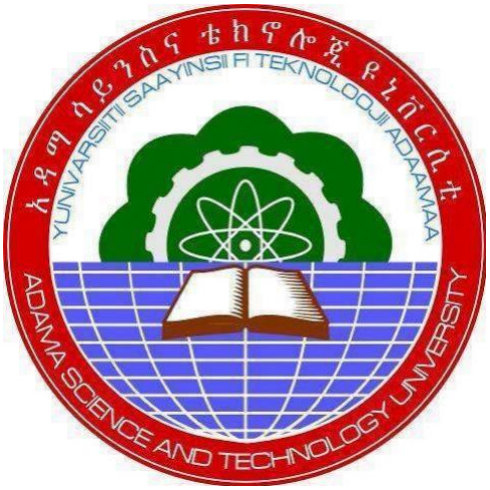


ADAMA SCIENCE AND TECHNOLOGY UNIVERSITY

DEPT OF SOFTWARE ENGINEERING

PROJECT
TITLE:- CHEWATA CHAT APP

Name	Id no
1. Abel Getahun	UGR/30057/15
2. Yasin Shalo	UGR/31366/15
3. Abreham Kifle	UGR/30101/15
4. Aklilu Desalegn	UGR/30121/15
5. Jaleta Kebede	UGR/30722/15
6. Liben Adugna	UGR/30828/15
7. Meklit Abeje	UGR/30881/15

Chewata Chat Application - Project Documentation

1. Overview

Chewata is a modern, cross-platform chat application meticulously crafted using Flutter for the user interface and Firebase for robust backend services. This document serves as a comprehensive guide, offering insights into the application's architectural blueprint, fundamental components, and underlying technical implementation. It is designed to be a valuable resource for developers seeking to understand the codebase structure and core systems of Chewata.

For in-depth details regarding the integration with Firebase, please refer to the dedicated document: **Firebase Integration**. For specific information on the implementation of state management, consult the document titled: **State Management**.

2. Relevant Source Files

- lib/main.dart
- lib/app.dart
- lib/firebase_options.dart
- lib/utils/theme/app_theme.dart
- pubspec.yaml

3. Purpose and Scope

The primary purpose of this document is to provide a clear and detailed overview of the Chewata chat application's technical aspects. It aims to establish a foundational understanding for developers involved in the project, covering the application's architecture, key features, and the technologies employed. The scope of this document includes:

- A high-level description of the application's functionality.
- An overview of the system and component architecture.
- Details on core components such as Firebase integration and state management using GetX.
- Information about the technology stack, theme configuration, application flow, and platform support.
- A summary of the assets and resources included in the project.

4. Application Description

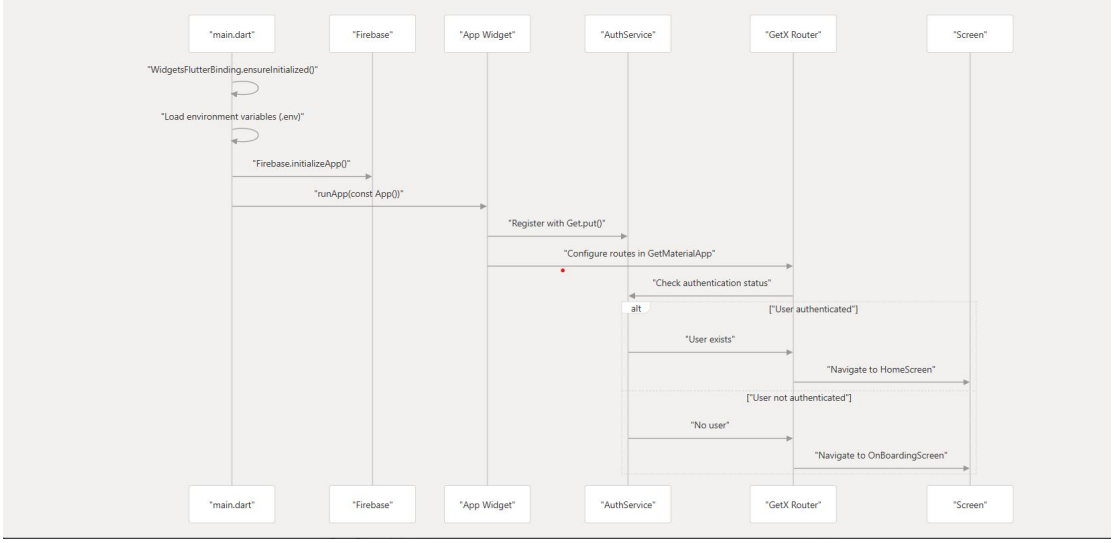
Chewata is a feature-rich chat application designed to provide users with seamless real-time messaging capabilities across various platforms. The application incorporates essential features such as secure user authentication and a smooth onboarding process for new users. Adhering to a clean architecture, Chewata ensures a distinct separation of concerns between the user interface, business logic, and underlying services, promoting maintainability and scalability.

5. Key Technical Features

- **Cross-platform Support:** Engineered to run seamlessly on Android, iOS, Web, macOS, and Windows platforms from a single codebase.
- **Firebase Integration:** Leverages Firebase for robust user authentication and efficient real-time data storage.
- **GetX Framework:** Employs GetX for comprehensive state management, streamlined dependency injection, and simplified route management.
- **Theme Support:** Offers users the flexibility to switch between light and dark theme modes for an enhanced user experience.
- **User Onboarding:** Implements a guided onboarding flow for new users to ensure a smooth introduction to the application.

6. System Architecture Overview

The initialization of the Chewata application involves a sequence of crucial steps to set up the environment and navigate the user to the appropriate screen based on their authentication status. The following diagram illustrates this flow:



7. Component Architecture

The Chewata application's component architecture is structured to handle both authenticated and non-authenticated user states, with GetX playing a central role in managing dependencies and routing. The following diagram outlines the key components and their relationships:

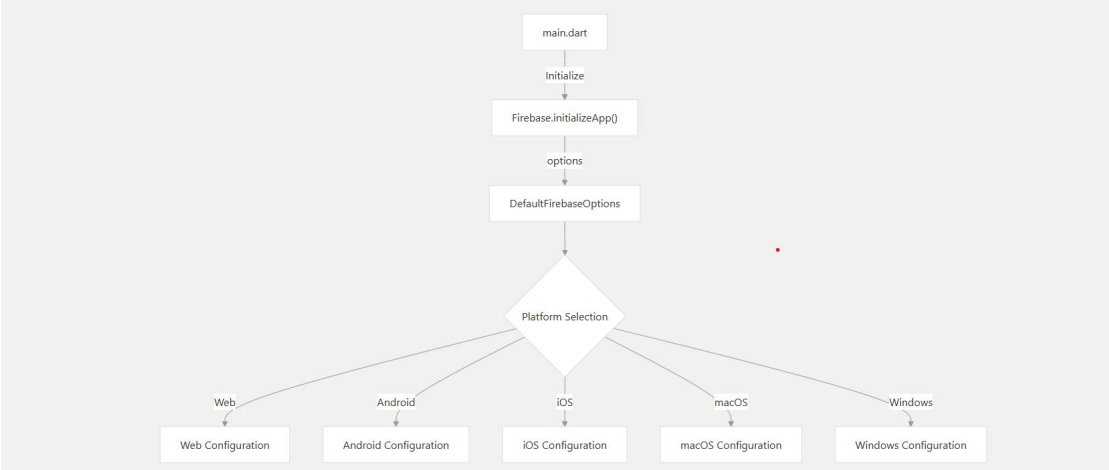
8. Core Components

8.1. Firebase Integration

Chewata utilizes Firebase as its backend-as-a-service platform, providing essential functionalities such as user authentication and data storage. Platform-specific Firebase configurations are managed through environment variables, ensuring a consistent setup across different operating systems.

Platform	Configuration Source	Initialization
Web	.env	DefaultFirebaseOptions.web
Android	.env	DefaultFirebaseOptions.android
iOS	.env	DefaultFirebaseOptions.ios
macOS	.env	DefaultFirebaseOptions.macos
Windows	.env	DefaultFirebaseOptions.windows

The initialization process in main.dart selects the appropriate Firebase options based on the running platform:



8.2. State Management with GetX

Chewata leverages the GetX framework for efficient and reactive state management, streamlined dependency injection, and simplified navigation between different screens.

Component	Purpose	Implementation
Dependency Injection	Manage service and controller instances	Get.put() in lib/app.dart:17-18
Reactive State	Track authentication status	Obx() in lib/app.dart:36-51
Routing	Navigate between different application screens	GetMaterialApp in lib/app.dart:20-32

9. Technology Stack

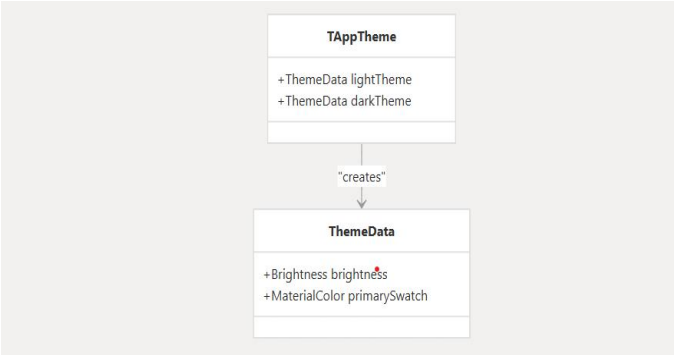
9.1. Core Dependencies

The Chewata application relies on the following core Flutter packages:

Dependency	Version	Purpose
flutter	^3.7.2	Core UI framework
firebase_core	^3.13.0	Firebase initialization
firebase_auth	^5.5.2	Firebase authentication services
cloud_firestore	^5.6.6	Firebase database services
get	^4.7.2	State management, dependency injection
flutter_dotenv	^5.0.2	Management of environment variables
smooth_page_indicator	^1.2.1	UI for onboarding screen pagination
lottie	^3.3.1	Library for displaying animations

10. Theme Configuration

Chewata supports both light and dark themes to cater to user preferences and improve accessibility. The theme configurations are defined within the TAppTheme class:



11. Application Flow

The typical user interaction flow within the Chewata application follows these key steps:

1. **Application Launch:** The `main.dart` file initiates the application by initializing Firebase and running the root `App` widget.
2. **Authentication Check:** The `App` widget, specifically within its `_determineInitialScreen()` method, checks the user's authentication status using the `AuthService`.
    - **Authenticated User:** If the user is already authenticated, the application navigates directly to the `HomeScreen`, providing access to the chat functionalities.
    - **Unauthenticated User:** If the user is not authenticated, the `OnBoardingScreen` is displayed as the initial screen.
3. **Onboarding:** New users are guided through the onboarding experience, which might include introductory screens and feature highlights.
4. **Authentication:** After or during the onboarding process, users are presented with the `AuthScreen`, allowing them to sign up for a new account or log in to an existing one. This process is handled by the `AuthController` and `AuthService`, utilizing Firebase Authentication.
5. **Main Application:** Upon successful authentication, users are directed to the `HomeScreen`, where they can engage with the core chat functionality of the application.

12. Platform Support

Chewata is designed to operate across a range of platforms, with specific configurations in place for Firebase on each supported operating system.

Platform	Support	Configuration Source
Android	✓	Environment variables in .env
iOS	✓	Environment variables in .env
Web	✓	Environment variables in .env
macOS	✓	Environment variables in .env
Windows	✓	Environment variables in .env
Linux	✗	Not configured

The platform-specific configurations are loaded using the `flutter_dotenv` package and utilized by the `DefaultFirebaseOptions` class to initialize Firebase correctly on each platform.

13. Assets and Resources

The Chewata application includes the following essential assets and resources to enhance the user interface and experience:

- **Images:**
    - Background images for the onboarding screens.
    - Background images for the authentication (sign-up and login) screens.
- **Fonts:**
    - **WinkySans:** Regular, Bold, and Italic variations.
    - **Poppins:** Regular and Bold variations.

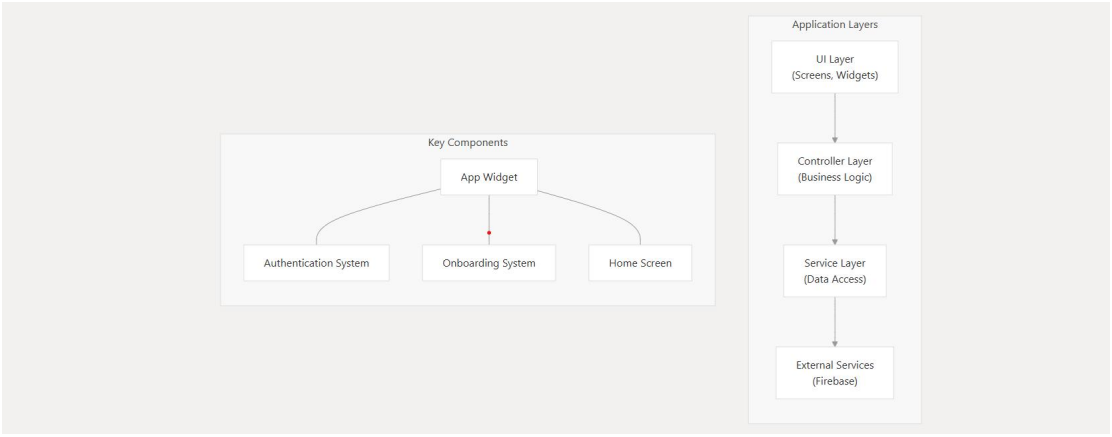
These assets are typically located within the `assets` directory of the Flutter project and are referenced within the application's UI components.

2. Application Architecture

This section details the high-level architecture of the Chewata chat application. It explains its initialization flow, component structure, and key design patterns, focusing on the core architectural elements rather than specific feature implementations.

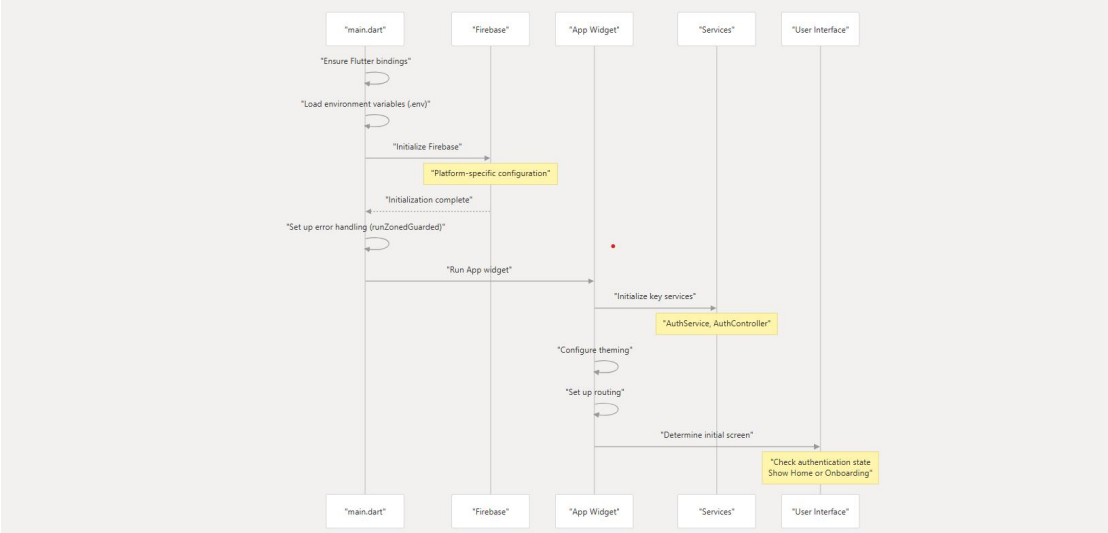
2.1. High-Level Overview

The Chewata application is built with Flutter and follows a clean architecture approach with clear separation of concerns. It uses Firebase for backend services and GetX for state management and dependency injection.



2.2. Application Initialization

The application follows a structured initialization process to ensure all required services and configurations are properly set up before rendering the UI.



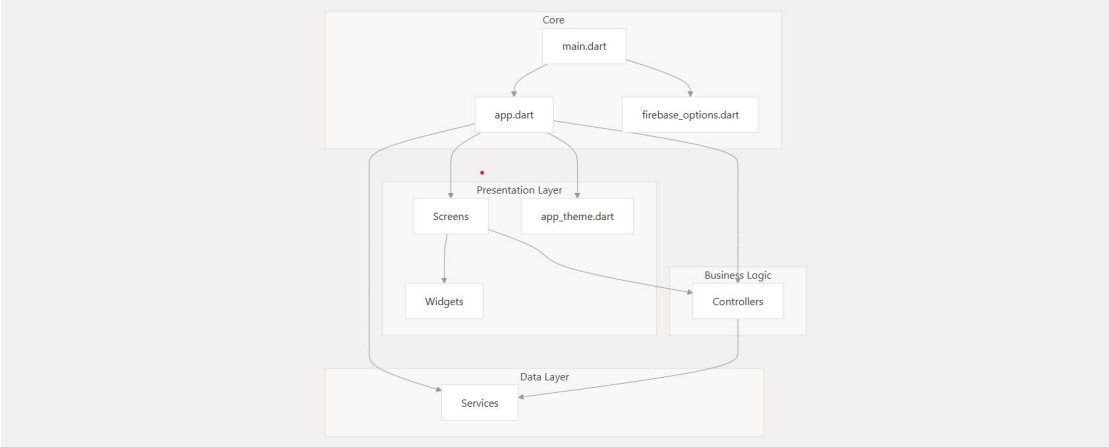
2.2.3. Environment Configuration and Error Handling

The application starts by initializing Flutter bindings, loading environment variables from a .env file, and setting up Firebase with platform-specific configurations. It employs runZonedGuarded for global error handling to catch and log any uncaught exceptions.

```
// Global error handling setup
runZonedGuarded(() {
  runApp(const App());
}, (error, stackTrace) {
  debugPrint('Uncaught Error: $error');
  debugPrint('Stack Trace: $stackTrace');
});
```

2.3. Component Structure

`The application is organized into several key components that work together to create the complete system.`

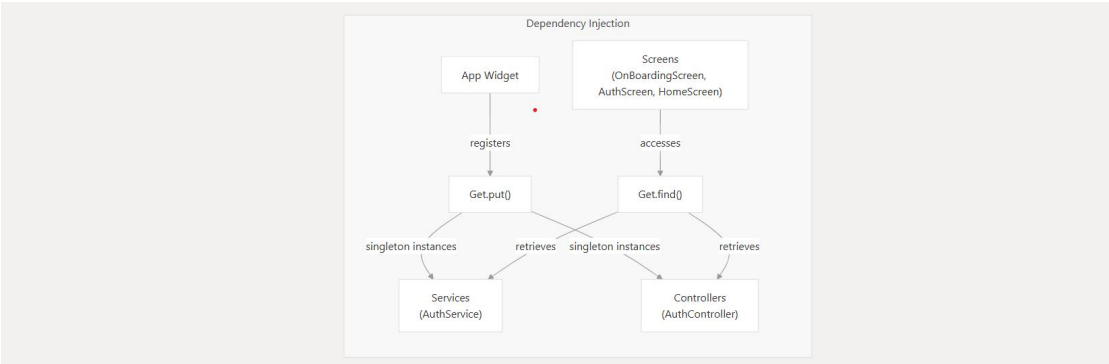


2.3.2. Component Responsibilities

Component	Responsibility	Implementation
main.dart	Application entry point	Initializes environment, Firebase, error handling
app.dart	Core application setup	Configures dependency injection, routing, theming
AuthService	Handles Firebase Authentication operations	Interacts with Firebase Auth SDK
AuthController	Manages authentication business logic	Controls authentication state and operations
Screens	User interface views	Displays UI elements and interacts with controllers
Widgets	Reusable UI building blocks	Creates specific UI components
app_theme.dart	Visual styling configuration	Defines light and dark themes

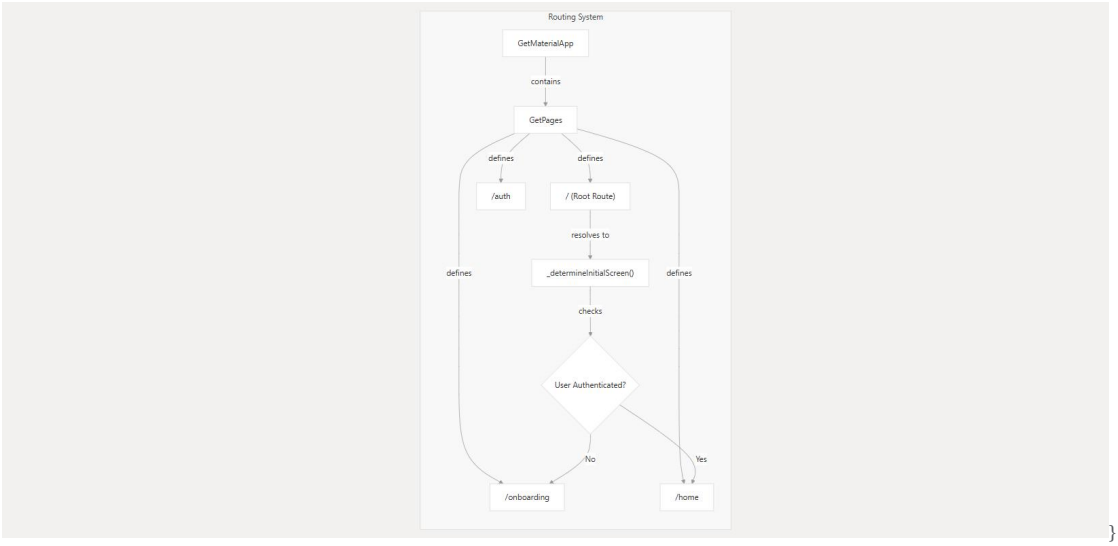
2.4.1. Dependency Injection and State Management

The application uses GetX for dependency injection and state management. Services and controllers are registered with Get.put() and accessed with Get.find().



2.4.3. Routing and Navigation

The application uses GetX's routing system to define and navigate between screens. The initial route is determined based on the authentication state.



}

Route Definition

The application defines its routes in the GetMaterialApp widget:

```
getPages: [
  GetPage(name: '/', page: () => _determineInitialScreen()),
  GetPage(name: '/onboarding', page: () => const OnBoardingScreen()),
  GetPage(name: '/auth', page: () => const AuthScreen()),
  GetPage(name: '/home', page: () => const HomeScreen()),
],
```

Authentication-Based Routing

The initial route is determined dynamically based on the user's authentication state

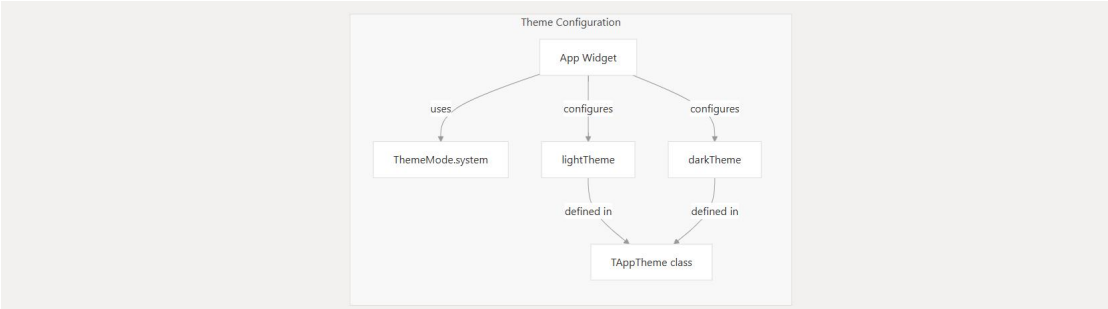
```
Widget _determineInitialScreen() {
  return Obx(() {
    try {
      final firebaseUser = AuthService.instance.firebaseUser.value;

      // If the user is logged in, show home screen
      if (firebaseUser != null) {
        return const HomeScreen();
      }

      // Otherwise show onboarding
      return const OnBoardingScreen();
    } catch (e) {
      // If there's an error, default to the onboarding screen
      return const OnBoardingScreen();
    }
  });
}
```

2.4.4. Theme Management

The application supports both light and dark themes defined in the TAppTheme class:



The light and dark themes are defined in the TAppTheme class and applied through the GetMaterialApp widget's theme properties:

2.5. Dependencies

Dependency	Purpose
firebase_core	Firebase initialization and core functionalities
firebase_auth	User authentication services
cloud_firestore	Real-time NoSQL cloud database services
get	State management, dependency injection, and routing
flutter_dotenv	Management of environment variables
smooth_page_indicator	UI component for interactive page indicators (e.g., onboarding)
lottie	Library for rendering vector-based animations

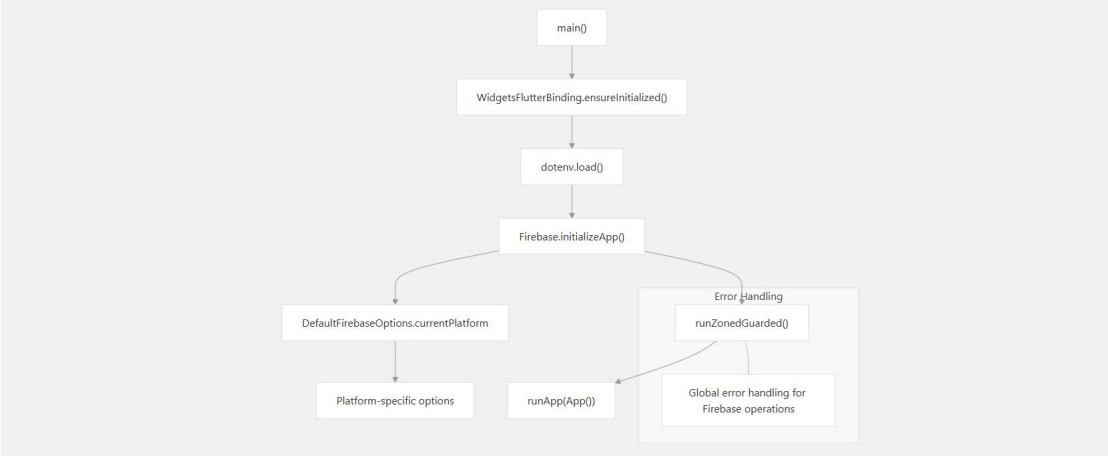
2.4.5. Firebase Integration

Overview

This document details how Firebase services are integrated into the Chewata chat application, covering initialization, configuration management, and platform-specific considerations. The Firebase integration serves as the foundation for the application's authentication and data storage capabilities. For information about authentication implementation details, see User Authentication.

Initialization Process

Firebase is initialized during application startup before the Flutter app begins rendering.



The initialization process takes place in the main() function and follows these steps:

Flutter bindings are initialized to ensure plugin functionality
Environment variables are loaded from a .env file
Firebase is initialized with platform-specific options
The application is launched within an error-handling zone
Sources: lib/main.dart9-28

Configuration Management

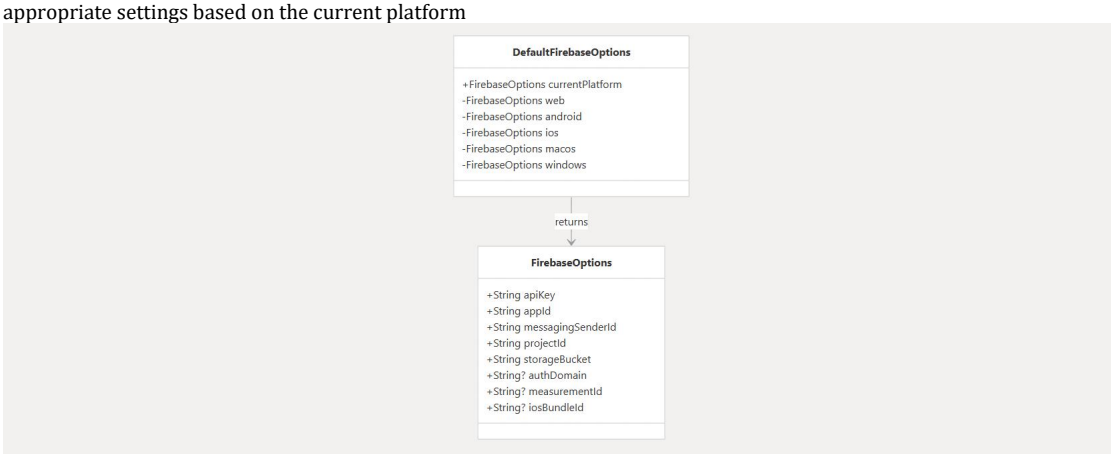
Firebase configuration is securely managed using environment variables to avoid hardcoding sensitive API keys and credentials in the source code.

Environment Variables Approach

The application uses the flutter_dotenv package to load Firebase configuration from a .env file, which is not included in version control for security reasons.

Platform-Specific Configuration

The DefaultFirebaseOptions class provides platform-specific Firebase configuration, dynamically selecting the appropriate settings based on the current platform



Each platform configuration includes the appropriate parameters required by Firebase for that platform, loaded from environment variables:

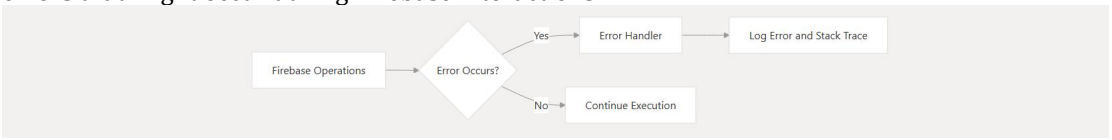
Platform	Configuration Parameters
Web	apiKey, appId, messagingSenderId, projectId, authDomain, storageBucket, measurementId
Android	apiKey, appId, messagingSenderId, projectId, storageBucket
iOS	apiKey, appId, messagingSenderId, projectId, storageBucket, iosBundleId
macOS	apiKey, appId, messagingSenderId, projectId, storageBucket, iosBundleId
Windows	apiKey, appId, messagingSenderId, projectId, authDomain, storageBucket, measurementId

Firebase Plugin Integration

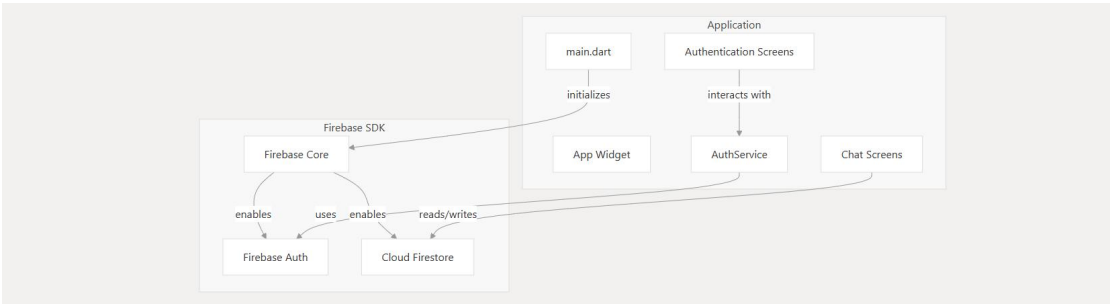
The Firebase Core plugin is registered with the Flutter plugin registry to enable Firebase functionality across platforms. This integration is managed automatically by Flutter for each supported platform.

Error Handling

Firebase operations are wrapped within a runZonedGuarded block to catch and log any uncaught errors that might occur during Firebase interactions.



Firebase Service Architecture
The overall architecture of Firebase integration within the Chewata application follows this pattern:



This separation of concerns allows the application to interact with Firebase services through dedicated service classes, which encapsulate the Firebase API calls and provide application-specific functionality.

Security Considerations

The Firebase integration in Chewata implements the following security measures:

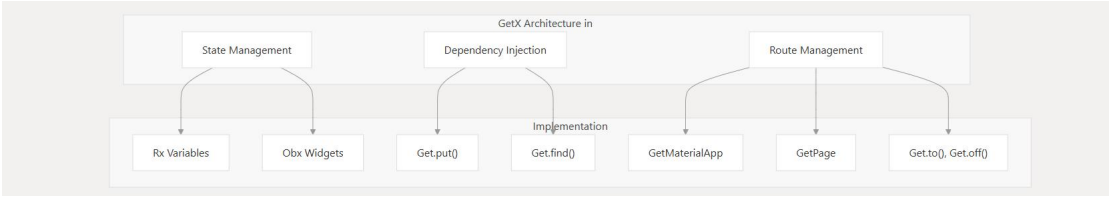
1. Sensitive configuration values stored in environment variables, not in source code
2. Platform-specific configurations to ensure appropriate settings per platform
3. Error handling to prevent Firebase operation failures from crashing the application
4. Proper initialization sequencing to ensure Firebase is ready before the application starts using Firebase services

State Management

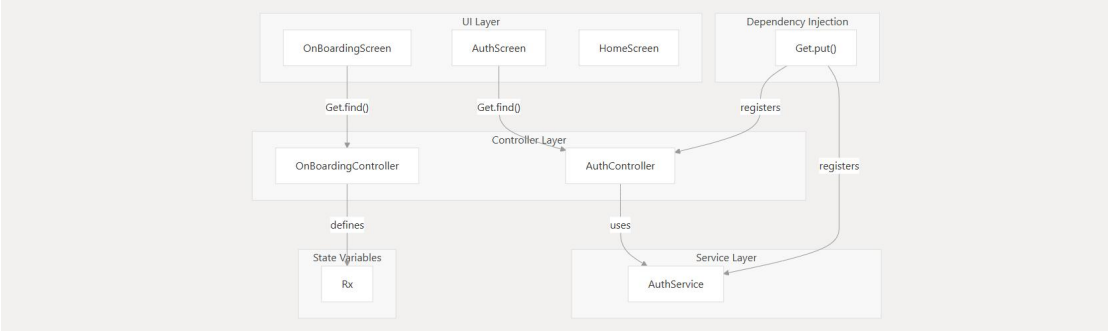
This document explains how the Chewata chat application uses the GetX framework for state management, dependency injection, and navigation routing. GetX provides a comprehensive solution that helps maintain clean code architecture by separating business logic from UI components.

Overview of State Management Approach
Chewata employs the GetX framework as its central architecture for managing application state and dependencies. This approach allows for:

- Reactive state management with automatic UI updates
- Simple dependency injection without complex provider hierarchies
- Streamlined navigation and routing
- Separation of concerns between UI, business logic, and services



GetX Component Structure
The application's GetX implementation follows a three-tier pattern with UI components, controllers, and services.



Reactive State Management
Chewata uses GetX's reactive state management to automatically update the UI when underlying data changes.

Reactive Variables (Rx)
Reactive variables are created using the .obs extension or Rx<Type> wrapper. When these variables change, any widget observing them will automatically rebuild.
This creates a reactive integer that tracks the current page in the onboarding flow.

Reactive UI Components
The application uses Obx widgets to create UI components that automatically update when observed reactive variables change

```
Obx(() {
  // UI code that depends on reactive variables
  // Will rebuild whenever those variables change
})
```

A key example is the initial screen determination in the App widget which reactively responds to authentication state changes:

```
Obx(() {
  final firebaseUser = AuthService.instance.firebaseUser.value;
  if (firebaseUser != null) {
    return const HomeScreen();
  }
  return const OnBoardingScreen();
})
```

Dependency Injection
Chewata uses GetX's dependency injection to manage service and controller instances throughout the application.

Registering Dependencies
Dependencies are registered in the App widget's build method:

```
Get.put(AuthService(), permanent: true);
Get.put(AuthController(), permanent: true);
```

The permanent: true flag ensures these instances persist throughout the application lifecycle.

Accessing Dependencies

Controllers and services are accessed throughout the application using Get.find() or a static getter pattern:

```
// Static getter pattern used in OnBoardingController
static OnBoardingController get instance => Get.find();
```

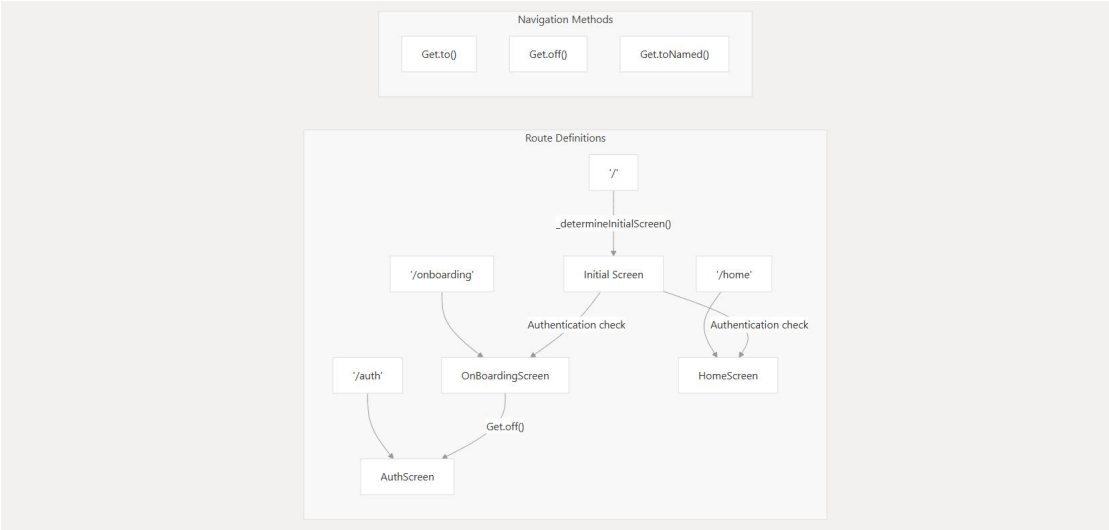
This pattern allows clean access to controllers from UI components without explicitly passing instances.

Navigation and Routing

GetX provides a clean routing system that's used throughout Chewata for navigation.

Route Definitions

Routes are defined in the GetMaterialApp widget:



Navigation Implementation

Navigation is performed using GetX methods like Get.to(), Get.off(), and Get.toNamed().
For example, in the OnBoardingController:

```
void nextPage() {
  if (currentIndex.value == 2) {
    // Navigate to LoginScreen and remove OnBoardingScreen from the stack
    Get.off(() => const AuthScreen());
  } else {
    // ... page animation code
  }
}
```

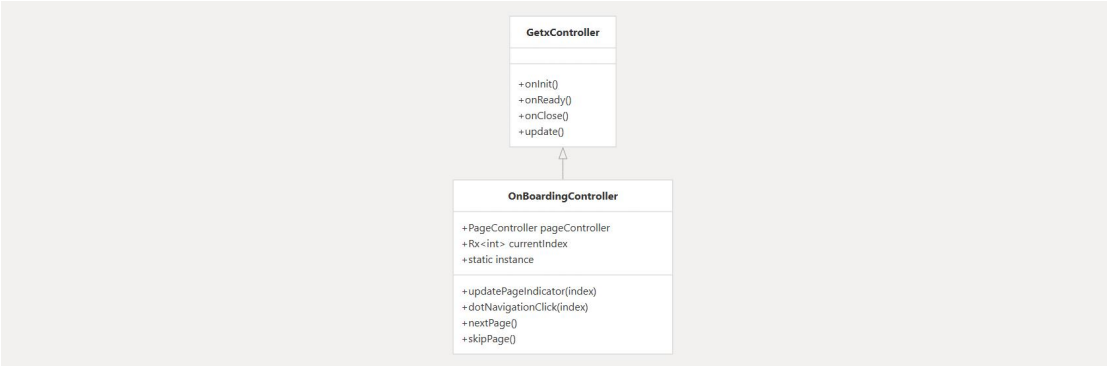
This code navigates to the AuthScreen and removes the current screen from the navigation stack when the user reaches the last onboarding page.

Controller Implementation Pattern

Chewata follows a consistent pattern for implementing controllers with GetX:

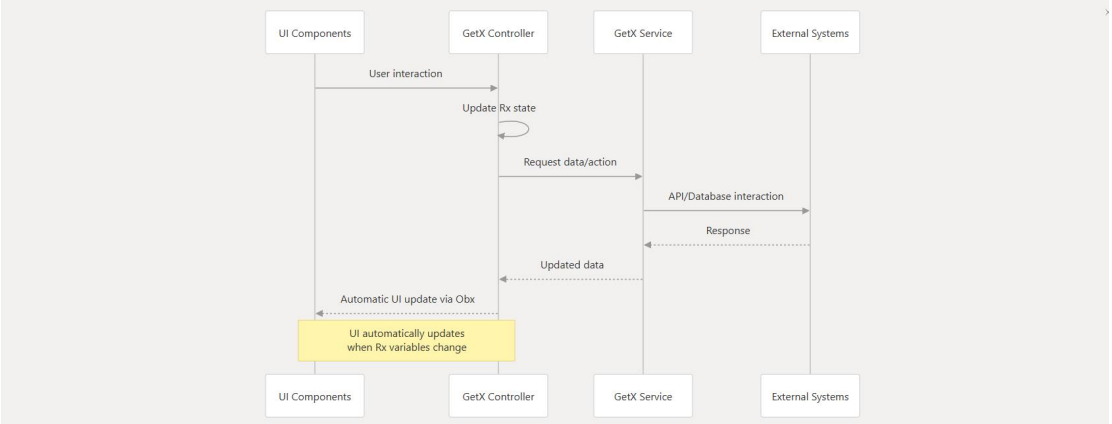
Component	Implementation	Purpose
Class Definition	class XController extends GetxController	Base class providing lifecycle hooks
Singleton Access	static XController get instance => Get.find()	Easy access throughout the app
State Variables	Rx<Type> variable = value.obs	Reactive state that updates UI
UI Binding	Obx(() => ...)	Reactive UI components
Navigation	Get.to(), Get.off(), etc.	Screen navigation

The OnBoardingController serves as a good example, managing the current page index reactively and handling navigation actions.



State Flow and Data Binding

The following diagram illustrates how state flows through the application and how UI components bind to this state:



Theme Management

The application also uses GetX for theme management, with the theme configuration defined in the GetMaterialApp:

```
GetMaterialApp(
  title: 'Chewata chat',
  themeMode: ThemeMode.system,
  theme: TAppTheme.lightTheme,
  darkTheme: TAppTheme.darkTheme,
  // ...
)
```

Benefits of GetX Architecture in Chewata

The GetX architecture provides several benefits to the Chewata application:
Simplified State Management: Reactive programming with minimal boilerplate
Clean Architecture: Clear separation between UI, business logic, and data layers
Code Reusability: Controllers and services can be easily reused across different UI components
Performance: Efficient rebuilds that only affect UI components observing changed state
Maintainability: Modular code structure with clear responsibilities

3. Project Team Responsibilities

This section outlines the responsibilities of each team member involved in the Chewata project. Clear role definitions are crucial for efficient collaboration, streamlined development, and successful project completion.

Team Members and Roles:

Yasin Shalo: Figma Design

Abel Getahun: Coordinator - Integration

Abreham Kifle: Frontend Developer

Jaleta Kebede : Business Logic Developer

Liben Adugna: QA | Tester

Akililu Dessalegn: Frontend Developer

Meklit Abeje : Database and Documentation

3.1 Detailed Responsibilities

3.1.1 Yasin - Figma Design

UI/UX Design: Responsible for creating the user interface (UI) and user experience (UX) design of the Chewata application using Figma.

Wireframing and Prototyping: Developing wireframes and interactive prototypes to visualize the application's layout, navigation, and functionality.

Design System Maintenance: Creating and maintaining a consistent design system, including style guides, component libraries, and visual assets.

Collaboration with Developers: Working closely with frontend developers (Abreham and Ake) to ensure the design is accurately implemented and technically feasible.

User Feedback Incorporation: Gathering and incorporating user feedback to iterate on the design and improve the overall user experience.

Asset Delivery: Providing developers with all necessary design assets (e.g., icons, images, fonts) in the appropriate formats.

3.1.2 Abel - Coordinator - Integration

Project Coordination: Overseeing the integration of different parts of the Chewata application, ensuring seamless interaction between frontend, backend, and database components.

Communication Facilitation: Acting as the primary point of contact for communication between team members, stakeholders, and external parties.

Meeting Organization: Scheduling and leading regular project meetings, including sprint planning, progress updates, and issue resolution sessions.

Task Management: Assisting in defining project tasks, assigning responsibilities, and tracking progress using project management tools.

Risk Management: Identifying potential integration risks, developing mitigation strategies, and proactively addressing any issues that may arise.

Dependency Management: Managing dependencies between different components and ensuring that all necessary resources are available for integration.

Release Management: Coordinating the release of new versions of the application, including testing, deployment, and documentation.

Technical Guidance: Providing technical guidance and support to the development team, particularly regarding integration challenges.

3.1.3 Abreham - Frontend Developer

UI Implementation: Developing the user interface (UI) of the Chewata application based on the designs provided by Yasin, using Flutter.

Frontend Architecture: Contributing to the design and implementation of the frontend architecture, ensuring scalability, maintainability, and performance.

State Management: Implementing state management solutions (likely using GetX) to handle data flow and UI updates.

API Integration: Integrating the frontend with backend APIs (developed by Jaleta) to fetch and display data.

Cross-Platform Development: Ensuring the application functions correctly and consistently across different platforms (iOS and Android).

Code Optimization: Writing clean, efficient, and well-documented code, adhering to coding standards and best practices.

Collaboration: Working closely with Ake, Jaleta, and Yasin to ensure seamless integration of UI, business logic, and design.

Testing: Participating in testing the frontend components to ensure they meet the requirements and function correctly.

3.1.4 Jaleta - Business Logic Developer

Backend Development: Designing, developing, and maintaining the backend logic of the Chewata application.

API Development: Creating and documenting RESTful APIs to expose application functionality to the frontend (developed by Abreham and Ake).

Data Modeling: Defining the data structures and schemas for the application, in collaboration with Meklit (Database Developer).

Business Rule Implementation: Implementing the core business rules and logic of the Chewata application.

Performance Optimization: Ensuring the backend is scalable, performant, and secure.

Database Interaction: Interacting with the database (designed and managed by Meklit) to store and retrieve data.

Collaboration: Working closely with Abreham, Ake, and Meklit to ensure seamless integration of the backend, frontend, and database.

Testing: Writing unit and integration tests to ensure the backend logic is robust and error-free.

3.1.5 Liben - QA | Tester

Test Planning: Developing comprehensive test plans and test cases to ensure the quality of the Chewata application.

Test Execution: Executing test cases across different platforms and devices to identify bugs, defects, and usability issues.

Bug Reporting: Accurately documenting and reporting bugs using a bug tracking system.

Test Automation: Developing and maintaining automated tests (if applicable) to improve testing efficiency.

Regression Testing: Performing regression testing to ensure that new changes do not introduce new issues.

Performance Testing: Conducting performance testing to evaluate the application's speed, stability, and responsiveness.

Usability Testing: Evaluating the application's usability to ensure it is intuitive and easy to use.

Collaboration: Working closely with developers (Abreham, Ake, and Jaleta) to communicate testing results and ensure timely bug fixes.

3.1.6 Akililu - Frontend Developer

UI Implementation: Developing the user interface (UI) of the Chewata application based on the designs provided by Yasin, using Flutter.

Frontend Development: Working alongside Abreham in developing frontend features.

State Management: Implementing state management solutions (likely using GetX) to handle data flow and UI updates.

API Integration: Integrating the frontend with backend APIs (developed by Jaleta) to fetch and display data.

Cross-Platform Development: Ensuring the application functions correctly and consistently across different platforms (iOS and Android).

Code Optimization: Writing clean, efficient, and well-documented code, adhering to coding standards and best practices.

Collaboration: Working closely with Abreham, Jaleta, and Yasin to ensure seamless integration of UI, business logic, and design.

Testing: Participating in testing the frontend components to ensure they meet the requirements and function correctly.

3.1.7 Meklit - Database and Documentation

Database Design: Designing the database schema for the Chewata application, considering factors such as scalability, performance, and data integrity.

Database Management: Setting up, configuring, and maintaining the database (e.g., using Cloud Firestore).

Data Modeling: Defining the data structures and relationships within the database, in collaboration with Jaleta (Backend Developer).

Documentation: Creating and maintaining comprehensive documentation for the Chewata project, including:

Technical documentation (e.g., architecture diagrams, API documentation).

User documentation (e.g., user manuals, help guides).

Database documentation (e.g., schema diagrams, data dictionaries)

Knowledge Sharing: Ensuring that all relevant information is documented and accessible to the team.

Collaboration: Working closely with all team members, especially Jaleta, to ensure that the database design meets the application's requirements.