IMAGE GALLERY

Gallery

JULY 20, 2005

BLACKBUCKS

ABSTRACT

In the era of mobile-first applications, the demand for visually engaging, efficient, and interactive image gallery applications has grown significantly. Our project, “Image Gallery”, is a dynamic mobile application developed using Flutter, designed to allow users to browse, interact with, and manage images stored in local directories. The application offers advanced functionalities such as folder-wise image categorization, fullscreen viewing with gestures, image sharing, and quick access to detailed metadata—all while maintaining simplicity and performance. The primary goal of the Image Gallery app is to provide users with a fluid and intuitive experience while navigating through their photo collections. The app is structured around Flutter’s rich UI components and uses Dart as the core programming language to implement responsive and efficient interaction. The application supports key features such as pinchtozoom, swipe navigation, favorite tagging, and image sharing, with planned extensibility for editing and more complex operations. Introduction about image gallery project Image Gallery App Using Flutter In the current digital era, the ability to manage, view, and share visual content effectively is an essential feature in any modern mobile device. With the increasing proliferation of smartphones and the rise in photo-centric user behavior, having a simple yet powerful image gallery application is no longer optional—it’s necessary. To cater to this demand, we have developed a mobile application titled “Image Gallery” using Flutter, which offers users an intuitive platform to organize, view, and share images stored locally on their device. Introduction and Objective The Image Gallery app is designed to provide users with an aesthetically pleasing and highly responsive interface for browsing image folders, viewing pictures in fullscreen, zooming into high-resolution images, and sharing content instantly. The primary objective is to maintain a lightweight app architecture while integrating essential features such as gesture-based navigation, folder-level image categorization, and social media sharing. Flutter was chosen as the core development framework due to its performance advantages, expressive UI capabilities, and single-codebase deployment for Android and iOS. The app usesthe Dart programming language, which is well-suited for UI development and asynchronous data processing.

CONTENT

CHAPATER 1

INTRODUCTION

Introduction to Image Gallery Applications

An Image Gallery App is a mobile application that enables users to view, manage, and interact with digital images. It presents images in a user-friendly format, typically using a grid or list layout, and offers features such as zooming, image sorting, sharing, and categorization.

Purpose of the App

Provide a clean and intuitive interface for viewing and managing images.

Allow seamless navigation through a collection of photos.

Support interaction with images, such as zooming or marking favorites.

In the age of smartphones and digital media, photo gallery apps have become essential. Whether for personal memories, artistic portfolios, or shared social content, such apps are widely used. By developing an image gallery app in Flutter, developers can create high-performance, cross-platform apps efficiently.

Why Use Flutter for Image Gallery Apps

Flutter is an open-source UI toolkit developed by Google for building natively compiled applications for mobile, web, and desktop from a single codebase. It uses the Dart programming language and provides a rich set of pre-built widgets and tools.

Advantages of Flutter for an Image Gallery App:

1. Cross-Platform Development: Write once, run on both Android and iOS.

2. Fast Development: Hot reload enables rapid UI experimentation and bug fixes.

3. Rich UI Components: Widgets like GridView, Image, and Hero help build beautiful UIs.

4. Performance: Flutter apps compile to native ARM code, ensuring smooth image rendering and scrolling.

5. Integration Ready: Easily integrates with device storage, camera, and cloud services like Firebase.

Core Features of the Flutter Image Gallery App

1. Image Grid Display

Utilizes GridView.builder to create a responsive image gallery.

Dynamically loads images either from local storage or URLs.

2. Full-Screen Image Viewer

Implements image zoom and swipe navigation using packages like photo_view.

Enhances user experience with animations (Hero widget).

3. Image Picker Integration

Allows users to upload or take photos using the image_picker package.

Provides options for capturing or selecting multiple images.

4. Image Caching

Uses cached_network_image to improve performance and reduce data usage when displaying network images.

5. Favorites and Sharing

Users can mark images as favorites.

Enables sharing images via built-in device options.

Technical Stack and Architecture

Flutter Packages Used:

image_picker: To pick images from camera or gallery.

photo_view: For zoomable full-screen image viewing.

cached_network_image: Efficient loading and caching of images from the web.

firebase_storage (optional): For storing images in the cloud.

App Architecture:

Model-View-Controller (MVC) or Provider Pattern for state management.

Separation of Concerns: UI, data logic, and storage access are modularized.

State Management: Uses Provider or Riverpod to manage app state (e.g., list of images, favorites).

Storage and Data Flow:

Images can be stored locally on the device or uploaded to Firebase.

Metadata (like image titles or favorite status) can be stored using SQLite or Firestore.

Future Scope and Conclusion

Enhancements and Future Scope:

AI Tagging: Use machine learning to auto-tag and categorize images.

Cloud Sync: Integrate full cloud backup and multi-device sync.

Albums and Categories: Organize images into folders or albums.

Editing Tools: Add filters, crop, or rotate images within the app.

User Authentication: Use Firebase Auth to create personal galleries.

Introduction and Objective

The Image Gallery app is designed to provide users with an aesthetically pleasing and highly responsive interface for browsing image folders, viewing pictures in fullscreen, zooming into high-resolution images, and sharing content instantly. The primary objective is to maintain a lightweight app architecture while integrating essential features such as gesture-based navigation, folder-level image categorization, and social media sharing. Flutter was chosen as the core development framework due to its performance advantages, expressive UI capabilities, and single-codebase deployment for Android and iOS. The app uses the Dart programming .

CHAPTER-2

REQUIREMENT

SPECIFICATION

2.1 System Requirements

✅ **1. Development System Requirements**

🔹 **For Windows:**

- **OS**: Windows 10 or later (64-bit)
- **RAM**: Minimum 8 GB (16 GB recommended)
- **Disk space**: 2.5 GB (plus space for Android Studio/Emulators)
- **Tools**:
    - Flutter SDK
    - Android Studio or Visual Studio Code
    - Git (for Flutter SDK installation)
- **Java**: JDK 11 or newer

🔹 **For macOS:**

- **OS**: macOS 11 (Big Sur) or newer
- **RAM**: 8 GB (16 GB recommended)
- **Disk space**: 2.5 GB (plus space for Xcode and Android Studio)
- **Tools**:
    - Flutter SDK
    - Android Studio/Xcode
    - Git
- **Java**: JDK 11 or newer
- **iOS development**: Xcode + CocoaPods

2.2 Development Environment

✅ 1. Flutter SDK

The **Flutter Software Development Kit (SDK)** is the heart of Flutter development.

🔍 What It Is:

- A complete framework to build cross-platform apps using a single codebase.
- Supports building apps for **Android, iOS, web, desktop (Windows, macOS, Linux)**.

🧰 What It Includes:

- **Flutter engine**: Renders UI components using Skia (a graphics engine).
- **Widgets framework**: Prebuilt UI components for fast UI development.
- **Flutter CLI**: Command-line tools (flutter doctor, flutter build, etc.).
- **Development tools**: For hot reload, state management, and debugging.

📥 Installation:

Download from flutter.dev and extract the SDK, then add it to the system PATH so commands like flutter doctor work.

✅ 2. Dart SDK

Dart is the programming language used to build Flutter apps.

🔍 What It Is:

- An object-oriented, class-based language developed by Google.
- Optimized for UI development.
- Used for writing Flutter app logic and business rules.

💡 Key Features:

- **Null safety**: Prevents many runtime errors.
- **JIT (Just-In-Time) compilation**: Enables **hot reload** during development.

- **AOT (Ahead-Of-Time) compilation**: Produces optimized binaries for production.

👀 **Note:**

- Dart SDK is **bundled** with the Flutter SDK, so you don't need to install it separately.
- You write Dart code in files with .dart extension (e.g., main.dart).

✅ **3. IDE (Integrated Development Environment)**

You can use one of these to write, debug, and run Flutter apps.

🟦 **Visual Studio Code (VS Code)**

✅ **Pros:**

- Lightweight and fast.
- Rich extension marketplace.
- Supports **Flutter and Dart extensions** for autocompletion, debugging, and hot reload.

📦 **Must-Have Extensions:**

- **Flutter**: Adds UI preview, commands, and tool integration.
- **Dart**: Enables language support like syntax highlighting and IntelliSense.

🟩 **Android Studio (or IntelliJ IDEA)**

✅ **Pros:**

- More powerful for Android development.
- Built-in emulator management.
- Great layout inspector and profiler.

⚠️ **Cons:**

- Heavier and uses more system resources than VS Code.

✅ 4. Emulator or Physical Device

You need a device to test your Flutter apps. Flutter supports:

📱 Android Emulator

- Comes with Android Studio.
- Simulates Android device on your PC.
- Can test different versions, screen sizes, and configurations.

📱 iOS Simulator

- Only available on **macOS**.
- Comes with **Xcode**.
- Simulates iPhone or iPad for iOS app testing.

📲 Physical Devices

- Can use USB debugging on Android or wireless debugging.
- iOS devices require a developer account to run custom apps.

✅ Example Workflow:

1. Install Flutter SDK and set PATH.
2. Install VS Code or Android Studio.
3. Connect a device or open emulator.
4. Run:
5. flutter doctor

This checks if everything is installed and configured correctly.

6. Start a new project with:
7. flutter create my_app
8. cd my_app
9. flutter run

CHAPTER-3

SYSTEM ARCHITECTURE

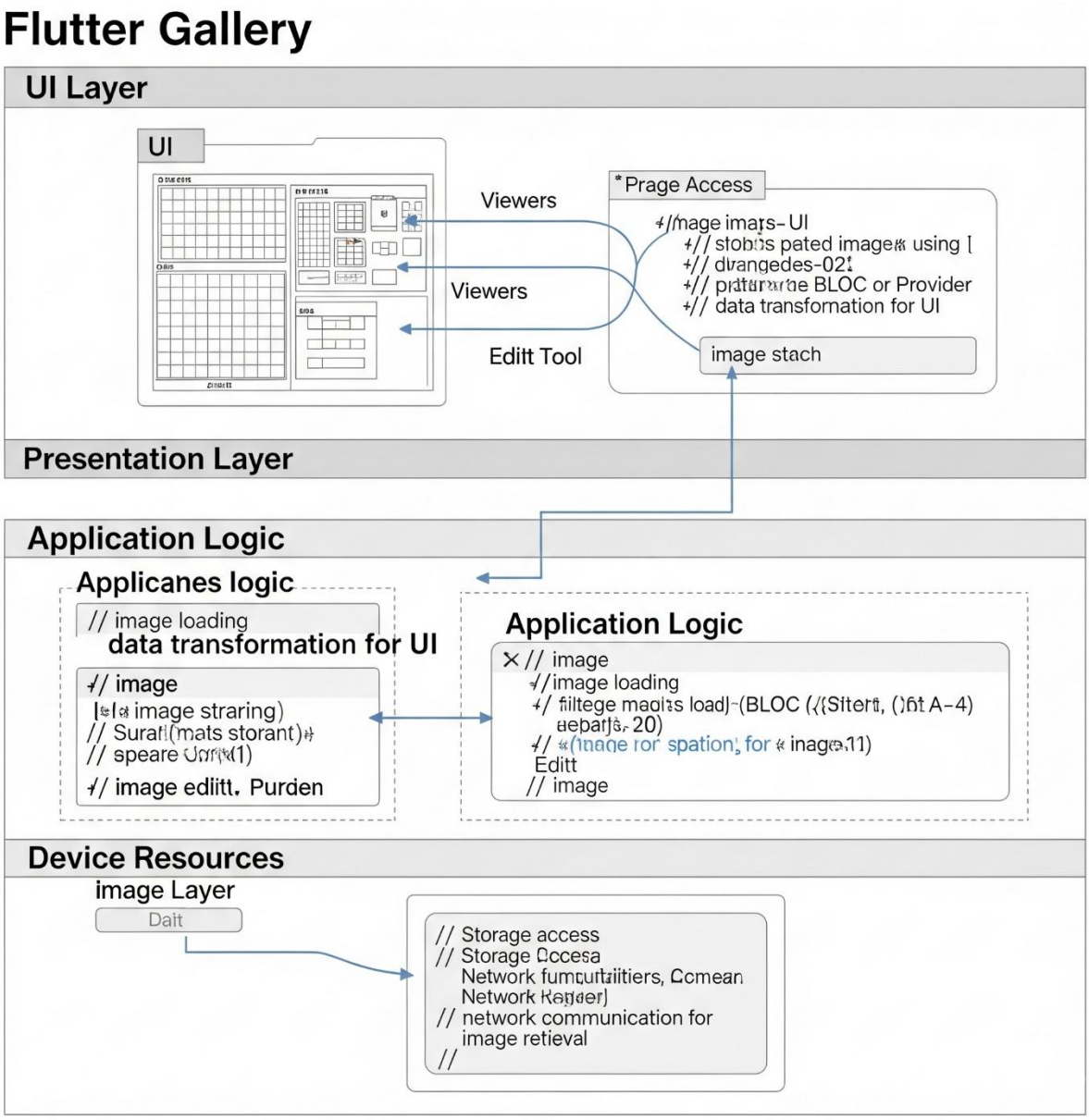

Fig 3.1:-System Architecture

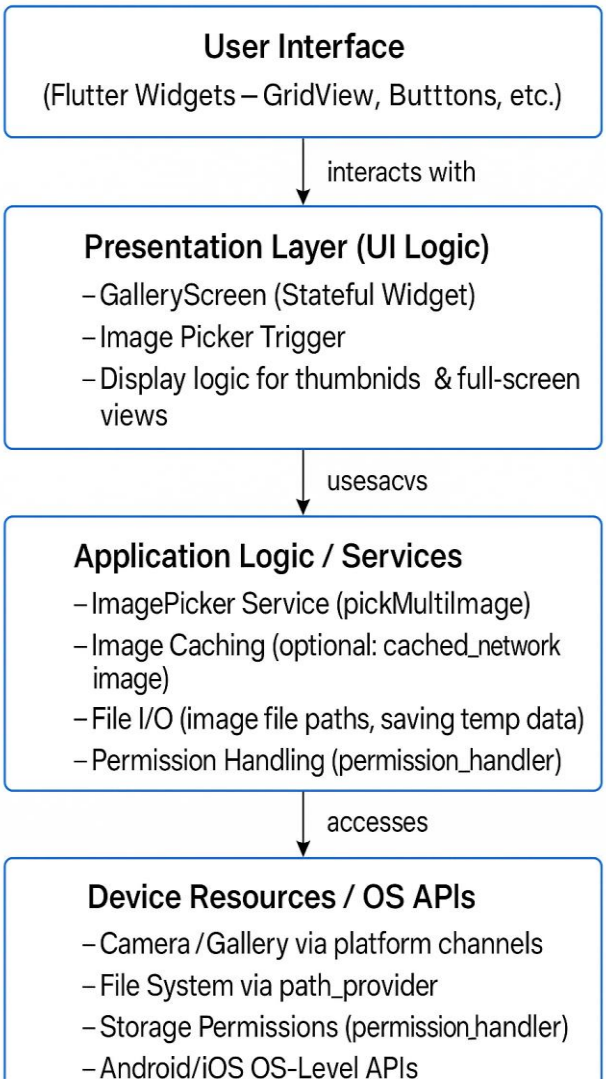

Fig:-Flow Chart of Architecture

Core Components:

- **ImageGalleryApp (StatelessWidget):** The root of the Flutter application, setting up the basic MaterialApp with a title and theme, and defining the initial screen as GalleryScreen.

- **GalleryScreen (StatefulWidget):** This is the main screen where users see their image folders and thumbnails.

    - **_GalleryScreenState:** Manages the state of GalleryScreen.

- **folderImages (Map<String, List<String>>):** This Map is the core data structure. The key is the folder name (e.g., "camera", "downloads"), and the value is a List of String paths to the images within that folder.
- **initState():** Called when the widget is inserted into the widget tree. It immediately calls loadImages() to populate folderImages.
- **loadImages():** This async function is responsible for finding image assets bundled with the application.
    - It reads AssetManifest.json, a file generated by Flutter that lists all included assets.
    - It filters these assets to find only image files (.png, .jpg, .jpeg, .gif).
    - It then parses the paths (e.g., "assets/camera/image1.jpg") to extract the folder name ("camera") and groups images into the folderImages map.
    - setState() is called to rebuild the UI once the images are loaded.
- **_isImageFile(String path):** A helper function to check if a given file path corresponds to an image based on its extension.
- **build(BuildContext context):**
    - Displays a CircularProgressIndicator while folderImages is empty (i.e., while images are loading).
    - Once loaded, it uses ListView.builder to create a scrollable list of folders.
    - For each folder, it displays the folderName as a heading.
    - It then uses a nested GridView.builder to show the thumbnails of images within that specific folder. shrinkWrap: true and NeverScrollableScrollPhysics() are important here to allow the GridView to size itself correctly within the ListView without its own scrolling.
    - Each image thumbnail is a GestureDetector, which, when tapped, navigates to the FullscreenGallery.

FullscreenGallery (StatefulWidget): This screen displays a single image in a zoomable and pannable view, along with various interactive options.

- **_FullscreenGalleryState:** Manages the state of FullscreenGallery.
  - **_pageController:** Used with PageView (internally by PhotoViewGallery) to allow swiping between images.
  - **currentIndex:** Tracks which image is currently being displayed in the gallery.
  - **favorites (Set<String>):** A simple Set to store the paths of images marked as favorites. (Note: This is not persisted across app sessions in this example).
  - **onPageChanged(int index):** Updates currentIndex when the user swipes to a new image.
  - **_shareImage():** Uses the share_plus package to open the device's sharing options for the current image.
  - **_deleteImage():** Removes the current image from the images list. It handles updating currentIndex and pops the screen if no images remain. (Note: This only removes the image from the in-memory list, not from the device's storage).
  - **_toggleFavorite():** Adds or removes the current image from the favorites set.
  - **_showDetails():** Displays an AlertDialog with mock image details (name, size, date, location). In a real app, this data would come from the actual image metadata.
  - **_editImage():** Shows a showModalBottomSheet with a list of mock editing options (Framing, Adjust, Beauty, Erase, Filters, Border).
  - **_moreOptions():** Shows another showModalBottomSheet with more general options (Set as Wallpaper, Google Lens, Super Document, Copy to, Hide, Rename, Slideshow).
  - **_bottomItem(IconData icon, String title):** A helper method to create the ListTile widgets used in the bottom sheets.
  - **build(BuildContext context):**
    - Displays the current image using PhotoViewGallery.builder, which provides zooming and panning capabilities.

- Includes an AppBar with the current image count (e.g., "Image 1 / 10"), a favorite icon (star/star_border), and an info icon.
- A Positioned Container at the bottom holds IconButtons for Share, Edit, Delete, and More Options.

Key Libraries Used:

- **flutter/material.dart:** The core Flutter UI library.
- **flutter/services.dart:** Used for rootBundle.loadString to read AssetManifest.json.
- **photo_view:** A powerful package for easily adding zoomable and pannable image capabilities.
- **share_plus:** A convenient package for sharing content from your Flutter app to other apps on the device.

To run this code, you would need:

1. A Flutter development environment set up.
2. Add photo_view and share_plus to your pubspec.yaml file: YAML dependencies:

flutter:

sdk: flutter photo_view: ^0.14.0 # Use the

latest version share_plus: ^7.0.2 # Use the

latest version

3. Create an assets folder in your project root, and then subfolders within it (e.g., assets/camera, assets/downloads). Place some image files (.png, .jpg, etc.) into these subfolders.
4. Declare your assets in pubspec.yaml: YAML

flutter: uses-material-

design: true assets:

- assets/

(Note: assets/ will include all files in that directory and its subdirectories).

5. Run flutter pub get.
6. Run the app on an emulator or physical device.

CHAPTER-4

SYSTEM IMPLEMENTATION

USE CASE DIAGRAM :-

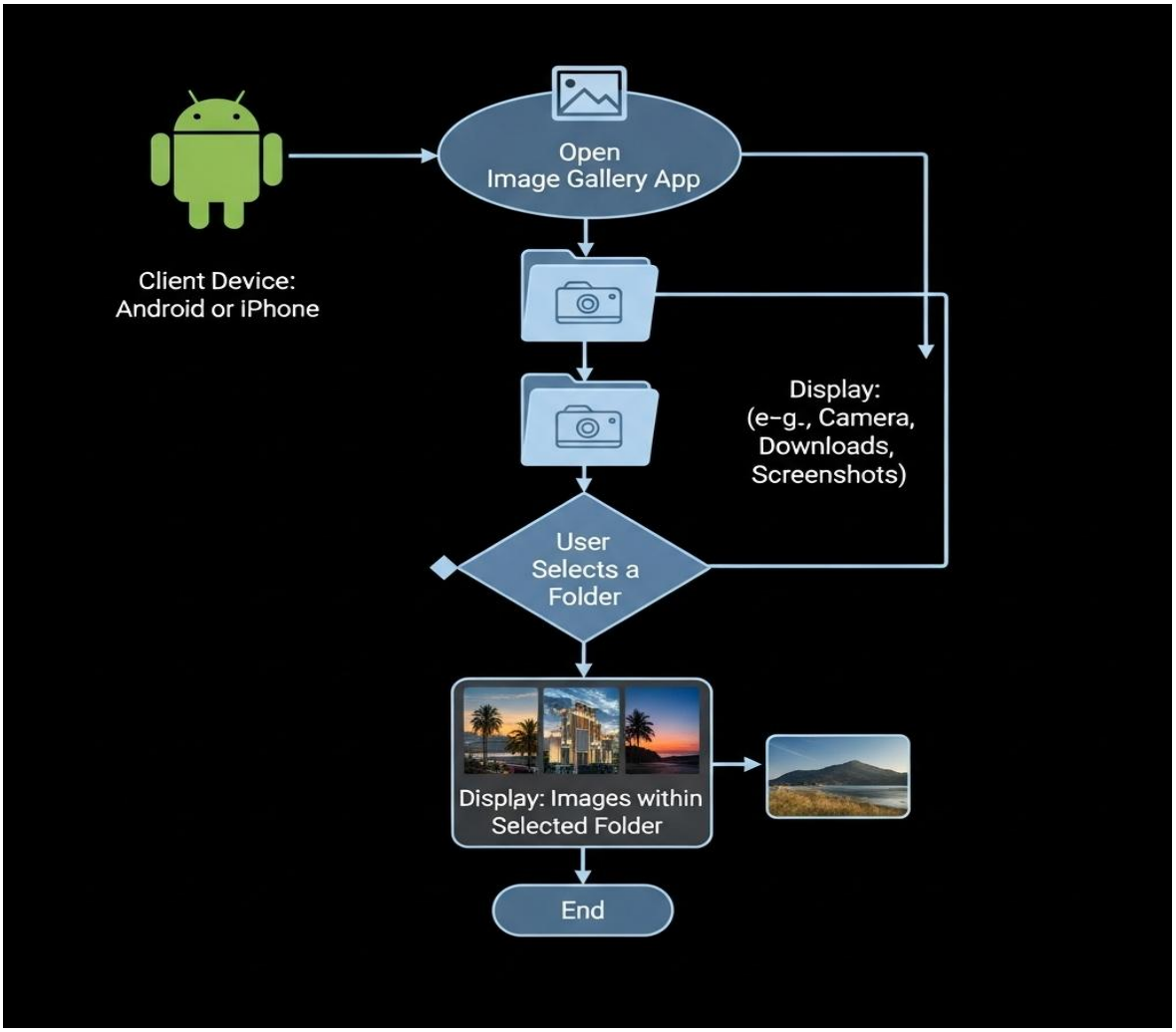

Fig 4.1:-Use Case Daigram

This flowchart outlines the initial steps a user takes to access images on their device:

1. Client Device (Android or iPhone): This is the starting point, indicating the application runs on either mobile platform.

2. Open Image Gallery App: The user's action to launch the gallery application.

3. Display Folders (e.g., Camera, Downloads, Screenshots): Once opened, the app presents a list of image-containing folders found on the device. This is crucial for organizing and categorizing images.

4. User Selects a Folder: The user interacts with the interface to choose a specific folder they wish to view.

SEQUENCE DIAGRAM :

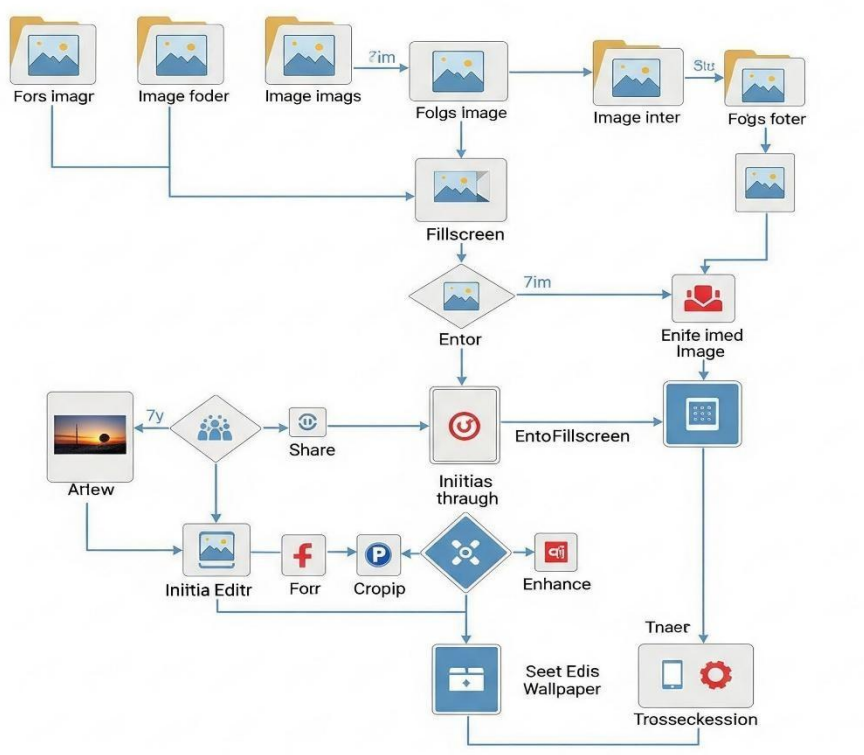

Fig 4.2:-Sequence Diagram(Flow Chart)

1. **Fors imagr / Image foder / Image imags (Folders/Images):** These likely represent the initial Browse of images and folders, leading to a selected image.
2. **Folgs image (Fullscreen Image):** This state indicates an image is now being viewed individually, likely taking up the entire screen.

3. **Entor (Enter/Interaction Point):** This diamond shape suggests a decision or action point for the user.
4. **Enite imed Image (Edit Image):** This path leads to editing functionalities.
    - **Inittias thrugh / Initia Editr (Initiate Editor):** Begins the editing process. ○ **Croppip (Crop):** A specific editing function, like cropping the image.
    - **Enhance:** Another editing function, likely for adjusting brightness, contrast, etc.
5. **Share:** An option to share the currently viewed image (e.g., via social media, messaging apps).
6. **Seet Edis Wallpaper (Set as Wallpaper):** An option to set the image as the device's wallpaper.
7. **Trosskeccssion (More Options/Settings):** This likely represents a broader menu of additional actions, similar to the "More Options" in the Flutter code.

CHAPTER-5

SOURCE CODE

Aim:- ***To implement and create an image Gallery Android app using flutter and dart in android Studio.***

Explanation:-

An image gallery app using Flutter allows users to view, pick, and manage images in an organized way. Built with Flutter's powerful UI toolkit, the app typically includes features like selecting images from the device using the image picker, displaying them in a grid layout with GridView, and opening them in full screen on tap. The interface is responsive and smooth, making it easy to navigate and view multiple images. Flutter's widgets make it simple to design a clean, interactive, and visually appealing gallery experience.

Source code(dart file):-

```
import 'dart:convert';
import 'package:flutter/material.dart';
import 'package:flutter/services.dart' show rootBundle;
import 'package:photo_view/photo_view.dart';
import 'package:photo_view/photo_view_gallery.dart';
import 'package:share_plus/share_plus.dart';
void main() {
 runApp(const ImageGalleryApp());
}
class ImageGalleryApp extends StatelessWidget {
 const ImageGalleryApp({super.key});
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   title: 'Image Folders Gallery',
   theme: ThemeData(primarySwatch: Colors.blue),
   home: const GalleryScreen(),
   debugShowCheckedModeBanner: false,
  );
```

```
 }
}
class GalleryScreen extends StatefulWidget {
 const GalleryScreen({super.key});
 @override
 State<GalleryScreen> createState() => _GalleryScreenState();
}
class _GalleryScreenState extends State<GalleryScreen> {
 Map<String, List<String>> folderImages = {};
 @override
 void initState() {
  super.initState();
  loadImages();
 }
 Future<void> loadImages() async {
  final manifestContent = await rootBundle.loadString('AssetManifest.json');
  final Map<String, dynamic> manifestMap = json.decode(manifestContent);
  final imagePaths = manifestMap.keys
      .where((String key) => _isImageFile(key))
      .toList();

  final Map<String, List<String>> folders = {};
  for (var path in imagePaths) {
   final folder = path.split('/')[1];
   folders.putIfAbsent(folder, () => []);
   folders[folder]!.add(path);
  }
  setState(() {
   folderImages = folders;
  });
 }
 bool _isImageFile(String path) {
  final lower = path.toLowerCase();
  return lower.endsWith('.png') ||
      lower.endsWith('.jpg') ||
      lower.endsWith('.jpeg') ||
```

```
      lower.endsWith('.gif');
}
@override
Widget build(BuildContext context) {
 return Scaffold(
  appBar: AppBar(
   title: const Text('Gallery'),
  ),
  body: folderImages.isEmpty
      ? const Center(child: CircularProgressIndicator())
      : ListView.builder(
    itemCount: folderImages.keys.length,
    itemBuilder: (context, index) {
     final folderName = folderImages.keys.elementAt(index);
     final images = folderImages[folderName]!;
     return Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
       Padding(
        padding:
        const EdgeInsets.symmetric(horizontal: 12, vertical: 8),
        child: Text(
         folderName.toUpperCase(),
         style: const TextStyle(
          fontSize: 20,
          fontWeight: FontWeight.bold,
         ),
        ),
       ),
       GridView.builder(
        shrinkWrap: true,
        physics: const NeverScrollableScrollPhysics(),
        padding: const EdgeInsets.symmetric(horizontal: 8),
        gridDelegate:
        const SliverGridDelegateWithFixedCrossAxisCount(
         crossAxisCount: 3,
```

```
            crossAxisSpacing: 4,
            mainAxisSpacing: 4,
          ),
          itemCount: images.length,
          itemBuilder: (context, imgIndex) {
            final imagePath = images[imgIndex];
            return GestureDetector(
              onTap: () {
                Navigator.push(
                  context,
                  MaterialPageRoute(
                    builder: (_) => FullscreenGallery(
                      images: images,
                      initialIndex: imgIndex,
                    ),
                  ),
                );
              },
              child: ClipRRect(
                borderRadius: BorderRadius.circular(6),
                child: Image.asset(
                  imagePath,
                  fit: BoxFit.cover,
                ),
              ),
            );
          },
        ),
      ],
    );
  },
 ),
 );
 }
}
class FullscreenGallery extends StatefulWidget {
```

```
 final List<String> images;
 final int initialIndex;
 const FullscreenGallery({
  super.key,
  required this.images,
  required this.initialIndex,
 });
 @override
 State<FullscreenGallery> createState() => _FullscreenGalleryState();
}
class _FullscreenGalleryState extends State<FullscreenGallery> {
 late PageController _pageController;
 late int currentIndex;
 Set<String> favorites = {};
 @override
 void initState() {
  super.initState();
  currentIndex = widget.initialIndex;
  _pageController = PageController(initialPage: currentIndex);
 }
 @override
 void dispose() {
  _pageController.dispose();
  super.dispose();
 }
 void onPageChanged(int index) {
  setState(() {
   currentIndex = index;
  });
 }
 void _shareImage() {
  final imagePath = widget.images[currentIndex];
  Share.share('Check out this image: $imagePath');
 }
 void _deleteImage() {
  setState(() {
```

```
    widget.images.removeAt(currentIndex);
    if (currentIndex >= widget.images.length) {
      currentIndex = widget.images.length - 1;
    }
  });
  if (widget.images.isEmpty) {
    Navigator.pop(context);
  }
}
void _toggleFavorite() {
  final imagePath = widget.images[currentIndex];
  setState(() {
    if (favorites.contains(imagePath)) {
      favorites.remove(imagePath);
    } else {
      favorites.add(imagePath);
    }
  });
}
void _showDetails() {
  final imagePath = widget.images[currentIndex];
  final imageName = imagePath.split('/').last;
  const imageSize = '2.3 MB';
  const dateTime = '5 July 2025, 10:25 AM';
  const location = 'Hyderabad, India';
  showDialog(
    context: context,
    builder: (_) => AlertDialog(
      title: const Text('Image Details'),
      content: Column(
        mainAxisSize: MainAxisSize.min,
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          Text('Name: $imageName'),
          Text('Size: $imageSize'),
          Text('Date & Time: $dateTime'),
```

```
        Text('Location: $location'),
       ],
      ),
      actions: [
       TextButton(
        onPressed: () => Navigator.pop(context),
        child: const Text('Close'),
       )
      ],
    ),
  );
}
void _editImage() {
  showModalBottomSheet(
    context: context,
    builder: (context) => Wrap(
      children: [
       _bottomItem(Icons.crop, 'Framing'),
       _bottomItem(Icons.tune, 'Adjust'),
       _bottomItem(Icons.face_retouching_natural, 'Beauty'),
       _bottomItem(Icons.brush, 'Erase'),
       _bottomItem(Icons.filter, 'Filters'),
       _bottomItem(Icons.border_outer, 'Border'),
      ],
    ),
  );
}
void _moreOptions() {
  showModalBottomSheet(
    context: context,
    builder: (context) => Wrap(
      children: [
       _bottomItem(Icons.wallpaper, 'Set as Wallpaper'),
       _bottomItem(Icons.search, 'Google Lens'),
       _bottomItem(Icons.picture_as_pdf, 'Super Document'),
       _bottomItem(Icons.copy, 'Copy to'),
```

```
        _bottomItem(Icons.visibility_off, 'Hide'),
        _bottomItem(Icons.edit, 'Rename'),
        _bottomItem(Icons.slideshow, 'Slideshow'),
      ],
    ),
  );
}
ListTile _bottomItem(IconData icon, String title) {
  return ListTile(
    leading: Icon(icon),
    title: Text(title),
    onTap: () {
      Navigator.pop(context);
      ScaffoldMessenger.of(context)
          .showSnackBar(SnackBar(content: Text('$title selected.')));
    },
  );
}
@override
Widget build(BuildContext context) {
  final imagePath = widget.images[currentIndex];
  final isFavorite = favorites.contains(imagePath);
  return Scaffold(
    appBar: AppBar(
      title: Text('Image ${currentIndex + 1} / ${widget.images.length}'),
      actions: [
        IconButton(
          icon: Icon(isFavorite ? Icons.star : Icons.star_border),
          onPressed: _toggleFavorite,
        ),
        IconButton(
          icon: const Icon(Icons.info_outline),
          onPressed: _showDetails,
        ),
      ],
    ),
```

```
body: Stack(
 children: [
  PhotoViewGallery.builder(
   itemCount: widget.images.length,
   pageController: _pageController,
   onPageChanged: onPageChanged,
   builder: (context, index) {
    return PhotoViewGalleryPageOptions(
     imageProvider: AssetImage(widget.images[index]),
     initialScale: PhotoViewComputedScale.contained,
     minScale: PhotoViewComputedScale.contained * 0.8,
     maxScale: PhotoViewComputedScale.covered * 3.0,
     heroAttributes:
     PhotoViewHeroAttributes(tag: widget.images[index]),
    );
   },
   scrollPhysics: const BouncingScrollPhysics(),
   backgroundDecoration: const BoxDecoration(color: Colors.black),
  ),
  Positioned(
   bottom: 20,
   left: 20,
   right: 20,
   child: Container(
    padding: const EdgeInsets.symmetric(horizontal: 12, vertical: 8),
    decoration: BoxDecoration(
     color: Colors.black.withAlpha((0.4 * 255).toInt()),
     borderRadius: BorderRadius.circular(16),
    ),
    child: Row(
     mainAxisAlignment: MainAxisAlignment.spaceAround,
     children: [
      IconButton(
       icon: const Icon(Icons.share, color: Colors.white),
       onPressed: _shareImage,
      ),
```

```
            IconButton(
              icon: const Icon(Icons.edit, color: Colors.white),
              onPressed: _editImage,
            ),
            IconButton(
              icon: const Icon(Icons.delete, color: Colors.white),
              onPressed: _deleteImage,
            ),
            IconButton(
              icon: const Icon(Icons.more_vert, color: Colors.white),
              onPressed: _moreOptions,
            ),
          ],
        ),
      ),
    )
  ],
 ),
 );
 }
}
```

Pubspec.yaml:-

```
name: image_gallery
description: A simple Flutter image gallery app.
publish_to: 'none'
version: 1.0.0+1
environment:
 sdk: '>=3.0.0 <4.0.0'
dependencies:
 flutter:
  sdk: flutter
 photo_view: ^0.15.0
 share_plus: ^7.0.2
dev_dependencies:
 flutter_test:
  sdk: flutter
flutter:
    uses-material-design: true
    assets:
      - assets/camera/
      - assets/images/
```

CHAPTER-6

OUTPUT

Fig 6.1:-The emulator screen

Fig 6.2:-Starting the Flutter app

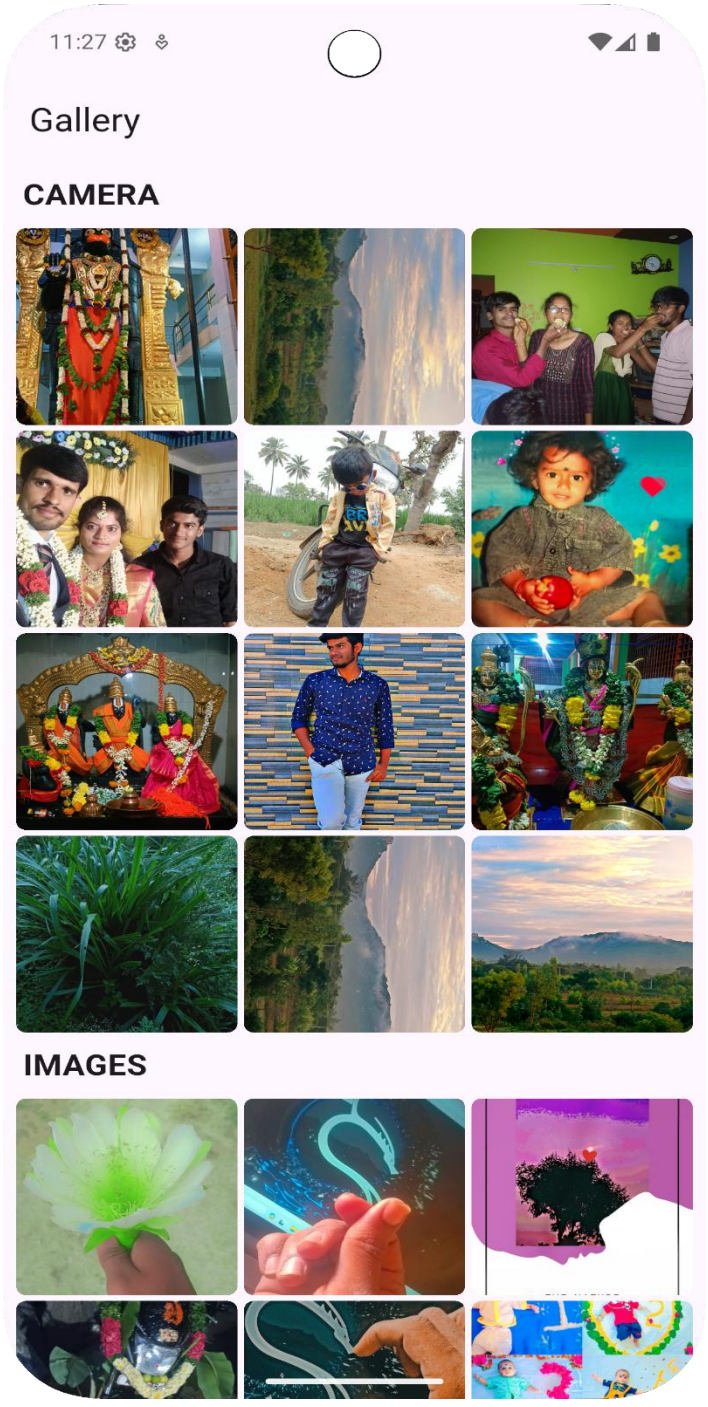

Fig 6.3:-Open the app display the images

Fig 6.4:-Open one picture

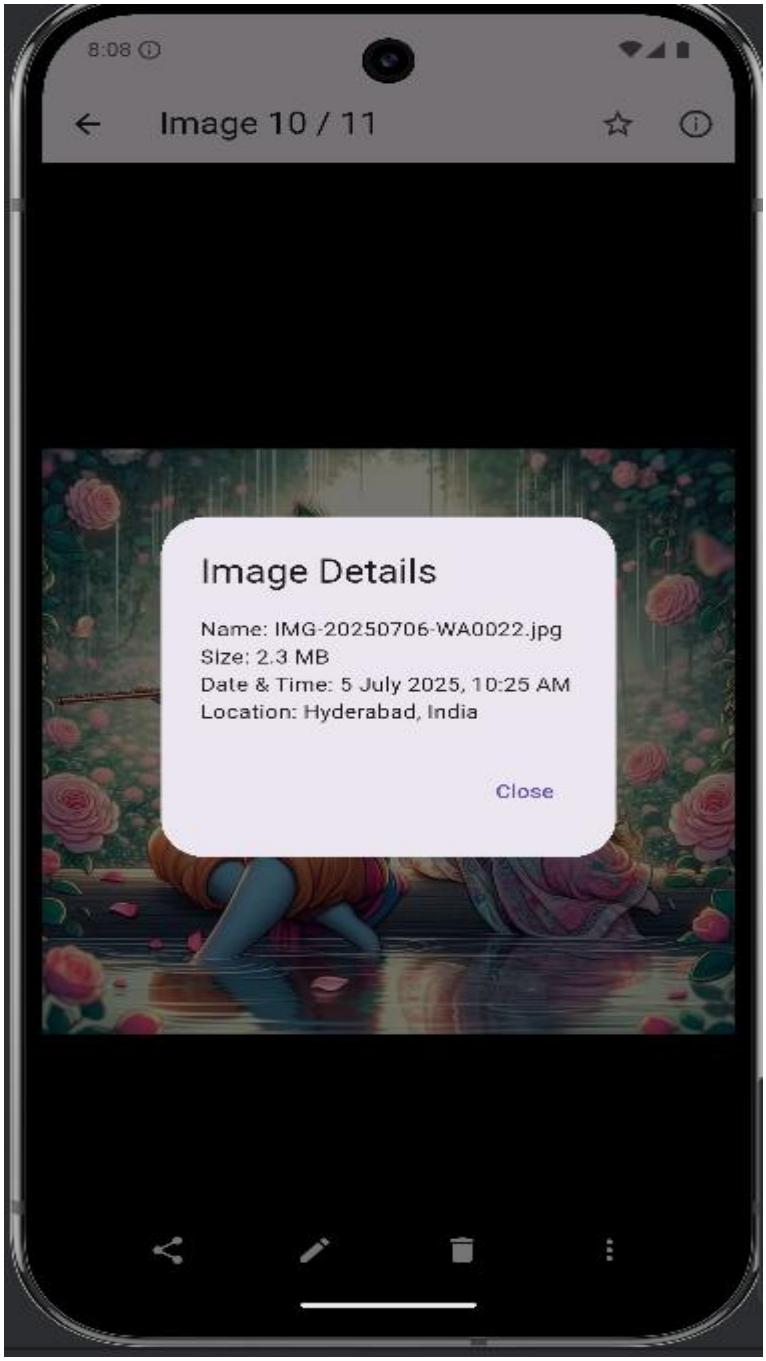

Fig 6.5:-View details of image

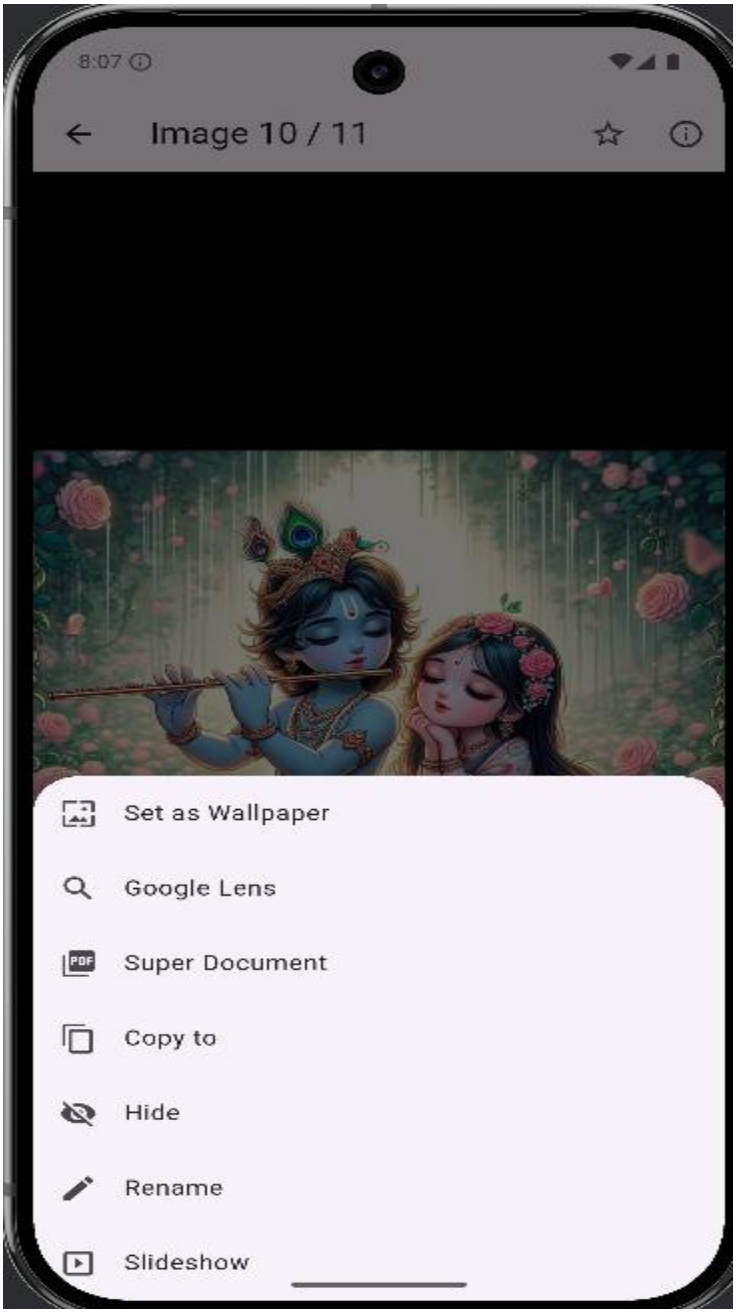

Fig 6.6:- view more options of image

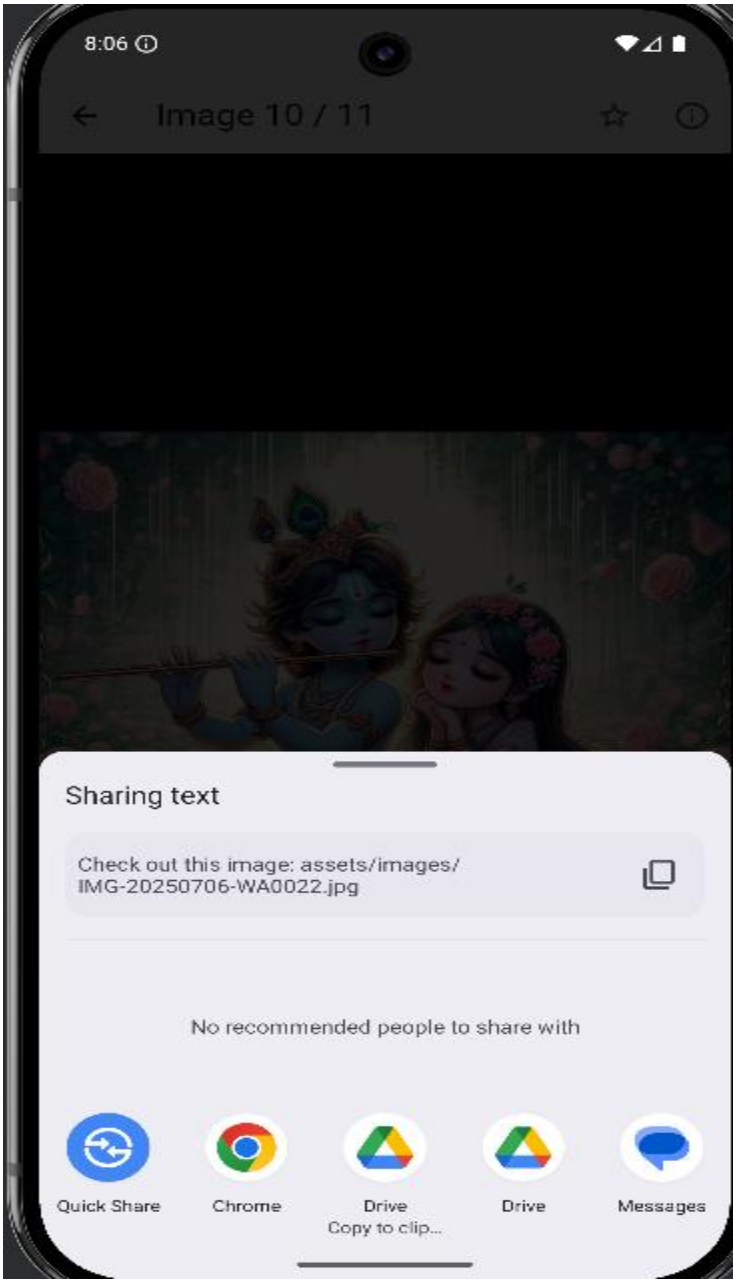

Fig 6.7:- Share the image

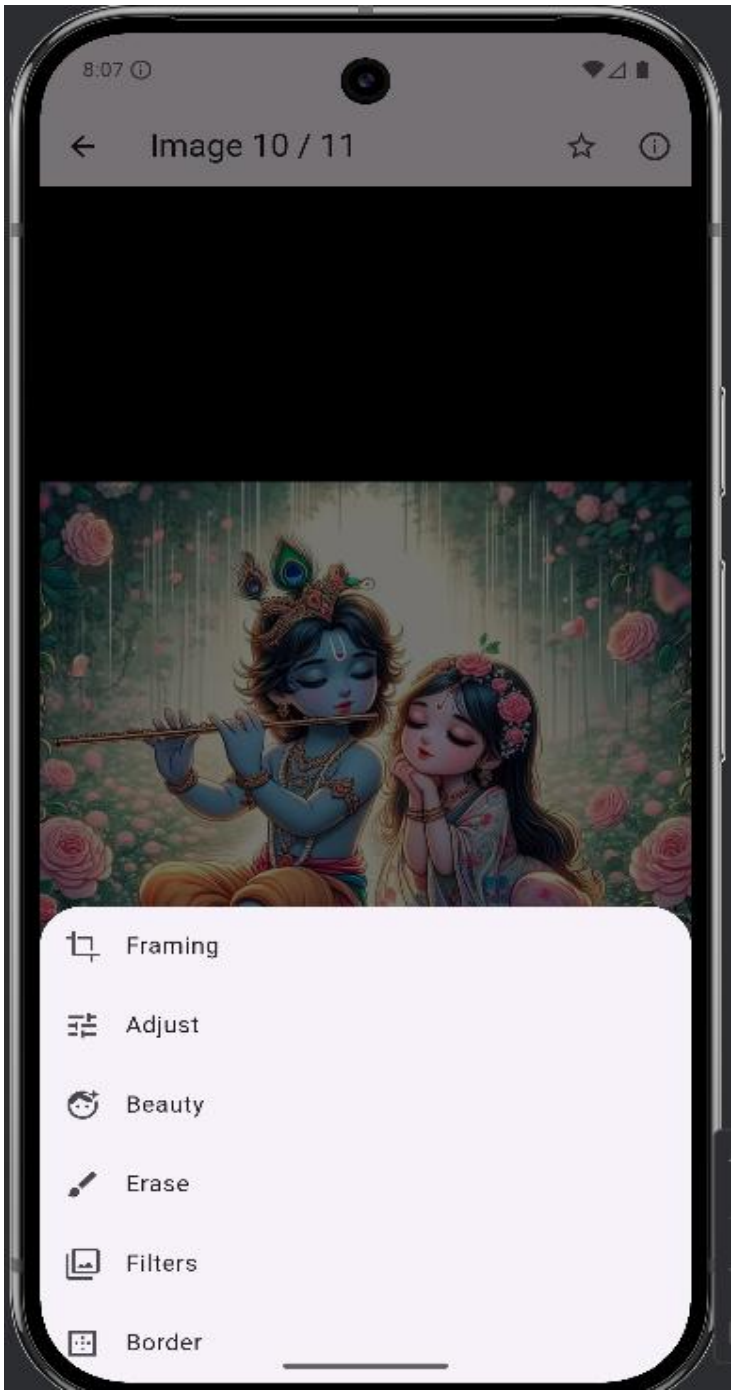

Fig 6.8:-Editing of image

CHAPTER-7

CONCLUSION

🟢 Conclusion for Image Gallery App Using Flutter (Without Firebase):

The **Image Gallery App** developed using **Flutter** (without Firebase) showcases the powerful capabilities of Flutter for building clean, efficient, and responsive user interfaces for mobile platforms. By utilizing **local device storage** and **Flutter plugins** like image_picker, path_provider, and gallery_saver, the app successfully allows users to:

- **Capture images** using the camera or select images from the device's gallery.
- **Store and retrieve images** locally without the need for cloud services.
- **Display images in a structured layout** such as GridView, providing a seamless gallery experience.
- **Provide basic UI interactivity** through features like adding, deleting, or viewing images full-screen.

This approach is particularly suitable for apps that:

- Need **offline functionality** (no internet required),
- Require **privacy** (user images stay on the device),
- Target **low-resource** environments where backend integration may not be feasible.

Overall, this app demonstrates how Flutter, combined with efficient use of native device features, can create a robust image gallery experience **without relying on Firebase or external databases**. It lays a strong foundation for extending to more advanced features like image editing, categorization, or cloud syncing in the future, if needed.

✅ Detailed Conclusion – Image Gallery App Using Flutter (Without Firebase)

The **Image Gallery App** built with **Flutter** and **device-based storage** is a lightweight, efficient, and privacy-focused mobile application. It leverages Flutter's cross-platform capabilities to deliver a visually appealing and highly responsive user experience, while completely avoiding dependency on cloud services like Firebase.

🌟 **Key Benefits:**

1. **Offline Functionality**:
   Fully usable without an internet connection – perfect for offline-first environments.
2. **Privacy and Data Control**:
   Since data is stored only on the user's device, it eliminates the need for user authentication, cloud storage permissions, or external APIs.
3. **Performance Optimization**:
   Fast loading and smooth transitions, as local access is quicker than network operations.
4. **Ease of Maintenance**:
   No need to manage backend services, databases, or real-time syncing complexities.

⚠️ **Limitations:**

- **Limited Backup**:
  Data is lost if the app is uninstalled or device is reset, unless manually backed up.
- **Scalability Constraints**:
  Managing thousands of images on local storage can become cumbersome without proper caching and indexing.
- **No Remote Access**:
  Images are only accessible from the device they're stored on, unlike cloud-based apps.

🚀 **Future Enhancement Ideas:**

1. **Add Image Categories/Tags**:
   Help users organize images more effectively.
2. **Basic Image Editing**:
   Include cropping, rotating, or applying filters using packages like image or flutter_image_editor.
3. **Slideshow Feature**:
   A fun and interactive way to view the image collection.

4. **Data Export/Import**:
   Allow users to export images to cloud or import them from external sources (like SD cards).
5. **Optional Cloud Syncing**:
   In future versions, optionally integrate Firebase, Supabase, or Google Drive sync to offer backup and multi-device access.

📌 **Final Thoughts:**

This app proves how **Flutter**, even without Firebase, is fully capable of building a complete and functional photo gallery. It is ideal for **offline-first use cases**, provides **better privacy**, and is a **great foundation** for developers looking to learn local storage management and image handling in Flutter. With some enhancements, it can evolve into a powerful gallery or photo management tool tailored for niche or personal use.

CHAPTER-8

GITHUB PROFILE

https://github.com/arunkumarreddyk890/Imagegallery

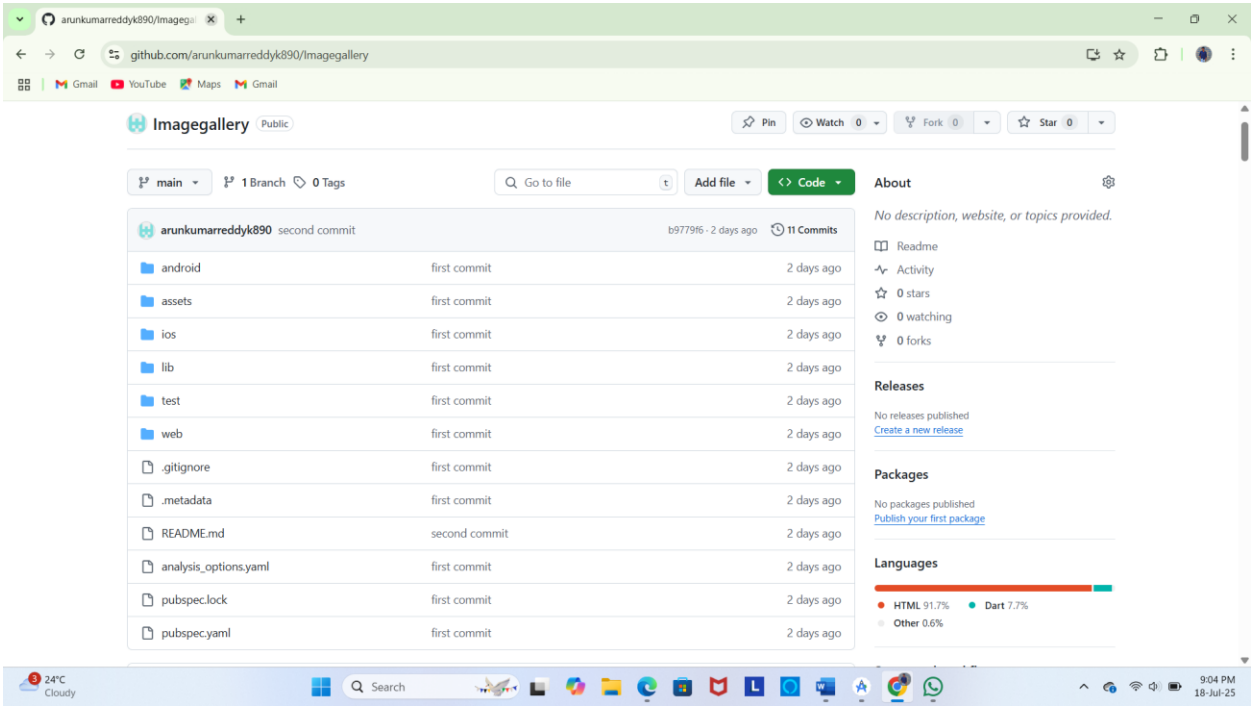

Fig 8.1:- Github uploaded project

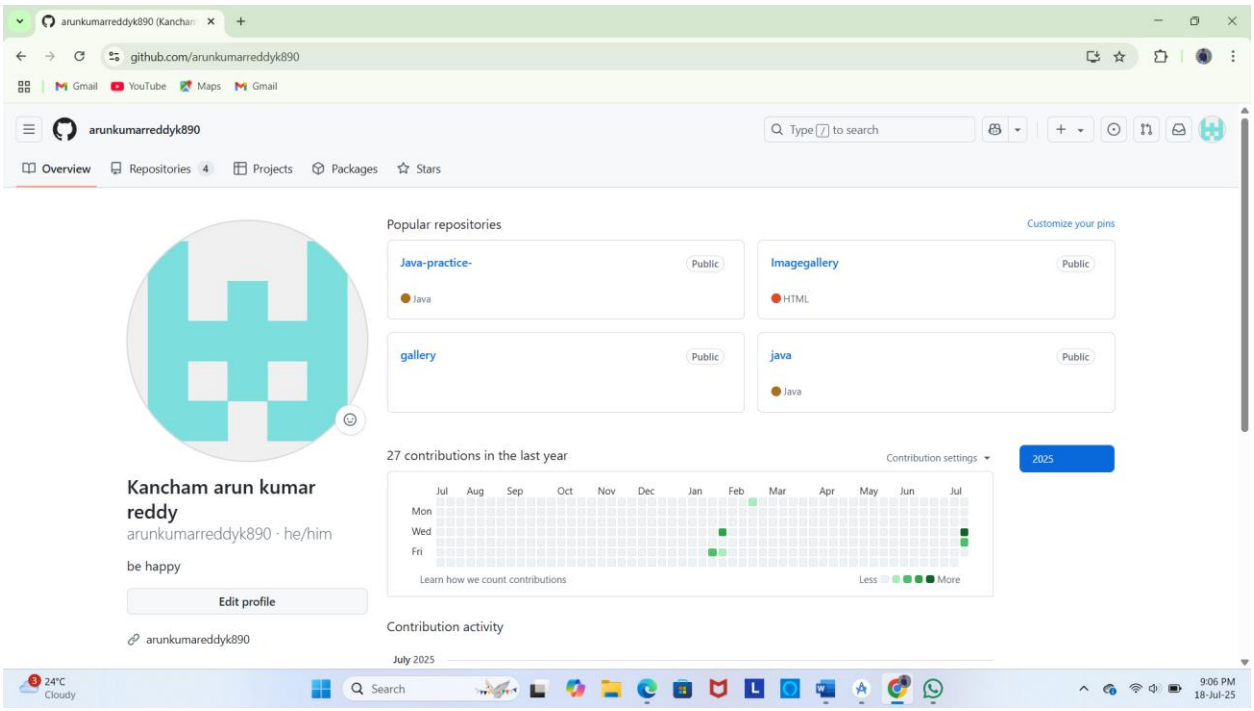

Fig 8.2:-Github profile