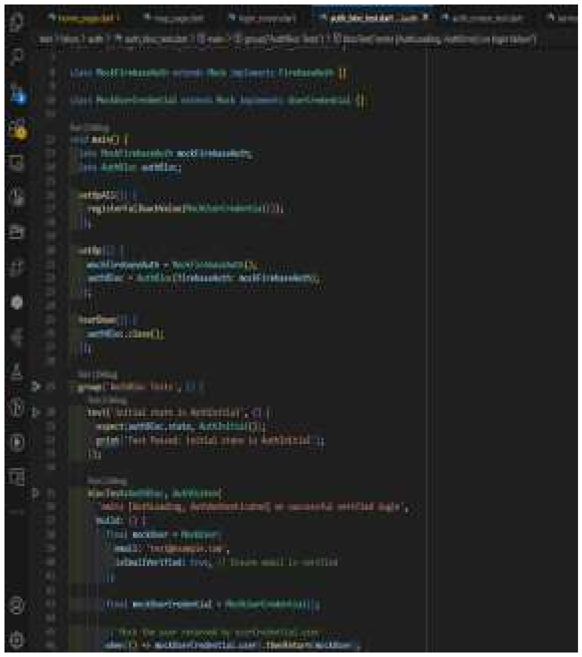
Figure 26. BLoC Consumer

TESTING

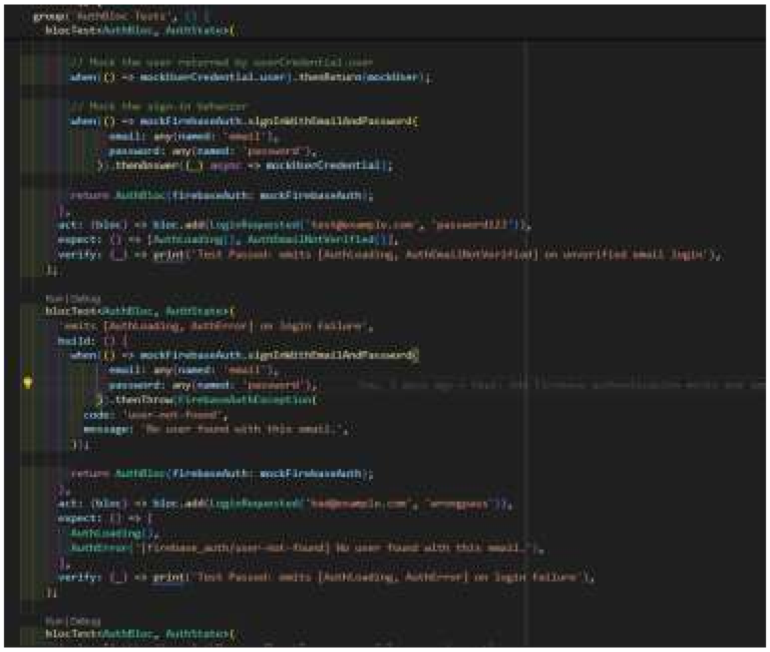
Testing is a critical part of building stable and reliable applications. For the 9jaclean project, tests were written specifically to verify that the authentication process worked as expected under various conditions. Since Bloc separates the business logic from the user interface, it becomes much easier to test logic independently, making it a highly preferred architecture for Flutter apps. With the help of flutter_test and bloc_test libraries, the Bloc logic was tested by simulating user actions and verifying that the appropriate states were emitted in sequence (Wogu, 2022).

In this project, most of the testing effort was directed toward the AuthBloc. As illustrated in Figures 27 several unit tests were written to cover the core aspects of authentication. These include verifying that the initial state of the bloc is AuthInitial, checking that a successful login attempt emits AuthLoading followed by AuthAuthenticated, and ensuring that login attempts with an unverified email trigger the AuthEmailNotVerified state. Other cases such as login with invalid credentials were tested to make sure they emitted AuthError, providing clear feedback to the user when something goes wrong.

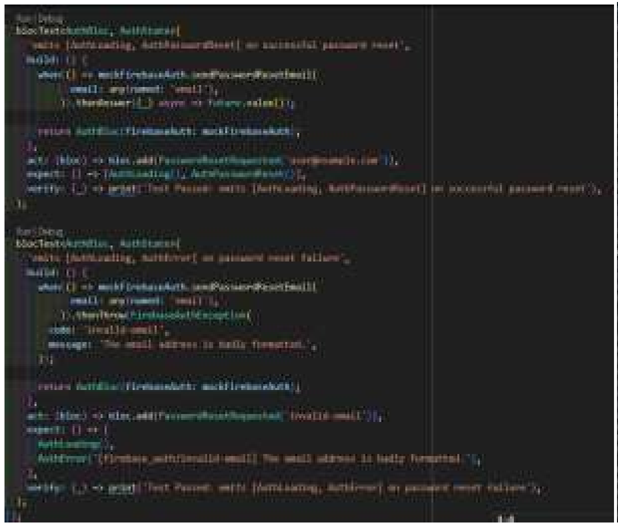
Mocks were created for Firebase authentication methods using MockFirebaseAuth and MockUserCredential. This allowed testing the authentication logic without needing to connect to an actual Firebase backend. For example, when a login event is triggered using mocked credentials, the test checks if the email is verified and responds accordingly. This approach ensures that all edge cases are covered, whether it’s a successful sign-in or a failed attempt due to invalid email formatting or incorrect password (Google Flutter, 2023).



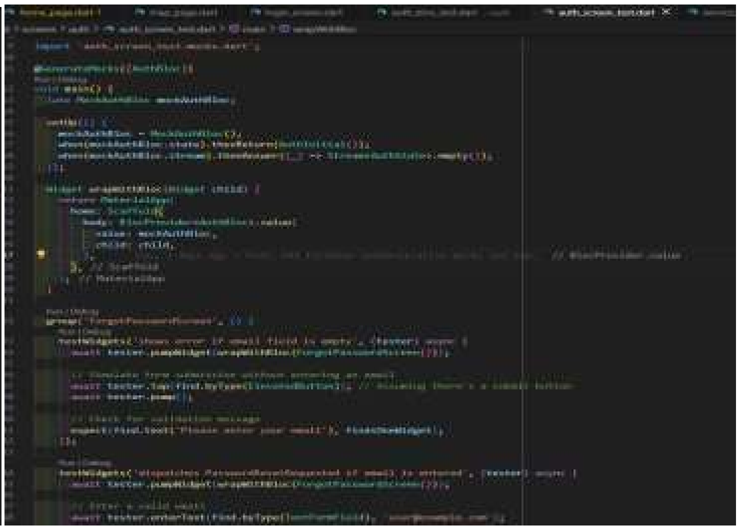
```
class MockFirebaseAuth extends Mock implements FirebaseAuth {}

class MockUserCredential extends Mock implements UserCredential {}

Run | Debug
void main() {
  late MockFirebaseAuth mockFirebaseAuth;
  late AuthBloc authBloc;

  setUpAll(() {
    registerFallbackValue(MockUserCredential());
  });

  setUp(() {
    mockFirebaseAuth = MockFirebaseAuth();
    authBloc = AuthBloc(firebaseAuth: mockFirebaseAuth);
  });

  tearDown(() {
    authBloc.close();
  });

  Run | Debug
  group('AuthBloc Tests', () {
    Run | Debug
    test('initial state is AuthInitial', () {
      expect(authBloc.state, AuthInitial());
      print('Test Passed: initial state is AuthInitial');
    });

    Run | Debug
    blocTest<AuthBloc, AuthState>(
      'emits [AuthLoading, AuthAuthenticated] on successful verified login',
      build: () {
        final mockUser = MockUser(
          email: 'test@example.com',
          isEmailVerified: true, // Ensure email is verified
        );

        final mockUserCredential = MockUserCredential();

        // Mock the user returned by userCredential.user
        when(() => mockUserCredential.user).thenReturn(mockUser);
```



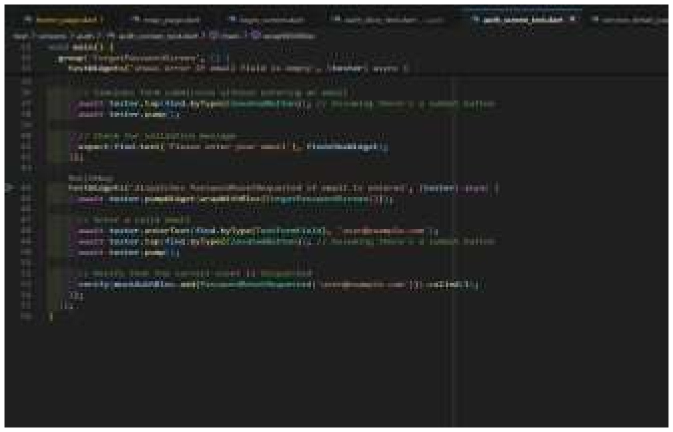
```
group('AuthBloc Tests', () {
  blocTest<AuthBloc, AuthState>(

      // Mock the user returned by userCredential.user
      when(() => mockUserCredential.user).thenReturn(mockUser);

      // Mock the sign-in behavior
      when(() => mockFirebaseAuth.signInWithEmailAndPassword(
            email: any(named: 'email'),
            password: any(named: 'password'),
          )).thenAnswer((_) async => mockUserCredential);

      return AuthBloc(firebaseAuth: mockFirebaseAuth);
    },
    act: (bloc) => bloc.add(LoginRequested('test@example.com', 'password123')),
    expect: () => [AuthLoading(), AuthEmailNotVerified()],
    verify: (_) => print('Test Passed: emits [AuthLoading, AuthEmailNotVerified] on unverified email login'),
  );

  Run | Debug
  blocTest<AuthBloc, AuthState>(
    'emits [AuthLoading, AuthError] on login failure',
    build: () {
      when(() => mockFirebaseAuth.signInWithEmailAndPassword(
            email: any(named: 'email'),
            password: any(named: 'password'),
          )).thenThrow(FirebaseAuthException(
        code: 'user-not-found',
        message: 'No user found with this email.',
      ));

      return AuthBloc(firebaseAuth: mockFirebaseAuth);
    },
    act: (bloc) => bloc.add(LoginRequested('bad@example.com', 'wrongpass')),
    expect: () => [
      AuthLoading(),
      AuthError('[firebase_auth/user-not-found] No user found with this email.'),
    ],
    verify: (_) => print('Test Passed: emits [AuthLoading, AuthError] on login failure'),
  );

  Run | Debug
  blocTest<AuthBloc, AuthState>(
```



```
Run | Debug
blocTest<AuthBloc, AuthState>(
  'emits [AuthLoading, AuthPasswordReset] on successful password reset',
  build: () {
    when(() => mockFirebaseAuth.sendPasswordResetEmail(
          email: any(named: 'email'),
        )).thenAnswer((_) async => Future.value());

    return AuthBloc(firebaseAuth: mockFirebaseAuth);
  },
  act: (bloc) => bloc.add(PasswordResetRequested('user@example.com')),
  expect: () => [AuthLoading(), AuthPasswordReset()],
  verify: (_) => print('Test Passed: emits [AuthLoading, AuthPasswordReset] on successful password reset'),
);

Run | Debug
blocTest<AuthBloc, AuthState>(
  'emits [AuthLoading, AuthError] on password reset failure',
  build: () {
    when(() => mockFirebaseAuth.sendPasswordResetEmail(
          email: any(named: 'email'),
        )).thenThrow(FirebaseAuthException(
      code: 'invalid-email',
      message: 'The email address is badly formatted.',
    ));

    return AuthBloc(firebaseAuth: mockFirebaseAuth);
  },
  act: (bloc) => bloc.add(PasswordResetRequested('invalid-email')),
  expect: () => [
    AuthLoading(),
    AuthError('[firebase_auth/invalid-email] The email address is badly formatted.'),
  ],
  verify: (_) => print('Test Passed: emits [AuthLoading, AuthError] on password reset failure'),
);
});
```



```
import 'auth_screen_test.mocks.dart';

@GenerateMocks([AuthBloc])
Run | Debug
void main() {
  late MockAuthBloc mockAuthBloc;

  setUp(() {
    mockAuthBloc = MockAuthBloc();
    when(mockAuthBloc.state).thenReturn(AuthInitial());
    when(mockAuthBloc.stream).thenAnswer((_) => Stream<AuthState>.empty());
  });

  Widget wrapWithBloc(Widget child) {
    return MaterialApp(
      home: Scaffold(
        body: BlocProvider<AuthBloc>.value(
          value: mockAuthBloc,
          child: child,
        ), // BlocProvider.value
      ), // Scaffold
    ); // MaterialApp
  }

  Run | Debug
  group('ForgotPasswordScreen', () {
    Run | Debug
    testWidgets('shows error if email field is empty', (tester) async {
      await tester.pumpWidget(wrapWithBloc(ForgotPasswordScreen()));

      // Simulate form submission without entering an email
      await tester.tap(find.byType(ElevatedButton)); // Assuming there's a submit button
      await tester.pump();

      // Check for validation message
      expect(find.text('Please enter your email'), findsOneWidget);
    });

    Run | Debug
    testWidgets('dispatches PasswordResetRequested if email is entered', (tester) async {
      await tester.pumpWidget(wrapWithBloc(ForgotPasswordScreen()));

      // Enter a valid email
      await tester.enterText(find.byType(TextFormField), 'user@example.com');
```



```
void main() {
  group('ForgotPasswordScreen', () {
    testWidgets('shows error if email field is empty', (tester) async {

      // Simulate form submission without entering an email
      await tester.tap(find.byType(ElevatedButton)); // Assuming there's a submit button
      await tester.pump();

      // Check for validation message
      expect(find.text('Please enter your email'), findsOneWidget);
    });

    Run | Debug
    testWidgets('dispatches PasswordResetRequested if email is entered', (tester) async {
      await tester.pumpWidget(wrapWithBloc(ForgotPasswordScreen()));

      // Enter a valid email
      await tester.enterText(find.byType(TextFormField), 'user@example.com');
      await tester.tap(find.byType(ElevatedButton)); // Assuming there's a submit button
      await tester.pump();

      // Verify that the correct event is dispatched
      verify(mockAuthBloc.add(PasswordResetRequested('user@example.com'))).called(1);
    });
  });
}
```

Figure 27: Tests cases for AuthBloc

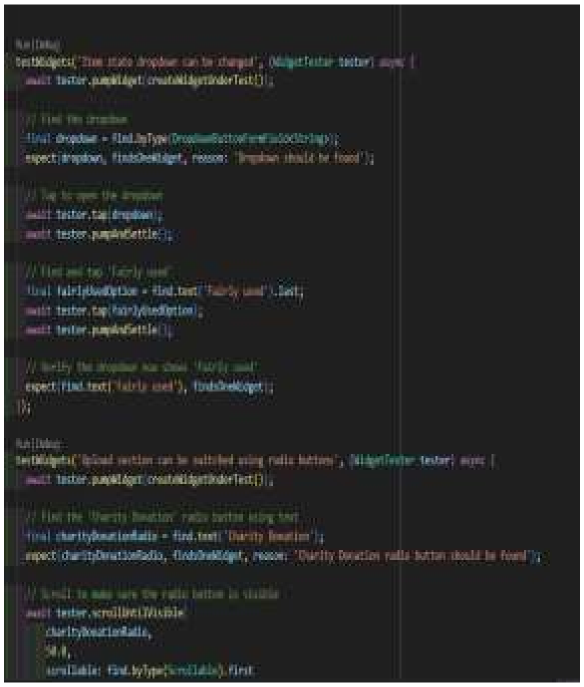
For the Marketplace and Review modules, Flutter widget testing was implemented to ensure expected UI behaviors and user interaction flow. Similar to how BLoC uses act and expect to validate state transitions, these widget tests simulate user actions and check visual and functional outcomes using the tester API in flutter_test.

As shown in Figures 28, the testWidgets function wraps the test scenario, while the tester.pumpWidget() method is used to render the desired screen. Within these tests, the actions such as tapping buttons, selecting from dropdowns, or entering text mimic how users would interact with the UI. Just like blocTest expects a certain sequence of emitted states, widget tests expect certain widgets or text to appear as a result of those interactions.

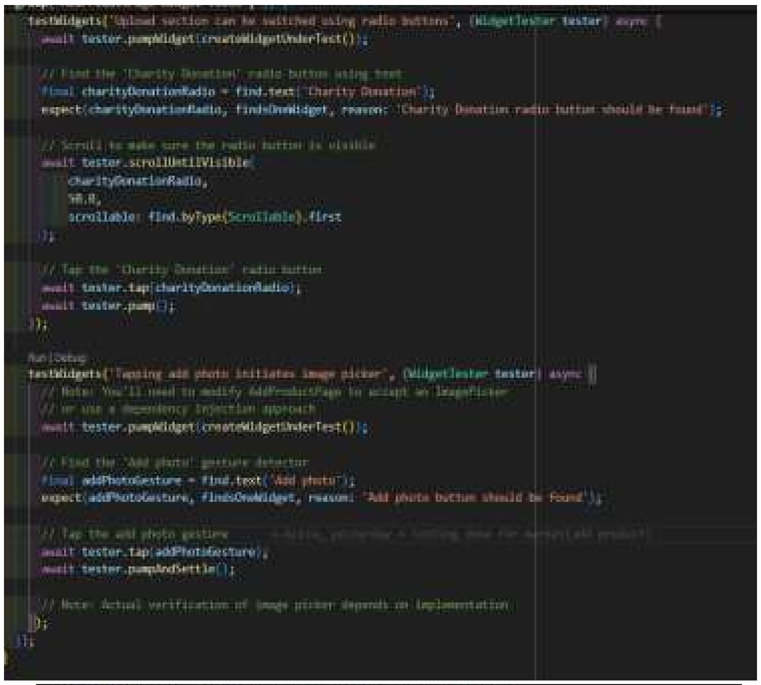
For example, in the marketplace test case, the dropdown field for item condition is interacted with by simulating a tap, selecting “Fairly used,” and then confirming that this selection is correctly reflected. The expected result (expect(find.text('Fairly used'), findsOneWidget)) verifies that the UI updated with the chosen value. This mirrors the concept of state matching in bloc tests only here, the expectation is for visible widgets.

Similarly, radio button interactions are validated by first scrolling to ensure visibility and then simulating taps. The test confirms that the correct radio option ("Charity Donation") appears on the screen. In another case, tapping the "Add Photo" section triggers an image picker gesture. While the actual image picker verification depends on the implementation, the test ensures the button exists and responds correctly when tapped.

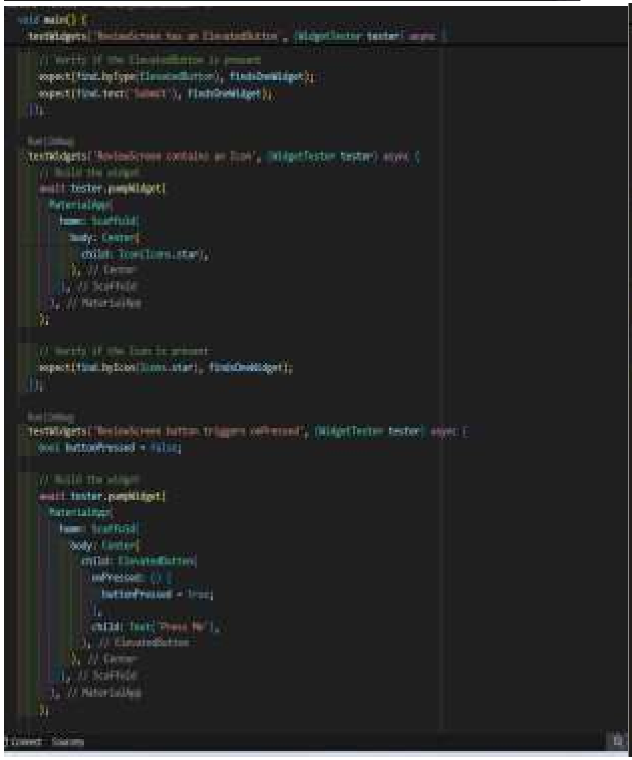
For the Review module, several interaction-based test cases are created. One verifies that an ElevatedButton is rendered, while another confirms that pressing this button changes the state of a boolean flag (buttonPressed = true). A more complex example involves showing a modal bottom sheet upon pressing a button. This is validated by checking for the presence of the bottom sheet content after the tap interaction, similar to how in bloc tests, we’d assert for a specific state being emitted after an event.



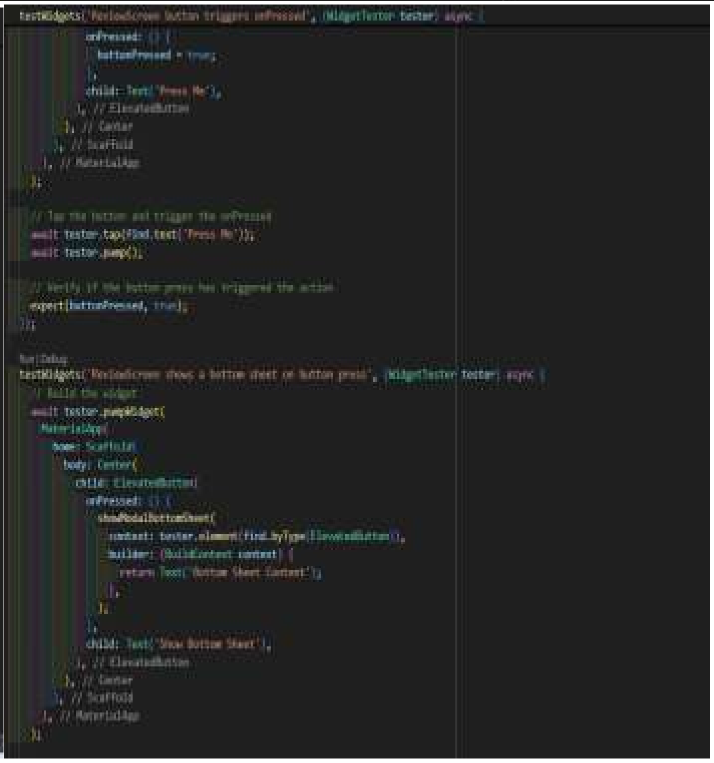
```
Run | Debug
testWidgets('Item state dropdown can be changed', (WidgetTester tester) async {
  await tester.pumpWidget(createWidgetUnderTest());

  // Find the dropdown
  final dropdown = find.byType(DropdownButtonFormField<String>);
  expect(dropdown, findsOneWidget, reason: 'Dropdown should be found');

  // Tap to open the dropdown
  await tester.tap(dropdown);
  await tester.pumpAndSettle();

  // Find and tap 'Fairly used'
  final fairlyUsedOption = find.text('Fairly used').last;
  await tester.tap(fairlyUsedOption);
  await tester.pumpAndSettle();

  // Verify the dropdown now shows 'Fairly used'
  expect(find.text('Fairly used'), findsOneWidget);
});

Run | Debug
testWidgets('Upload section can be switched using radio buttons', (WidgetTester tester) async {
  await tester.pumpWidget(createWidgetUnderTest());

  // Find the 'Charity Donation' radio button using text
  final charityDonationRadio = find.text('Charity Donation');
  expect(charityDonationRadio, findsOneWidget, reason: 'Charity Donation radio button should be found');

  // Scroll to make sure the radio button is visible
  await tester.scrollUntilVisible(
    charityDonationRadio,
    50.0,
    scrollable: find.byType(Scrollable).first
```



```
testWidgets('Upload section can be switched using radio buttons', (WidgetTester tester) async {
  await tester.pumpWidget(createWidgetUnderTest());

  // Find the 'Charity Donation' radio button using text
  final charityDonationRadio = find.text('Charity Donation');
  expect(charityDonationRadio, findsOneWidget, reason: 'Charity Donation radio button should be found');

  // Scroll to make sure the radio button is visible
  await tester.scrollUntilVisible(
    charityDonationRadio,
    50.0,
    scrollable: find.byType(Scrollable).first
  );

  // Tap the 'Charity Donation' radio button
  await tester.tap(charityDonationRadio);
  await tester.pump();
});

Run | Debug
testWidgets('Tapping add photo initiates image picker', (WidgetTester tester) async {
  // Note: You'll need to modify AddProductPage to accept an ImagePicker
  // or use a dependency injection approach
  await tester.pumpWidget(createWidgetUnderTest());

  // Find the 'Add photo' gesture detector
  final addPhotoGesture = find.text('Add photo');
  expect(addPhotoGesture, findsOneWidget, reason: 'Add photo button should be found');

  // Tap the add photo gesture      [illegible]
  await tester.tap(addPhotoGesture);
  await tester.pumpAndSettle();

  // Note: Actual verification of image picker depends on implementation
});
});
```



```
void main() {
  testWidgets('ReviewScreen has an ElevatedButton', (WidgetTester tester) async {
    // Verify if the ElevatedButton is present
    expect(find.byType(ElevatedButton), findsOneWidget);
    expect(find.text('Submit'), findsOneWidget);
  });

  Run | Debug
  testWidgets('ReviewScreen contains an Icon', (WidgetTester tester) async {
    // Build the widget
    await tester.pumpWidget(
      MaterialApp(
        home: Scaffold(
          body: Center(
            child: Icon(Icons.star),
          ), // Center
        ), // Scaffold
      ), // MaterialApp
    );

    // Verify if the Icon is present
    expect(find.byIcon(Icons.star), findsOneWidget);
  });

  Run | Debug
  testWidgets('ReviewScreen button triggers onPressed', (WidgetTester tester) async {
    bool buttonPressed = false;

    // Build the widget
    await tester.pumpWidget(
      MaterialApp(
        home: Scaffold(
          body: Center(
            child: ElevatedButton(
              onPressed: () {
                buttonPressed = true;
              },
              child: Text('Press Me'),
            ), // ElevatedButton
          ), // Center
        ), // Scaffold
      ), // MaterialApp
    );
```



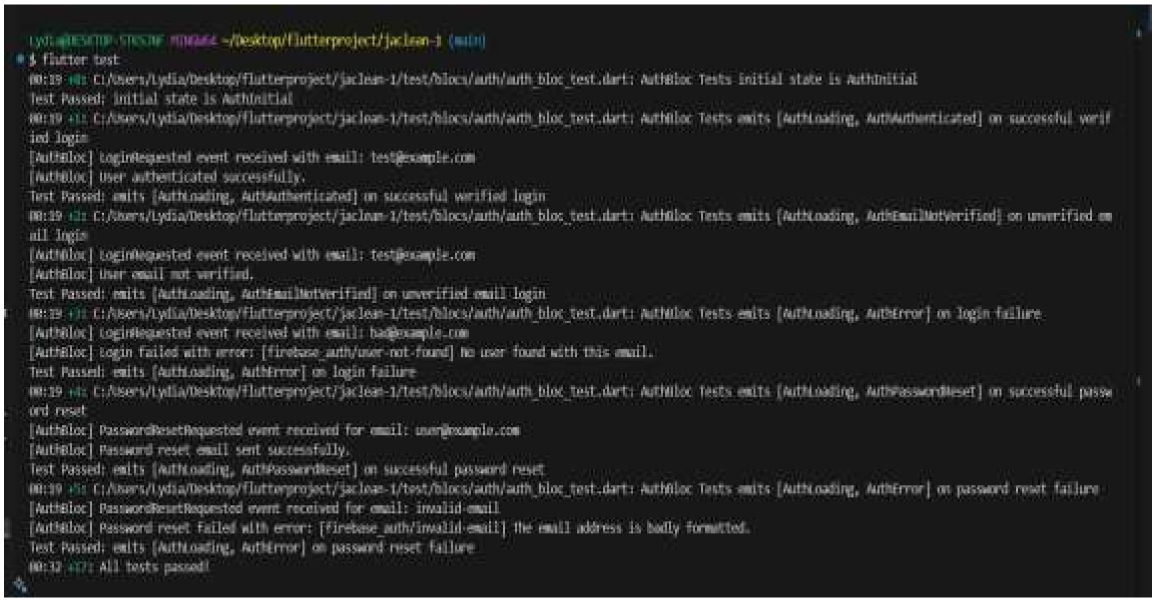
```
testWidgets('ReviewScreen button triggers onPressed', (WidgetTester tester) async {
            onPressed: () {
              buttonPressed = true;
            },
            child: Text('Press Me'),
          ), // ElevatedButton
        ), // Center
      ), // Scaffold
    ), // MaterialApp
  );

  // Tap the button and trigger the onPressed
  await tester.tap(find.text('Press Me'));
  await tester.pump();

  // Verify if the button press has triggered the action
  expect(buttonPressed, true);
});

Run | Debug
testWidgets('ReviewScreen shows a bottom sheet on button press', (WidgetTester tester) async {
  // Build the widget
  await tester.pumpWidget(
    MaterialApp(
      home: Scaffold(
        body: Center(
          child: ElevatedButton(
            onPressed: () {
              showModalBottomSheet(
                context: tester.element(find.byType(ElevatedButton)),
                builder: (BuildContext context) {
                  return Text('Bottom Sheet Content');
                },
              );
            },
            child: Text('Show Bottom Sheet'),
          ), // ElevatedButton
        ), // Center
      ), // Scaffold
    ), // MaterialApp
  );
```

Figure 28. Test cases for Add product and Review Screen

After implementing the necessary test cases to validate both authentication logic and theming functionality in the 9jaclean application, the tests were executed to confirm expected behaviors. As displayed in Figure 29, the tests were triggered using the built-in Dart and Flutter testing features in Visual Studio Code. These can be run either by pressing the green play icon next to individual test methods or by using extensions like "Test Explorer UI" to manage and run multiple tests at once.

In the terminal output, we can observe the progression of each test, including login attempts with various conditions such as successful login, unverified emails, and incorrect credentials as well as password reset scenarios. Each step logs internal state changes, such as when the AuthBloc transitions from AuthLoading to either AuthAuthenticated, AuthError, or other relevant states. This ensures that our state management is predictable and handles each edge case accordingly.

All seventeen test cases shown passed successfully, indicated by the final summary message “All tests passed!” and green checkmarks. This result builds confidence that the AuthBloc handles login and password reset flows robustly and as intended, contributing to a reliable user experience.



```
Lydia@DESKTOP-[illegible] MINGW64 ~/Desktop/flutterproject/jaclean-1 (main)
$ flutter test
00:19 +0: C:/Users/Lydia/Desktop/flutterproject/jaclean-1/test/blocs/auth/auth_bloc_test.dart: AuthBloc Tests initial state is AuthInitial
Test Passed: initial state is AuthInitial
00:19 +1: C:/Users/Lydia/Desktop/flutterproject/jaclean-1/test/blocs/auth/auth_bloc_test.dart: AuthBloc Tests emits [AuthLoading, AuthAuthenticated] on successful verif
ied login
[AuthBloc] LoginRequested event received with email: test@example.com
[AuthBloc] User authenticated successfully.
Test Passed: emits [AuthLoading, AuthAuthenticated] on successful verified login
00:19 +2: C:/Users/Lydia/Desktop/flutterproject/jaclean-1/test/blocs/auth/auth_bloc_test.dart: AuthBloc Tests emits [AuthLoading, AuthEmailNotVerified] on unverified em
ail login
[AuthBloc] LoginRequested event received with email: test@example.com
[AuthBloc] User email not verified.
Test Passed: emits [AuthLoading, AuthEmailNotVerified] on unverified email login
00:19 +3: C:/Users/Lydia/Desktop/flutterproject/jaclean-1/test/blocs/auth/auth_bloc_test.dart: AuthBloc Tests emits [AuthLoading, AuthError] on login failure
[AuthBloc] LoginRequested event received with email: bad@example.com
[AuthBloc] Login failed with error: [firebase_auth/user-not-found] No user found with this email.
Test Passed: emits [AuthLoading, AuthError] on login failure
00:19 +4: C:/Users/Lydia/Desktop/flutterproject/jaclean-1/test/blocs/auth/auth_bloc_test.dart: AuthBloc Tests emits [AuthLoading, AuthPasswordReset] on successful passw
ord reset
[AuthBloc] PasswordResetRequested event received for email: user@example.com
[AuthBloc] Password reset email sent successfully.
Test Passed: emits [AuthLoading, AuthPasswordReset] on successful password reset
00:19 +5: C:/Users/Lydia/Desktop/flutterproject/jaclean-1/test/blocs/auth/auth_bloc_test.dart: AuthBloc Tests emits [AuthLoading, AuthError] on password reset failure
[AuthBloc] PasswordResetRequested event received for email: invalid-email
[AuthBloc] Password reset failed with error: [firebase_auth/invalid-email] The email address is badly formatted.
Test Passed: emits [AuthLoading, AuthError] on password reset failure
00:32 +17: All tests passed!
```

Figure 29. Test results

CONCLUSION

The development of the 9jaclean mobile application demonstrated how Flutter can be effectively used to build scalable, cross-platform solutions aimed at addressing real-world challenges like waste management and environmental sustainability in Nigeria.

By leveraging the Flutter framework along with Firebase and BLoC state management, the project successfully delivered an application with robust features including waste pickup scheduling, second-hand goods marketplace, and user wallet functionality. The use of customized widgets, reusable UI components, and clearly organized file structures made the development process cleaner and more maintainable.

Agile Scrum methodology enabled efficient project management and iterative development, ensuring that core features were prioritized and implemented within the given timeframe. Sprint-based planning facilitated continuous feedback and improvement, while unit testing further strengthened code reliability.

Despite encountering technical challenges most notably, the inability to fully integrate Google Maps due to billing issues the project adapted by integrating alternative APIs like OpenRouteServices for navigation functionality. Such flexibility showcases the adaptability of the development approach.

While the application is currently in a functional and deployable prototype state, there remains significant potential for enhancement. Future versions could integrate advanced features such as push notifications, referral systems, or partnerships with local recycling companies. The insights gained during this project have equipped the development team with practical experience in mobile app development, UI design, and cloud integration skills that are critical for future endeavors.

Ultimately, 9jaclean not only contributes to sustainable waste management but also empowers users to take part in a greener future through technology. With further iteration and user feedback, the app has the potential to scale and deliver real environmental and social impact.

REFERENCES

Dart Dev Team. (2024). *pub.dev – Dart & Flutter package manager*. Retrieved from https://pub.dev

Felangel. (2023). *flutter_bloc documentation*. Retrieved from https://pub.dev/packages/flutter_bloc

Firebase CLI Docs. (2024). *Command Line Interface Overview*. Retrieved from https://firebase.google.com/docs/cli

Firebase Documentation. (2024). *Firebase Authentication & Firestore*. Retrieved from https://firebase.google.com/docs

Flutter Documentation. (2023). *Introduction to Flutter*. Retrieved from https://docs.flutter.dev

Flutter Documentation. (2023). *Managing state with BLoC and Cubit*. Retrieved from https://docs.flutter.dev

Goswami, M. (2023). *Understanding the BLoC Pattern in Flutter: A Beginner's Guide*. Medium. Retrieved from https://medium.com/@mayurgoswami/flutter-bloc-pattern-beginners-guide

Google Cloud. (2023). *Getting started with Maps SDK for Android & iOS*. https://cloud.google.com/maps-platform

Google Developers. (2023). *Flutter Documentation*. Retrieved from https://flutter.dev/docs

Highsmith, J. (2022). *Agile Project Management: Creating Innovative Products*. Addison-Wesley.

IBM. (2023). *Agile software development*. Retrieved from https://www.ibm.com/topics/agilesoftware-development

Islomov, S. (2022). *Flutter Bloc Architecture: A Complete Guide*. Retrieved from https://medium.flutterdevs.com/flutter-bloc-architecture-a-complete-guide-2022-65d7d8d92747

Janson, T. (2022). *Getting Started with Flutter: Installing SDK and Setting Up Your Environment*. CodeWithTom. Retrieved from https://codewithtom.dev/flutter-install-guide

Krause, M. (2022). *Mastering Cloud Firestore: Real-time data and offline-first architecture*. Medium. Retrieved from https://medium.com/firebase-tips-tricks

Lucidchart. (2023). *What is Agile methodology?*. Retrieved from https://www.lucidchart.com/blog/what-is-agile-methodology

Martin, E. (2021). *Why Flutter is the future of cross-platform development*. Medium. Retrieved from https://medium.com/@ericmartin/why-flutter-is-the-future-of-cross-platform-developmentb1d295a8f5c6

O'Connell, C. (2022). *Using Google Workspace to Manage Agile Projects*. *Project Management Journal*, 53(1), 34–45.

OpenRouteService. (2023). *Developer Documentation*. https://openrouteservice.org/dev

Patel, R. (2023). *Why Cloud Firestore is ideal for real-time mobile apps*. Dev.to. Retrieved from https://dev.to/firebase/firestore-realtime-apps

Rohan, A. (2020). *How to integrate OpenRouteService API with Flutter*. Medium. https://medium.com

Rosencrance, L. (2019). *What is Firebase? Features and Benefits*. TechTarget. Retrieved from https://www.techtarget.com/searchmobilecomputing/definition/Firebase

Sannino, M. (2022). *Architecting Flutter apps: The BLoC pattern explained*. LogRocket Blog. Retrieved from https://blog.logrocket.com/architecting-flutter-apps-bloc-pattern-explained

Schwaber, K., & Sutherland, J. (2020). *The Scrum Guide™: The Definitive Guide to Scrum: The Rules of the Game*. Scrum.org. Retrieved from https://scrumguides.org

Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education.

Wang, C., Zhao, X., & Li, H. (2021). *Accelerating Mobile Development Using Reusable Software Components*. *International Journal of Software Engineering and Technology*, 13(2), 45–57.

Wogu, J. (2022). *Mastering Flutter BLoC Pattern: Build scalable apps*. Leanpub.