Notes

▼ **Syllabus**

Q1 - Activities and Fragments (Android), Stateful and Stateless Widgets (Flutter), Android Layouts (Android), Row and Column (Flutter)

Q2 - Widgets (`TextField`, `FormField`, Validators, `RecyclerView`) (first 3 flutter, recycleview android), Async and await callbacks and background services (Flutter), Navigator and Routes (flutter)

Q3 - Shared Preferences and Local DB (Android), Firebase (Flutter), Code Firebase Auth Firestore or Shared Preferences (Flutter), DB connectivity, Hosting Application Play Store, Loader and Permissions (Android)

Q4 - `pubspec.yaml`, Light/dark mode code, Intents (Android), User Control (permissions) (Android), MVC Pattern (Flutter), CRUD, Internal DB, Null Safety (Android), Android Studio Folder Structure (Android/Flutter), Handling Images (Flutter)

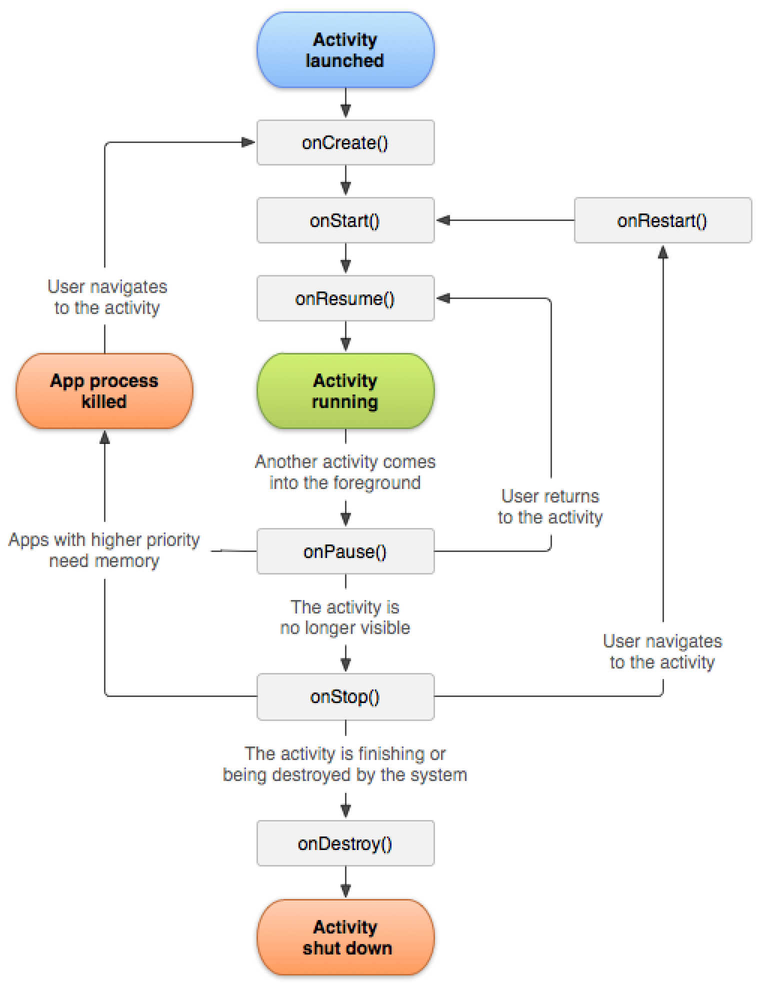
Activities

💡 An activity is a single, focused thing that the user can do.

- An activity provides the window in which the app draws its UI.
- Most apps contain multiple screens, which means they comprise multiple activities. Typically, one activity in an app is specified as the main activity.
- Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI with `setContentView(View)`
- Activities are often presented to the user as full-screen windows, they can also be used in other ways: as floating windows, multi-window mode or embedded into other windows.
- Two methods all subclasses of Activity will implement
  1. `onCreate(bundle)`: Where you initialize your activity
  2. `onPause()`: Where you deal with the user pausing active interaction with the activity

Activity Lifecycle

- Activities in the system are managed as activity stacks.
- When a new activity is started, it is usually placed on the top of the current stack and becomes the running activity.
- An activity has 4 states:
  - **Active or Running**: If an activity is in the foreground of the screen (on top of topmost stack).
  - **Visible**: If an activity has lost focus but is still presented to the user
  - **Hidden**: If an activity is completely obscured by another activity
  - **Destroyed**: The system can drop the activity from memory by either asking it to finish, or simply killing its process.


Activity launched
onCreate()
onStart()
onRestart()
onResume()
User navigates to the activity
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

- **Activity Lifecycle Methods**
  1. `onCreate()`: Called when the activity is first created.
  2. `onRestart()`: Called after your activity has been stopped, prior to it being started again.
  3. `onStart()`: Called when the activity is becoming visible to the user.
  4. `onResume()`: Called when the activity will start interacting with the user.
  5. `onPause()`: Called when the activity loses foreground state, is no longer focusable or before transition to stopped/hidden or destroyed state.
  6. `onStop()`: Called when the activity is no longer visible to the user.
  7. `onDestroy()`: The final call you receive before your activity is destroyed.
     - If the activity is finishing
     - If the system is temporarily destroying this instance of the activity to save space

Fragments

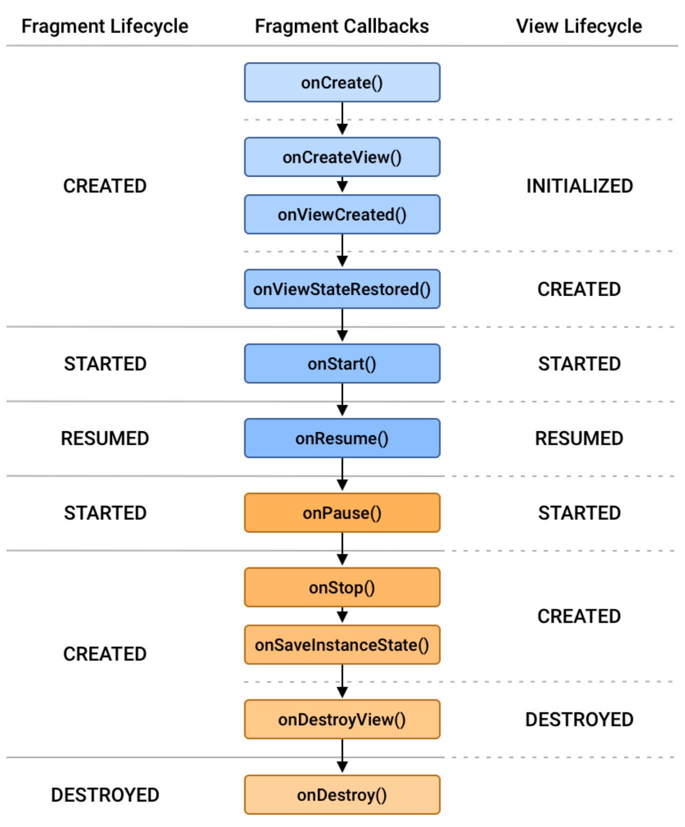
💡 A Fragment represents a reusable portion of your app’s UI.

- A fragment defines and manages its own layout, has its own lifecycle, and can handle its own input events.
- Fragments must be hosted by an activity or another fragment.
- The fragment’s view hierarchy becomes part of the host’s view hierarchy.
- Fragments introduce modularity and reusability into your activity’s UI by letting you divide the UI into discrete chunks.

Fragment Lifecycle

- A fragment has 5 states:
  - Initialized
  - Created
  - Started
  - Resumes

- Destroyed



- **Fragment Lifecycle Methods**
    1. `onAttach(Activity a)`: Called when the fragment has been added to a `FragmentManager` and is attached to its host activity.
    2. `onCreate()`: Called to do initial creation of a fragment.
    3. `onCreateView()`: Called to have the fragment instantiate its user interface view (Optional)
    4. `onViewCreated()`: Called immediately after `onCreateView()` has returned, but before any saved state has been restored in to the view.
    5. `onViewStateRestored()`: Called when all saved state has been restored into the view hierarchy of the fragment.
    6. `onStart()`: Called when the Fragment is visible to the user.
    7. `onResume()`: Called when the fragment is visible to the user and actively running.
    8. `onPause()`: Called when the Fragment is no longer resumed.
    9. `onStop()`: Called when the Fragment is no longer started.
    10. `onSaveInstanceState()`: Called to ask the fragment to save its current dynamic state, so it can later be reconstructed.
    11. `onDestroyView()`: Called when the view previously created by `onCreateView()` has been detached from the fragment.
    12. `onDestroy()`: Called when the fragment is no longer in use.
    13. `onDetach()`: Called when the fragment is no longer attached to its host activity.

Activities vs Fragments

	Activity	Fragment
Definition	An activity is a single, focused thing that the user can do	A Fragment is a reusable portion of your app's UI
Represents	It represents a single screen with a user interface	It represents a modular part of an Activity's UI, which can be combined to create complex and flexible UI layouts
Independent	Activities can exist without Fragments	Fragments need to be attached to an Activity
Nesting	Cannot be nested within each other	Can be nested within Activities and other Fragments
UI Update	Entire UI is updated at once	Can update only specific portions of UI
Reusability	Generally less reusable	Highly reusable, can be used in multiple Activities
manifest.xml	Must be mentioned in `manifest.xml`	Need not be mentioned in `manifest.xml`

	Activity	Fragment
Weight	Not light-weight	Light-weight
Lifecycle Management	Manage their own lifecycle	Lifecycle depends on the Activity in which it is hosted
Display	Display only one activity to the user	Display multiple fragments at a time in a single Activity
Navigation	Involves intents and back stack	Involves `FragmentManager` and the back stack

Widgets

💡 Each element on a screen of the Flutter app is a widget.

- Flutter widgets are built using a modern framework that takes inspiration from React.
- The central idea is that you build your UI out of widgets.
- Widgets describe what their view should look like given their current configuration and state.
- State is information that
  - can be read synchronously when the widget is built
  - might change during the lifetime of the widget

Stateless Widget

💡 The widgets whose state can not be altered once they are built are called stateless widgets.

- The only area of focus of a stateless widget is the information displayed and the UI.
- They deal with actions that do not depend on the user’s input.
- They create UIs that do not change dynamically upon updation in the nearby values.

```
class MyWidget extends StatelessWidget {
  const MyWidget({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Container(
         child: Text('Hello World'),
    );
  }
}
```

- Every widget while entering the `Widget build(Build Context context)` has to override it in order to enter the widget tree.
- Build Method is called inside a stateless widget in only three cases:
  1. Initially when it is built for the very first time
  2. If the parent widget is changed
  3. If the inherited widget is changed

Stateful Widget

💡 A Stateful Widget has its own mutable state that it needs to track.

- Stateful widgets are useful when the part of the UI you are describing can change dynamically.
- Stateful widget can change when there is
  - user input included
  - user interaction
  - dynamic change

```
class MyApp extends StatefulWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  State<MyApp> createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
     @override
   Widget build(BuildContext context) {
    return Container(
         child: Text('Hello World'),
    );
  }
}
```

- **Methods**
  1. `setState()`: A State object is used to modify the UI.
  2. `initState()`: This method is the entry point of a widget. Called only once.
  3. `didChangeDependencies()`: Used for loading the dependencies required for execution of a state.
  4. `dispose()`: Used for removing an object permanently from the widget tree.

Stateless vs Stateful Widgets

	Stateless Widgets	Stateful Widgets
State	No internal state; immutable	Have mutable state, which can change over time
Rebuild	Rebuilds entirely on parent's `setState()` call	Rebuilds only when `setState()` is called
Usage	Suitable for static UI components	Suitable for UI components with dynamic or changing content
Performance	Generally more performant	May have slightly lower performance
Memory	Generally less memory usage	May use more memory due to state management
Rendering	Constructed once and remain unchanged	Constructed once, but can update over time
Reusability	Highly reusable across different screens	Less reusable due to internal state management
Examples	Text, Icon, Image	Form, List, Interactive UI elements

---

TextField

💡 A text field lets the user enter text, either with hardware keyboard or with an onscreen keyboard.

- They are used to build forms, send messages, create search experiences, and more.
- The text field calls the `onChanged` callback whenever the user changes the text in the field.
- To control the text that is displayed in the text field, use the `controller`.

- If the user indicates that they are done typing in the field, the text field calls the `onSubmitted` callback.
- You can use the `decoration` property to control the decoration/styling of the text field.

```
TextEditingController emailController = TextEditingController();

TextField(
        onChanged: (text) { print(text); },
        controller: emailController,
        obscureText: false,
        decoration: InputDecoration(
                prefixIcon: const Icon(Icons.email),
                border: InputBorder.none,
                labelText: "Email"
        hintText: "abc@mail.com",
        hintStyle: const TextStyle(
                  color: Colors.black,
        ),
    ),
    maxLines: 4,
    maxLength: 100,
);
```

TextFormField

Validators

RecyclerView

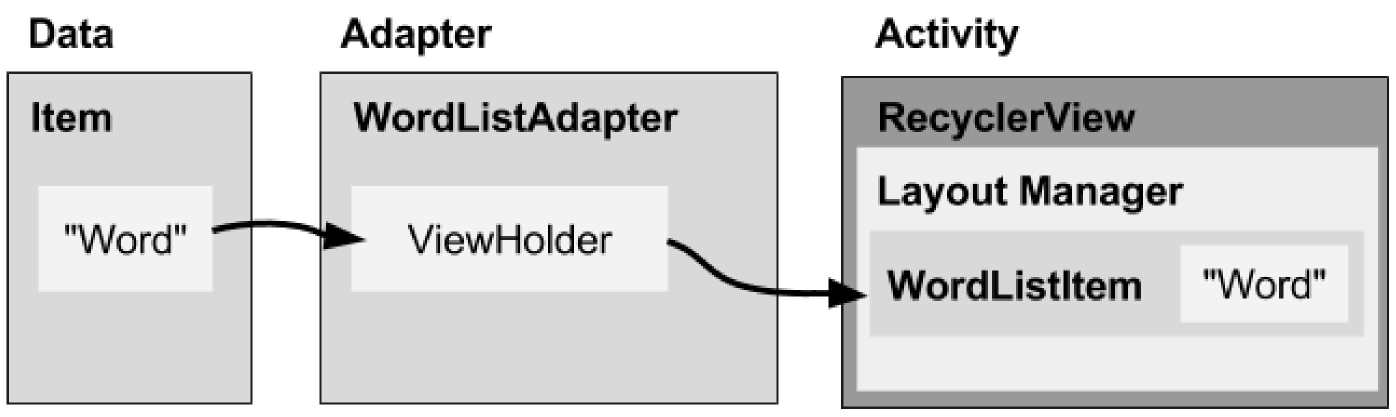
💡 `RecyclerView` is a container for displaying large, scrollable data sets efficiently by maintaining a limited number of `View` items.

- A flexible and efficient view for providing a limited window into a large data set.
- A more advanced and flexible version of `ListView`
- Use `RecyclerView` when you need to display
    - a large amount of scrollable data
    - data collections whose elements change at runtime (user actions or network events)

Components

1. **Data**
    - Any displayable data (text, images, icons)
    - Can come from any source (created by app, local database, or cloud storage)
2. **RecyclerView**: An instance of `RecyclerView`, the scrolling list that contains the list items.
3. **Layout (for one data item)**: The item layout has to be created separately from the `Activity` layout.
4. **Layout manager**: The layout manager handles the organization (layout) of user interface components in a `View` (`LinearLayoutManager` `GridLayoutManager` `StaggeredGridLayoutManager`)

5. **Adapter**
    - Helps two incompatible interfaces to work together.
    - Intermediary between data and `View`
    - When the data changes, the adapter updates the contents of the respective list item view in the `RecyclerView`.
6. **ViewHolder**: To contain the information for displaying one `View` item using the item's layout.



Steps to implement `RecyclerView`

1. Add `RecyclerView` dependency to `build.gradle` if needed

```
dependencies {
...
compile 'com.android.support:recyclerview-v7:26.1.0'
...
}
```

2. Add `RecyclerView` to XML layout

```
<android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerview"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
>
</android.support.v7.widget.RecyclerView>
```

3. Create XML layout for item

```
<LinearLayout>
    <TextView android:id="@+id/word" style="@style/word_title" />
</LinearLayout>
```

4. Extend `RecyclerView.Adapter`

```
public class WordListAdapter
    extends RecyclerView.Adapter<WordListAdapter.WordViewHolder> {
    public WordListAdapter(Context context, LinkedList<String> wordList) {
        mInflater = LayoutInflater.from(context);
        this.mWordList = wordList;
    }
}
```

5. Extend `RecyclerView.ViewHolder`

```
class WordViewHolder extends RecyclerView.ViewHolder{
    // code
}
```

6. In Activity `onCreate()`, create `RecyclerView` with adapter and layout manager

```
mRecyclerView = findViewById(R.id.recyclerview);
mAdapter = new WordListAdapter(this, mWordList);
mRecyclerView.setAdapter(mAdapter);
mRecyclerView.setLayoutManager(new LinearLayoutManager(this));
```

Async and Await Callbacks

- Asynchronous operations let your program complete work while waiting for another operation to finish.
- Examples:
    - Fetching data over a network
    - Writing to a database
    - Reading data from a file
- Asynchronous programming allows developers to write code that appears sequential, making it easier to understand and maintain.
- Asynchronous computations usually provide their result as a `Future` (or `Stream`)
- In Dart, a `Future` represents a value that may not be available yet.
- To interact with these asynchronous results, you can use the `async` and `await` keywords.

💡 A future represents the result of an asynchronous operation, and can have two states: uncompleted or completed.

- **Two guidelines**
    1. To define an async function, add `async` before the function body
    2. The `await` keyword works only in `async` functions.
- If the function has a declared return type, then update the type to be `Future<T>`, where `T` is the type of the value that the function returns.

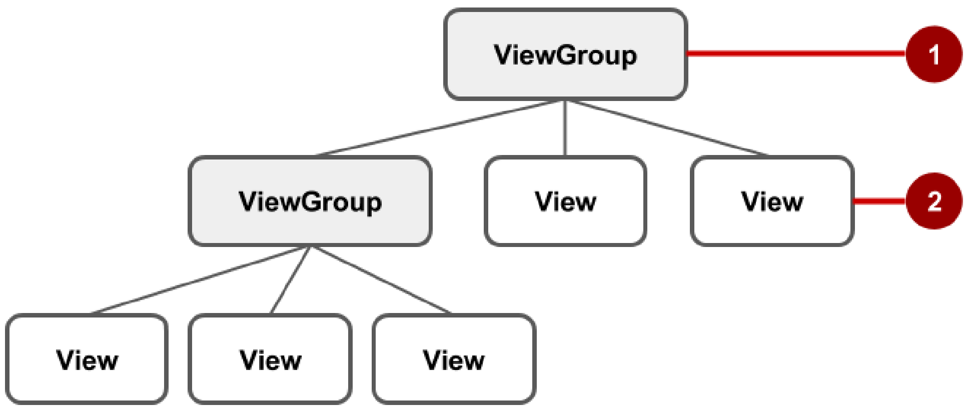
```
Future<void> func() async { ··· } // returns void
Future<int> func() async { ··· } // return int
```

- You can use the `await` keyword to wait for a future to complete

```
print(await func());
```

Android Layouts

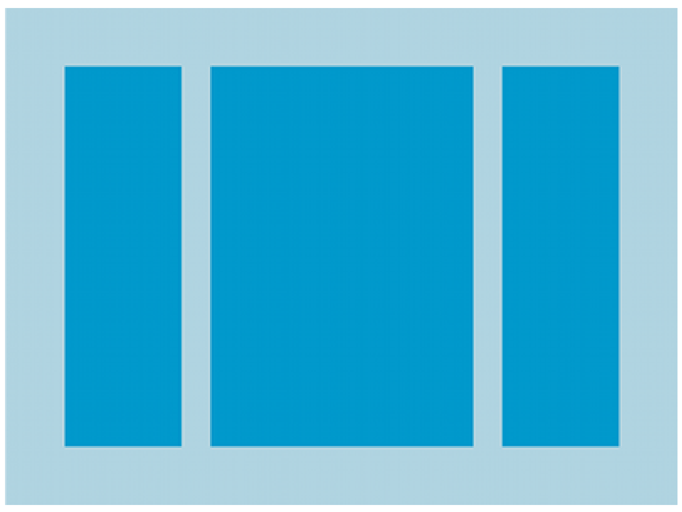
- The UI consists of a hierarchy of objects called views, every element of the screen is a `View`
- The `View` elements for a screen are organized in a hierarchy.
- At the top of this hierarchy is a `ViewGroup` that contains the layout of the entire screen.



- Some `ViewGroup` groups are designated as **layouts** because they organize child `View` elements in a specific way.
- Examples
  - `ConstraintLayout`
  - `LinearLayout`
  - `RelativeLayout`
  - `TableLayout`
  - `FrameLayout`
  - `GridLayout`

LinearLayout

💡 `LinearLayout` is a view group that aligns all children in a single direction, vertically or horizontally.



- All children of a `LinearLayout` are stacked one after the other, so a vertical list only has one child per row, no matter how wide they are.
- A horizontal list is only one row high, and it's the height of the tallest child, plus padding.
- `LinearLayout` respects margins between children, and the right, center, or left alignment of each child.

Attribute	Description
`android:orientation`	Specify the layout direction (vertical, horizontal)
`android:layout_weight`	Assign a value to a view in terms of how much space it occupies on the screen.
`android:padding_left` `android:padding_right`	Sets the padding, in pixels, of the left or right edge
`android:gravity`	To control how linear layout aligns all the views it contains (left, right, top, bottom, center)
`android:layout_height` `android:layout_width`	Specifies the basic height/width of the view.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    android:gravity="center">
```

```
  <!-- Include "child views" or "children" of the linear layout -->
</LinearLayout>
```

ConstraintLayout

💡 `ConstraintLayout` is a view group which allows you to position and size widgets in a flexible way.

- Relative positioning is one of the basic building blocks of creating layouts in `ConstraintLayout`.
- Those constraints allow you to position a given widget relative to another one.

```
<Button android:id="@+id/buttonA" ... />
<Button android:id="@+id/buttonB" app:layout_constraintLeft_toRightOf="@+id/buttonA" />
<!-- We want the left side of button B to be constrained to the right side of button A.  -->
```

Attribute	Description
`android:orientation`	Specify the layout direction (vertical, horizontal)
`android:layout_weight`	Assign a value to a view in terms of how much space it occupies on the screen.
`android:padding_left` `android:padding_right`	Sets the padding, in pixels, of the left or right edge
`android:gravity`	To control how linear layout aligns all the views it contains (left, right, top, bottom, center)
`android:layout_height` `android:layout_width`	Specifies the basic height/width of the view.

GridLayout

RelativeLayout

Row and Columns

- One of the most common layout patterns is to arrange widgets vertically or horizontally.
- You can use a `Row` widget to arrange widgets horizontally, and a `Column` widget to arrange widgets vertically.
- `Row` and `Column` are two of the most commonly used layout patterns.
- `Row` and `Column` each take a list of child widgets.
- A child widget can itself be a `Row`, `Column`, or other complex widget.
- You can specify how a `Row` or `Column` aligns its children, both vertically and horizontally.
- You can stretch or constrain specific child widgets.
- You can specify how child widgets use the `Row`’s or `Column`’s available space.
- You control how a row or column aligns its children using the `mainAxisAlignment` and `crossAxisAlignment` properties.
- `mainAxisAlignment`
    - How the children should be placed along the main axis in a flex layout.
    - start, end, center, spaceBetween, spaceAround, spaceEvenly
- `crossAxisAlignment`
    - How the children should be placed along the cross axis in a flex layout.
    - start, end, center, stretch, baseline
- **Drawbacks**
    - The row/column has no horizontal/vertical scrolling so when a large number of children are inserted in a single row/column that is not able to fit in the row/column then it will give us a Render Overflow message.

Navigators and Routes

- Flutter provides a complete system for navigating between screens and handling deep links.
- In Android, a route is equivalent to an `Activity`. In iOS, a route is equivalent to a `ViewController`. In Flutter, a route is just a widget.
- Small applications without complex deep linking can use `Navigator`
- The `Navigator` widget displays screens as a stack using the correct transition animations for the target platform.
- The `Navigator.push()` method adds a `Route` to the stack of routes managed by the `Navigator`.
- This method pushes the route on top of the first route, thereby displaying the second route.

```
// Within the First Page widget
onPressed: () {
  Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => const SecondPage()),
  );
}
```

- The `Navigator.pop()` method removes the current `Route` from the stack of routes managed by the `Navigator`.
- This helps us to remove the present route from the stack so that we go back to our first route.

```
// Within the Second Page widget
onPressed: () {
  Navigator.pop(context);
}
```

- Use a `MaterialPageRoute`, because it transitions to the new route using a platform-specific animation.

Shared Preferences

💡 Shared Preferences allow you to read and write small amounts of primitive data as key/value pairs to a file on the device storage.

- `SharedPreference` class provides APIs for reading, writing, and managing this data.
- Data is private to the application.
- Data persists across user sessions, even if app is killed and restarted, or device is rebooted.
- `SharedPreferences` is best suited to storing data about how the user prefers to experience the app, like
  - whether the user prefers a particular UI theme
  - whether they prefer viewing particular content in a list vs. a grid
- A `SharedPreferences` object points to a file containing key-value pairs and provides simple methods to read and write them.
- For managing large amounts of data, use an SQLite database.

Creating Shared Preferences

- Need only one Shared Preferences file per app
- Name it with package name of your app
- You create the shared preferences file in the `onCreate()`

```
private String sharedPrefFile =
   "com.example.android.hellosharedprefs";
mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
```

Saving Shared Preferences

- You save preferences in the `onPause()` state of the activity lifecycle.

```
@Override
protected void onPause() {
        super.onPause();
        SharedPreferences.Editor preferencesEditor = mPreferences.edit();
        preferencesEditor.putInt("count", 3);
        preferencesEditor.apply();
}
```

Reading Shared Preferences

- You read shared preferences in the `onCreate()` method of your activity.

```
mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
mCount = mPreferences.getInt("count", 1); // 1 is default value if count does not exist
```

Clearing Shared Preferenes

```
SharedPreferences.Editor preferencesEditor = mPreferences.edit();
preferencesEditor.clear();
preferencesEditor.apply();
```

Install Firebase

1. Install the Firebase CLI via npm `npm install -g firebase-tools`
2. Log into Firebase using your Google account `firebase login`
3. Install the FlutterFire CLI `dart pub global activate flutterfire_cli`
4. Use the FlutterFire CLI to configure your Flutter app to connect to Firebase `flutterfire configure`
5. From your Flutter project directory, install core plugin `flutter pub add firebase_core`
6. In your `lib/main.dart` file, import the Firebase core plugin and the configuration file
7. Initialize Firebase using the `DefaultFirebaseOptions`

   ```
   import 'package:firebase_core/firebase_core.dart';
   import 'firebase_options.dart';

   void main() async{
     await Firebase.initializeApp(
       options: DefaultFirebaseOptions.currentPlatform,
     );
     runApp(const MyApp());
   }
   ```

8. Rebuild Flutter app `flutter run`

Firebase Authentication

1. From the root of your Flutter project, install plugin `flutter pub add firebase_auth`
2. Rebuild Flutter app `flutter run`
3. Import plugin in your code

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

class AuthService{
  final FirebaseAuth _auth = FirebaseAuth.instance;

  //Sign in function
  Future<UserCredential> signInWithEmailAndPassword(String email, password) async {
    try{
      UserCredential userCredential = await _auth.signInWithEmailAndPassword(
              email: email,
              password: password
          );
      return userCredential;
    } on FirebaseAuthException catch (e) {
      throw Exception(e.code);
    }
  }

  //Sign up function
  Future<UserCredential> signUpWithEmailAndPassword(String name, email, password) async{
    try{
      UserCredential userCredential = await _auth.createUserWithEmailAndPassword(
              email: email,
              password: password
          );
      return userCredential;
    } on FirebaseAuthException catch (e) {
      throw Exception(e.code);
    }
  }

  //Sign out function
  Future<void> signOut() async{
    return await _auth.signOut();
  }
```

Firebase Firestore

1. From the root of your Flutter project, install plugin `flutter pub add cloud_firestore`
2. Rebuild Flutter app `flutter run`
3. Import plugin in your code

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

class FirestoreData{
  final FirebaseFirestore _firestore = FirebaseFirestore.instance;

  //Insert user data
  void insertUser(String name, email, password) async{
    _firestore.collection("users").add(
    {
            'name': name,
            'email': email,
            'password': password,
        });
  }

  //Get all users IDs
  void getUsers() async{
          await db.collection("users").get().then((event) {
                  for (var doc in event.docs) {
                        print("${doc.id} => ${doc.data()}");
                  }
            });
    }
}
```

Publish App on Play Store

https://developer.android.com/studio/publish

💡 Publishing is the general process that makes your Android app available to users.

- When you publish an Android app, you do the following:
  1. Prepare the app for release (build a release version of your app)
  2. Release the app to users (publicize, sell, and distribute the release version of your app)

Preparing App for Release

To prepare your app for release, you need to configure, build, and test a release version of your app.

The configuration tasks involve basic code cleanup and code modification tasks that help optimize your app.

1. **Gather materials and resources**
   - Cryptographic keys for signing your app
   - An app icon
   - EULA (end-user license agreement).
2. **Configure your app for release**
   - Make sure you choose an application ID that is suitable over the life of your app.
   - Turn of debugging and logging
   - Enable and configure app shrinking
   - Clean up your project directories
   - Review and update your manifest and Gradle build settings
   - Update URLs for servers and services
   - Implement licensing for Google Play
3. **Build your app for release**
   - Build your application into a release-ready APK file that is signed and optimized.
4. **Prepare external servers and resources**
5. **Test your app**

pubspec.yaml

- Every Flutter project includes a `pubspec.yaml` file, often referred to as the pubspec.
- A basic pubspec is generated when you create a new Flutter project.]
- It's located at the top of the project tree and contains metadata about the project that the Dart and Flutter tooling needs to know.
- The pubspec is written in YAML (Yet Another Markup Language) which is human readable, but be aware that whitespaces and tab spaces matter.
- The pubspec file specifies dependencies that the project requires, such as particular packages (and their versions), fonts, or image files.
- It also specifies other requirements, such as dependencies on developer packages, or particular constraints on the version of the Flutter SDK.
- Sample File:

```
name: <project name>
description: A new Flutter project.
publish_to: none
version: 1.2.3
environment:
  sdk: ^3.3.0

dependencies:
  flutter:       # Required for every Flutter project
    sdk: flutter # Required for every Flutter project
  flutter_localizations: # Required to enable localization
    sdk: flutter         # Required to enable localization
  cupertino_icons: ^1.0.6 # Only required if you use Cupertino (iOS style) icons

dev_dependencies:
  flutter_test:
    sdk: flutter # Required for a Flutter project that includes tests

flutter:
  assets:  # Lists assets, such as images
    - images/

  fonts:
    - family: Inter
      fonts:
        - asset: fonts/Inter.ttf
```

Light/Dark Mode

```
class _MyAppState extends State<MyApp> {
  ThemeMode _themeMode = ThemeMode.system; // Create a state for the mode

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Light/Dark Mode',
      theme: ThemeData(),
      darkTheme: ThemeData.dark(),
      themeMode: _themeMode, // Use state here
      home: MyHomePage(title: 'Flutter Light/Dark Mode'),
    );
  }

  void changeTheme(ThemeMode themeMode) {
    setState(() {
      _themeMode = themeMode;
    });
  }
}

class MyHomePage extends StatelessWidget {
  final String title;

  MyHomePage({required this.title});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(title)),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'Choose your theme:',
            ),
            Row(
              mainAxisAlignment: MainAxisAlignment.spaceEvenly,
              children: [
                ElevatedButton(
                    onPressed: () => MyApp.of(context).changeTheme(ThemeMode.light), // Change s
                    child: Text('Light')),
                ElevatedButton(
                    onPressed: () => MyApp.of(context).changeTheme(ThemeMode.dark), // Change st
                    child: Text('Dark')),
              ],
            ),
          ],
        ),
      ),
    );
  }
}
```

Intents (Android)

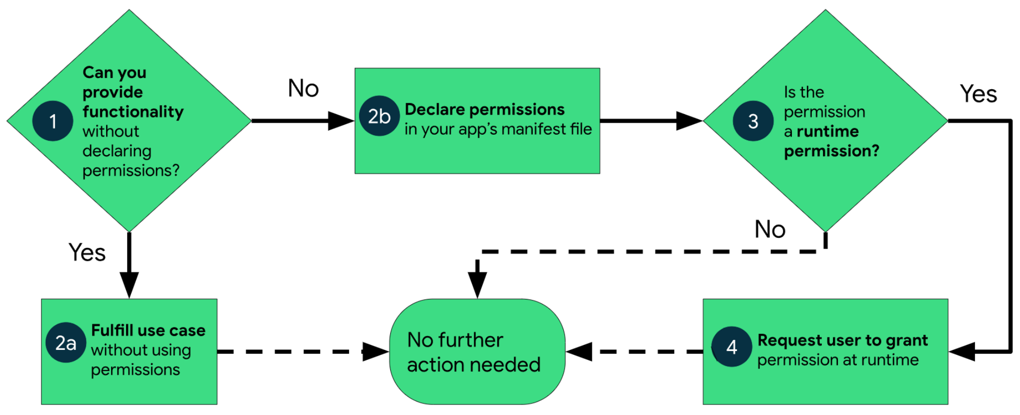
💡 An intent is a messaging object used to request an action from another app component.

- It is a description of an operation to be performed.
- Intent is a message that is passed between components such as activities, content providers, broadcast receivers, services etc.
- Used to
    - start an Activity

    - start a Service
    - deliver a Broadcast
- **Implicit Intent**
    - You don't specify the exact activity (or component) to run.
    - You include just enough information in the intent about the task you want to perform.
    - The Android system matches the information in your request intent with any activity available on the device that can perform that task.
- **Explicit Intent**
    - Explicit Intents have specified a component, which provides the exact class to be run.
    - We can also pass the information from one activity to another using explicit intent.
    - Used when you know the name of the target component within your app (or package name and class name for external app).

User Control (Permissions)

- App permissions help support user privacy by protecting access to:
    - Restricted Data (system state, user's contact information)
    - Restricted Actions (connecting to a paired device, recording audio)


1 Can you provide functionality without declaring permissions?
No
2b Declare permissions in your app's manifest file
3 Is the permission a runtime permission?
Yes
No
Yes
2a Fulfill use case without using permissions
No further action needed
4 Request user to grant permission at runtime

Types of Permissions

1. **Install-time permissions**
    - Give your app limited access to restricted data or let your app perform restricted actions.
    - The app store presents an install-time permission notice to the user when they view an app's details page.

    Two types:

    1. Normal permissions: Allow access to data and actions that extend beyond your app's sandbox but present very little risk.
    2. Signature permissions: The system grants a signature permission to an app only when the app is signed by the same certificate as the app or the OS that defines the permission.
2. **Runtime permissions**
    - Give your app additional access to restricted data or let your app perform restricted actions that more substantially affect the system and other apps.
    - When your app requests a runtime permission, the system presents a runtime permission prompt.
3. **Special permissions**
    - Special permissions correspond to particular app operations.
    - Only the platform and OEMs can define special permissions.
4. **Permission groups**
    - Permission groups consist of a set of logically related permissions.

    - For example, permissions to send and receive SMS messages might belong to the same group, as they both relate to the application's interaction with SMS.

Declaring App Permissions

- To declare a permission that your app might request, include the appropriate `<uses-permission>` element in your app's manifest file.

  ```
  <manifest ...>
      <uses-permission android:name="android.permission.CAMERA"/>
      <application ...>
          ...
      </application>
  </manifest>
  ```

- Some permissions, such as `CAMERA`, let your app access pieces of hardware that only some Android devices have.

  ```
  <uses-feature android:name="android.hardware.camera" android:required="false" />
  ```

- To check whether a device has a specific piece of hardware, use the `hasSystemFeature()` method

  ```
  if (getApplicationContext().getPackageManager().
          hasSystemFeature(PackageManager.FEATURE_CAMERA_FRONT)) {
          //code
  } else {
          //code
  }
  ```

Requesting App Permissions

- If your app needs to use resources or information outside of its own sandbox, you can declare a runtime permission and set up a permission request that provides this access.
- To check whether the user already granted your app a particular permission, pass that permission into the `ContextCompat.checkSelfPermission()` method.
    - Returns either `PERMISSION_GRANTED` or `PERMISSION_DENIED`
- Examples in manifest.xml

  ```
  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
  <uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
  ```

MVC Pattern

> *Mvc ka ek example leke explain karo*
> - Mr. Omkar Mohite

- Flutter follows a reactive programming model, where widgets update themselves when their state changes.
- To build complex applications, it’s important to follow a pattern that helps to separate concerns and maintain code structure.

💡 MVC pattern is a software design pattern that separates an application into three main components: Model, View, and Controller.

1. The Model represents the data and business logic of our application.

2. The View represents the user interface of our application.
3. The Controller acts as an intermediary between the Model and View, controlling the flow of data and updating the View based on changes in the Model

Steps

1. **Creating the Model**

   In Flutter, we can create a Model class that contains the data we want to represent.

   ```
   class User {
     String name;
     int age;

     User({required this.name, required this.age});
   }
   ```

2. **Creating the View**

   In Flutter, we can create a View using Widgets.

   ```
   class UserProfileView extends StatelessWidget {
     final User user;

     const UserProfileView({required this.user});

     @override
     Widget build(BuildContext context) {
       return Scaffold(
         appBar: AppBar(
           title: Text('User Profile'),
         ),
         body: Center(
           child: Column(
             mainAxisAlignment: MainAxisAlignment.center,
             children: [
               Text('Name: ${user.name}'),
               Text('Age: ${user.age}'),
             ],
           ),
         ),
       );
     }
   }
   ```

3. **Creating the Controller**

   In Flutter, we can create a Controller by creating a `StatefulWidget` that contains the Model and View.

   ```
   class UserProfileController extends StatefulWidget {
     final User user;

     const UserProfileController({required this.user});

     @override
     _UserProfileControllerState createState() => _UserProfileControllerState();
   }

   class _UserProfileControllerState extends State<UserProfileController> {
     late User _user;
   ```

```
  @override
  void initState() {
    super.initState();
    _user = widget.user;
  }

  @override
  Widget build(BuildContext context) {
    return UserProfileView(user: _user);
  }
}
```

4. **Using the Controller in the App**

   To use the Controller in our app, we can create a `MaterialApp` widget and use the `UserProfileController` as the home screen.

```
void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  final User user = User(name: 'John', age: 30);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'MVC in Flutter',
      home: UserProfileController(user: user),
    );
  }
}
```

Android Studio Structure (Android)

- Each project in Android Studio contains one or more modules with source code files and resource files.
- A module is a collection of source files and build settings that let you divide your project into discrete units of functionality.

Tree Structure

```
├── app/
|   ├── build
|   ├── libs
|   ├── src/
|   |   ├── androidTest
|   |   ├── cpp
|   |   ├── main/
|   |   |   ├── AndroidManifest.xml
|   |   |   ├── java
|   |   |   ├── kotlin
|   |   |   ├── res
|   |   |   └── assets
|   |   └── test
```

```
│    └── build.gradle
└── build.gradle
```

- `build` : Contains build outputs
- `libs` : Contains private libraries
- `src` : Contains all code and resource files for the module
- `androidTest` : Contains code for instrumentation tests that run on an Android device.
- `cpp` : Contains native C or C++ code using the Java Native Interface (JNI).
- `main` : Contains the "main" source set files; the Android code and resources shared by all build variants.
- `AndroidManifest.xml` : Describes the nature of the application and each of its components.
- `java` : Contains Kotlin or Java code sources.
- `kotlin` : Contains only Kotlin code sources.
- `res` : Contains application resources, such as drawable files and UI string files.
- `assets` : Contains files to be compiled into an APK file as-is.
- `test` : Contains code for local tests that run on your host JVM.
- `build.gradle` (module): This defines the module-specific build configurations.
- `build.gradle` (app): This defines your build configuration that applies to all modules.

Handling Images

- Displaying images is the fundamental concept of most of the mobile apps.
- Flutter has an `Image` widget that allows displaying different types of images in the mobile application.
- **Steps**
  1. First, we need to create a new folder inside the root of the Flutter project and name it `assets`
  2. Next, add the image file manually inside this folder.
  3. Update the **`pubspec.yaml`** file.

     ```
     assets:
         - assets/

     ## or specify the filename
     assets:
         - assets/image.png
     ```

  4. Use the widget

     ```
      Image.asset('assets/image.png')

      Image.network('url') // no steps needed for network image

      // specify height width
      Image.asset(
              'assets/image.png',
                 height: 69,
                 width: 420,
      )
     ```