Econix Documentation

Flutter

Setting Up

“Flutter is Google’s UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase.” as stated at Flutter official site.
We can configure flutter at different IDEs like Android Studio, VS Code, Intellji, etc. Here we will make a setup of Flutter SDK in Android Studio and provide the necessary information related setup and configuration.

Let’s start our setup for the Android Studio:

Android Studio

❖ **Overview:** Android Studio is the official Integrated Development Environment (IDE) for Android app development, based on IntelliJ IDEA . On top of IntelliJ's powerful code editor and developer tools, Android Studio offers even more features that enhance your productivity when building Android apps. Learn more

❖ **Installation:** Installation of Android Studio is fun, we need to follow the proper installation guidelines which are provided here and also referred to the Google official documentation.

- ➢ Lets know some **requirements** of installation at first
  - ■ Windows:
    - Microsoft® Windows® 7/8/10 (64-bit)
    - 4 GB RAM minimum, 8 GB RAM recommended
    - 2 GB of available disk space minimum,
      4 GB Recommended (500 MB for IDE + 1.5 GB for Android SDK and emulator system image)
    - 1280 x 800 minimum screen resolution
  - ■ Mac:
    - Mac® OS X® 10.10 (Yosemite) or higher, up to 10.14 (macOS Mojave)
    - 4 GB RAM minimum, 8 GB RAM recommended
    - 2 GB of available disk space minimum,
      4 GB Recommended (500 MB for IDE + 1.5 GB for Android SDK and emulator system image)
    - 1280 x 800 minimum screen resolution

After the requirements meet we are ready to start installation at our own platform os.

- ➢ Download Android Studio and SDK tools
    - ■ After completion of downloads open from the downloaded location folder and launch the **.exe file**.
    - ■ If you download the zip file, unzip the file and launch it
    - ■ After double clicking on the Android Studio icon it will launch and download the necessary SDK which takes some time.
    - ■ Follow this link for better understand

- ❖ **Configure the IDE:** Android Studio provides wizards and templates that verify your system requirements, such as the Java Development Kit (JDK) and available RAM, and configure default settings, such as an optimized default Android Virtual Device (AVD) emulation and updated system images.
    - ➢ Find your configuration files from here
    - ➢ Configure Android Studio bookmark_border or you can download jdk manually from here as your platform os based.

Flutter SDK Setup

1. **Requirements:**

   To install and run flutter these minimum requirements must have been met in your development environment.

➔ **For windows:**

- Operating Systems: Windows 7 SP1 or later (64-bit), x86-64 based
- Disk Space: 1.64 GB (does not include disk space for IDE/tools).
- Tools: Flutter depends on these tools being available in your environment.
- Windows PowerShell 5.0 or newer (this is pre-installed with Windows 10)
- Git for Windows 2.x, with the Use Git from the Windows Command Prompt option.
  If Git for Windows is already installed, make sure you can run git commands from the command prompt or PowerShell.

➔ **For macOs:**

- Operating Systems: macOS (64-bit)
- Disk Space: 2.8 GB (does not include disk space for IDE/tools).
- Tools: Flutter uses git for installation and upgrade. We recommend installing Xcode, which includes git, but you can also install git separately.

2. Download Sdk
   - Get The Flutter Sdk from here.

3. Update your path

   If you wish to run Flutter commands in the regular Windows console, take these steps to add Flutter to the PATH environment variable:

   - From the Start search bar, enter ‘env’ and select Edit environment variables for your account.
   - Under User variables check if there is an entry called Path:
     - If the entry exists, append the full path to flutter\bin using ; as a separator from existing values.
     - If the entry doesn’t exist, create a new user variable named Path with the full path to **flutter\bin** as its value.

You have to close and reopen any existing console windows for these changes to take effect.

4. Set up your Android device

To prepare to run and test your Flutter app on an Android device, you need an Android device running Android 4.1 (API level 16) or higher.

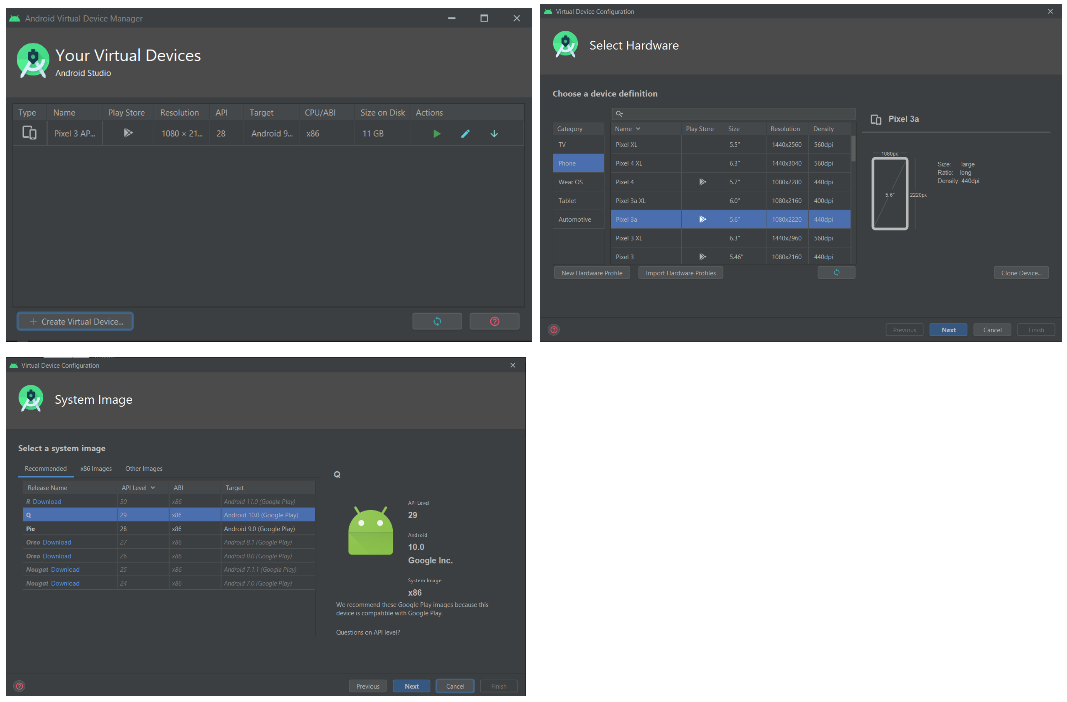
1. Enable Developer options and USB debugging on your device. Detailed instructions are available in the Android documentation.
2. Windows-only: Install the Google USB Driver.
3. Using a USB cable, plug your phone into your computer. If prompted on your device, authorize your computer to access your device.
4. In the terminal, run the flutter devices command to verify that Flutter recognizes your connected Android device. By default, Flutter uses the version of the Android SDK where your adb tool is based. If you want Flutter to use a different installation of the Android SDK, you must set the ANDROID_SDK_ROOT environment variable to that installation directory.

5. Set up the Android emulator

To prepare to run and test your Flutter app on the Android emulator, follow these steps:

1. `Enable VM acceleration on your machine.
2. Launch Android Studio, click the AVD Manager icon, and select Create Virtual Device…
    - In older versions of Android Studio, you should instead launch Android Studio > Tools > Android > AVD Manager and select Create Virtual Device…. (The Android submenu is only present when inside an Android project.)
    - If you do not have a project open, you can choose Configure > AVD Manager and select Create Virtual Device…
3. Choose a device definition and select Next.
4. Select one or more system images for the Android versions you want to emulate, and select Next. An *x86* or *x86_64* image is recommended.
5. Under Emulated Performance, select Hardware - GLES 2.0 to enable hardware acceleration.
6. Verify the AVD configuration is correct, and select Finish.
7. For details on the above steps, see Managing AVDs.

8. For details on the above steps, see Managing AVDs.
9. In Android Virtual Device Manager, click Run in the toolbar. The emulator starts up and displays the default canvas for your selected OS version and device.



Type	Name	Play Store	Resolution	API	Target	CPU/ABI	Size on Disk
	Pixel 3 AP...		1080 × 21...	28	Android 9...	x86	11 GB

Category	Name	Size	Resolution	Density
TV	Pixel XL	5.5"	1440x2560	560dpi
Phone	Pixel 4 XL	6.3"	1440x3040	560dpi
Wear OS	Pixel 4	5.7"	1080x2280	440dpi
Tablet	Pixel 3a XL	6.0"	1080x2160	400dpi
Automotive	Pixel 3a	5.6"	1080x2220	440dpi
	Pixel 3 XL	6.3"	1440x2960	560dpi
	Pixel 3	5.46"	1080x2160	440dpi

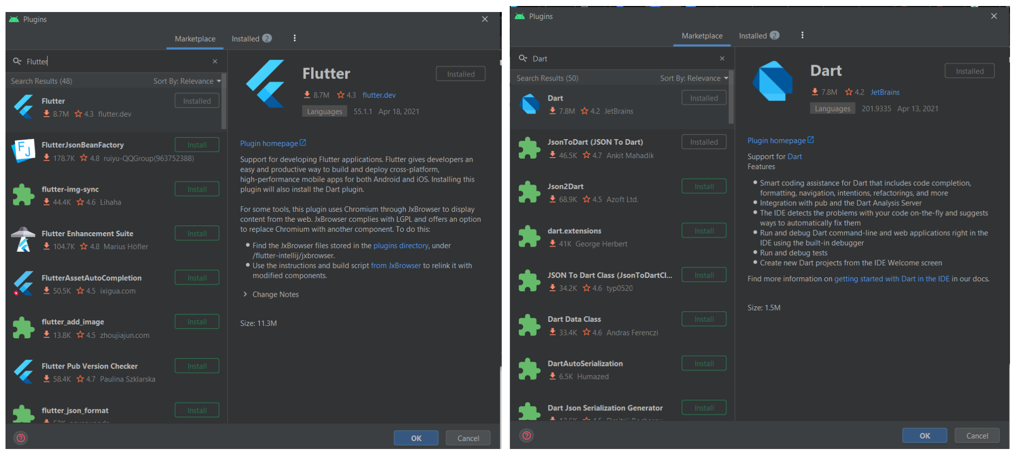
Release Name	API Level	ABI	Target
R Download	30	x86	Android 11.0 (Google Play)
Q	29	x86	Android 10.0 (Google Play)
Pie	28	x86	Android 9.0 (Google Play)
Oreo Download	27	x86	Android 8.1 (Google Play)
Oreo Download	26	x86	Android 8.0 (Google Play)
Nougat Download	25	x86	Android 7.1.1 (Google Play)
Nougat Download	24	x86	Android 7.0 (Google Play)

API Level 29, Android 10.0 Google Inc., System Image x86

Fig: Create AVD

6. Configure plugin

Install Flutter and Dart plugin in your Android Studio from **File > Settings > Plugin** and search in the search box respectively **Flutter** and **Dart** to configure your ide with the flutter and sdk.



Flutter — 8.7M ☆ 4.3 flutter.dev — Languages 55.1.1 Apr 18, 2021

Support for developing Flutter applications. Flutter gives developers an easy and productive way to build and deploy cross-platform, high-performance mobile apps for both Android and iOS. Installing this plugin will also install the Dart plugin.

For some tools, this plugin uses Chromium through JxBrowser to display content from the web. JxBrowser complies with LGPL and offers an option to replace Chromium with another component. To do this:

- Find the JxBrowser files stored in the plugins directory, under /flutter-intellij/jxbrowser.
- Use the instructions and build script from JxBrowser to relink it with modified components.

Dart — 7.8M ☆ 4.2 JetBrains — Languages 201.9335 Apr 13, 2021

Support for Dart
Features

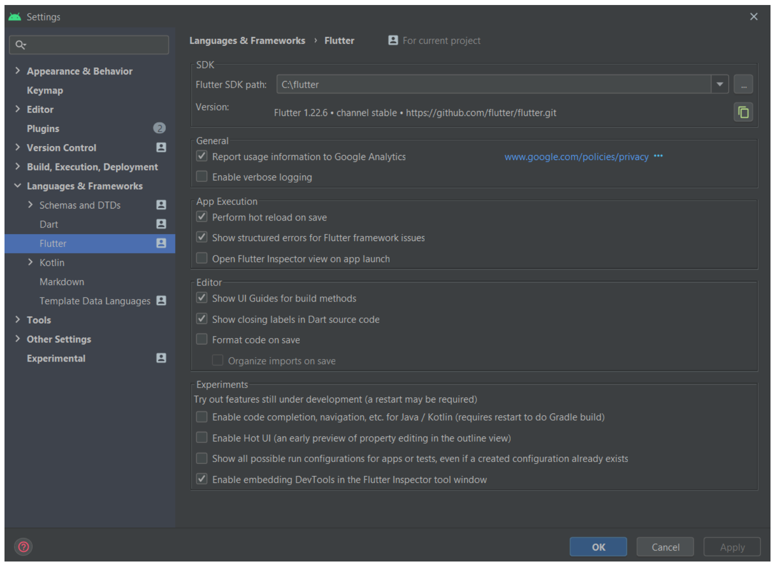
- Smart coding assistance for Dart that includes code completion, formatting, navigation, intentions, refactorings, and more
- Integration with pub and the Dart Analysis Server
- The IDE detects the problems with your code on-the-fly and suggests ways to automatically fix them
- Run and debug Dart command-line and web applications right in the IDE using the built-in debugger
- Run and debug tests
- Create new Dart projects from the IDE Welcome screen

Find more information on getting started with Dart in the IDE in our docs.

Fig: Plugin install

7. Set Fluter Sdk Path

Need to set Flutter Sdk path location at **File > Settings > Language & Frameworks > Flutter > “your flutter sdk file location”**



Languages & Frameworks › Flutter   For current project

SDK
Flutter SDK path: C:\flutter
Version: Flutter 1.22.6 • channel stable • https://github.com/flutter/flutter.git

General
- [✓] Report usage information to Google Analytics   www.google.com/policies/privacy
- [ ] Enable verbose logging

App Execution
- [✓] Perform hot reload on save
- [✓] Show structured errors for Flutter framework issues
- [ ] Open Flutter Inspector view on app launch

Editor
- [✓] Show UI Guides for build methods
- [✓] Show closing labels in Dart source code
- [ ] Format code on save
  - [ ] Organize imports on save

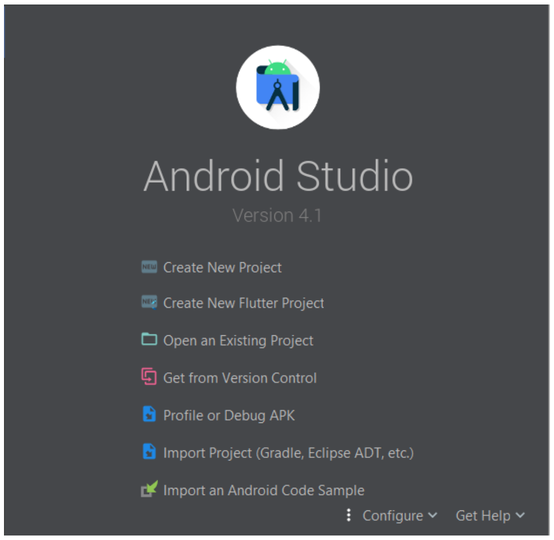
Experiments
Try out features still under development (a restart may be required)
- [ ] Enable code completion, navigation, etc. for Java / Kotlin (requires restart to do Gradle build)
- [ ] Enable Hot UI (an early preview of property editing in the outline view)
- [ ] Show all possible run configurations for apps or tests, even if a created configuration already exists
- [✓] Enable embedding DevTools in the Flutter Inspector tool window

Running

Now we are ready to run our Econix flutter project on Android Studio. There are a few steps given below to open an existing flutter project from Android Studio.

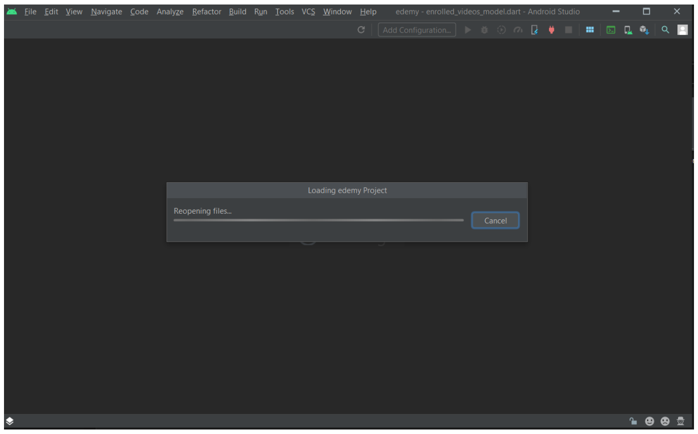
After opening the Android Studio there will appear a window just like the picture.

1. Click on Open an existing project

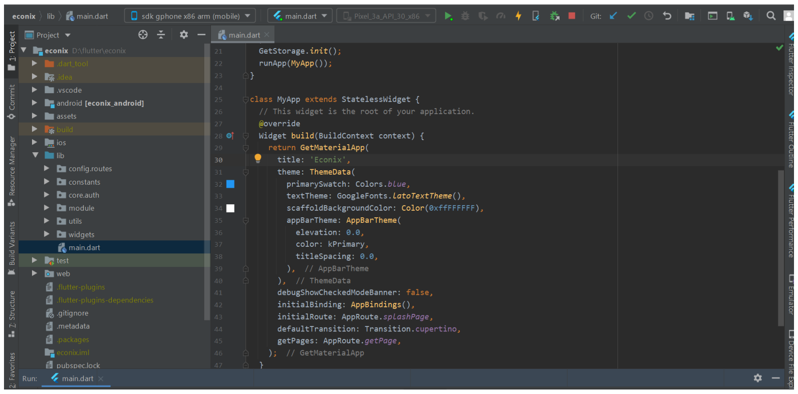


2. Select the econix file from the folder of econix at your device storage.

   It will take few time to analyze the project and build the necessary configuration


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
edemy - enrolled_videos_model.dart - Android Studio
Add Configuration...
Loading edemy Project
Reopening files...
Cancel

Root widget of the econix can be found in the main.dart file.

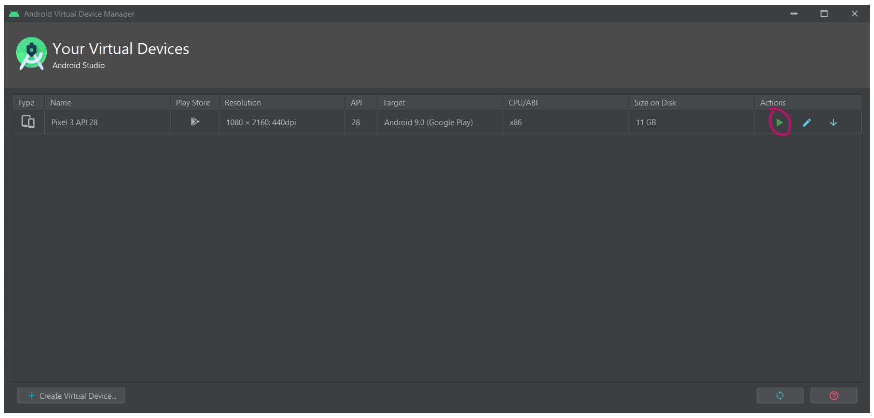


```dart
  GetStorage.init();
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return GetMaterialApp(
      title: 'Econix',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        textTheme: GoogleFonts.latoTextTheme(),
        scaffoldBackgroundColor: Color(0xffFFFFFF),
        appBarTheme: AppBarTheme(
          elevation: 0.0,
          color: kPrimary,
          titleSpacing: 0.0,
        ),  // AppBarTheme
      ),  // ThemeData
      debugShowCheckedModeBanner: false,
      initialBinding: AppBindings(),
      initialRoute: AppRoute.splashPage,
      defaultTransition: Transition.cupertino,
      getPages: AppRoute.getPage,
    );  // GetMaterialApp
  }
```

3. Now write **Flutter clean** and **Flutter pub get** at the terminal respectively to get packages.
4. Now we need to click on the AVD icon located at the top toolbar of Android Studio and start launching the emulator or we can run it on our real physical device.

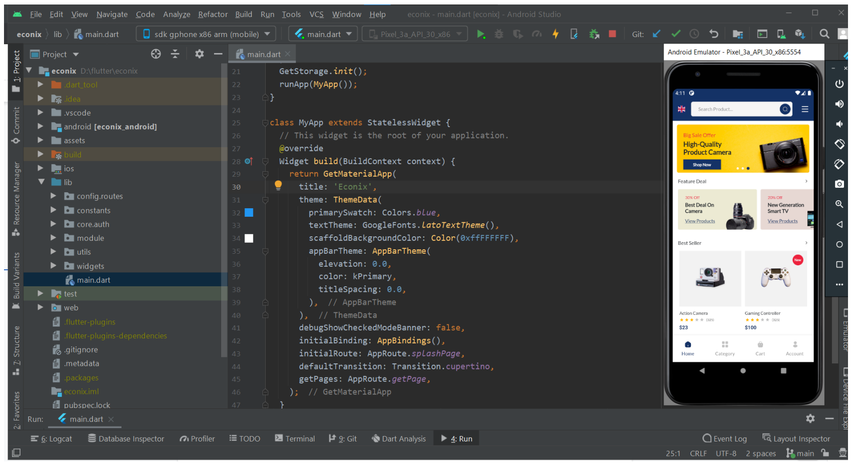
Click on AVD icon



Type	Name	Play Store	Resolution	API	Target	CPU/ABI	Size on Disk	Actions
	Pixel 3 API 28		1080 × 2160: 440dpi	28	Android 9.0 (Google Play)	x86	11 GB	

Fig: Click on arrow button to launch Emulator

5. After launching the emulator we can click at **Run button** or write **Flutter run** at terminal to run our code in debug mode



```dart
  GetStorage.init();
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return GetMaterialApp(
      title: 'Econix',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        textTheme: GoogleFonts.latoTextTheme(),
        scaffoldBackgroundColor: Color(0xffFFFFFF),
        appBarTheme: AppBarTheme(
          elevation: 0.0,
          color: kPrimary,
          titleSpacing: 0.0,
        ),  // AppBarTheme
      ),  // ThemeData
      debugShowCheckedModeBanner: false,
      initialBinding: AppBindings(),
      initialRoute: AppRoute.splashPage,
      defaultTransition: Transition.cupertino,
      getPages: AppRoute.getPage,
    );  // GetMaterialApp
  }
```

Fig: main.dart code and fresh launch on emulator

Rename project

To update our project name we have to follow the below points:

1. Open the android folder under the root project folder.
2. Go to app > src > main
3. Open AndroidManifest.xml
4. Within the <application tag `android:label="Econix"`
5. Now rename the project as you want.

Renaming the package

Two steps are available to renaming the package:

1. To update our project package name, we can do it by manually to go where package name contains and change it
2. On the other hand we can use a plugin called "`change_app_package_name:`

With it's latest version. And run this below command line with the package name as we want.

```
flutter pub run change_app_package_name:main com.envytheme.app.econix
```

Changing the logo

To change our project logo we have to follow the below points:

1. Pick the required app icon at android > app > src > main > res within the mipmap folder
2. Open the android folder.
3. Go to app > src > main
4. Open AndroidManifest.xml
5. Within the <application tag `android:icon="@mipmap/app_icon"`
6. Now set the icon as you want.

Project Edit

Folder Structure:

Let's take a look at our project folder structure so that we can easily route to one another.

Lib is a root folder of any flutter project and we need to create our own folder/files as per our requirements on it. By default we can find main.dart on it and some boilerplate code which need to be customized as per our requirements.

In our project we create several folders under the lib folder.

1. **lib: main.dart** is the root file of the flutter application where the root widget of the application navigates from this file.Our initial route is AppRoute.splash.
2. **lib > core** where we keep auth login and register screen.
3. **lib > config** We create a separate AppRoute singleton class where we define all the classes name in instance and assign them to GetPage list. In our project we used Getx State management approach, It also offers us Navigation Management.
4. **lib > constants** In constants folder we keep application constants contents.
    a. api_path.dart

        i. All api urls which we've used in our project we keep them within this file.
    b. app_constants.dart
        i. All static and const text, color, decoration keep them in this file
    c. asset_path.dart
        i. All assets path are kept here for easy use
    d. size.dart
        i. In the size file we estimated user device size with based size and made our application responsible for all kinds of device height and width.

5. **lib > module**

   Under the module folder we create a **controller, model, views** folder.

   - ➢ **controller:** within this controller folder we manage our application state. We create controller files named,
     - app_controller.dart,
     - auth_controller.dart,
     - base_controller.dart,
     - cart_controller.dart.
     - data_controller.dart
   - ➢ **model:** we create data model classes as per our different json objects and keep them within this model folder.
     - banner_model.dart
     - category_model.dart
     - product_model.dart
     - my_order_model.dart
     - checkout_model.dart
     - user_model.dart
   - ➢ **views:** In our application UI has been placed within this views folder and we manage them separate by keeping in separate folder given below,
     - about_&_conditions
     - cart
     - category
     - home
     - products
     - product_details
     - order
     - wishlist
     - profile
     - splash

6. **lib > utils > services**
   a. **Api:** In the api folder we call our all endpoints from different files.
      i. auth_service.dart
      ii. checkout.dart
      iii. product_service.dart

b. **local-storage**: To saving the data locally we manage this folder within service folder
    i. local_storage.dart
c. **widgets:** Separate different widgets from the ui so that code looks clean and increases readability. Refactored widgets we keep them within this widget folder.

Folder structure is very clean and easy to understand. Now we will focus on the **Api section**. As we said earlier that we keep different files and folders for different task execution, now we have to look at those api related files to call the api.

Api Integration:

We integrated all api related tasks within our **service** folder, and each service file is singleton class. As we said before we used the getX approach for data management in our project,we linked our api endpoint response with the getX controller class as separate business logic.
We try to demonstrate by diagram how we call the data from the internet and store them.

Diagram:

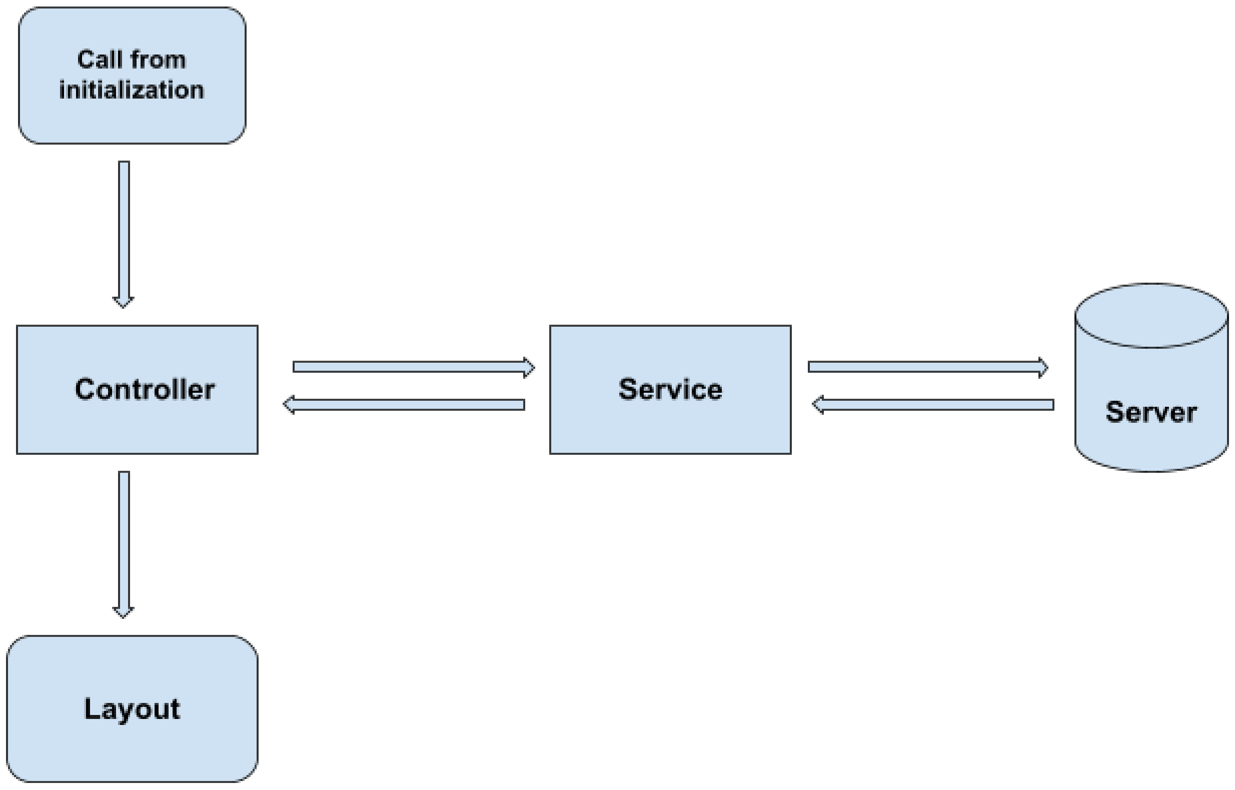


Fig: Flow chart of data fetching

Api endpoints:

- **baseUrl** = https://econix-react.envytheme.com
- **Signup**: baseUrl/user/register
- **Login:** baseUrl/user/login
- **All products:** baseUrl/products
- **Single products details:** baseUrl/products/fetch-product
- **Checkout:** baseUrl/order/add-order-info

- **User-account:** baseUrl/user

- **Order-history:** baseUrl/order/myorders

1. **Authentication:** Login and Signup layout can be found in the core folder.
   a. **Signup:** baseUrl/user/register this blue colored text is our signup api endpoint
   And we pass the name,username, phone, email, and password value as parameters in the body.
   To manage this signup we call userRegister method from auth_controller.dart and Then from here call the registerFromApi method with name, username, phone, email, password parameter at ApiService class to call the http POST method request for signup.
   b. **Login:** Making http POST method request at baseUrl/user/login this api endpoint by passing the email and password parameter at the body, from auth_controller.dart.
2. **Products:**
   a. **All products:** Using our products endpoint we fetch all the products data from the server and place them in our layout. We used product data mainly in two separate screens named Home and Products. Which are found under the screens folder. We initialize the product_controller.dart to the bindings and it load and from this method call getProductController for http GET request to baseUrl/products this endpoint. And set json data to a data model named **ProductModel**.

3. **Single product details**
   a. Single product details are extracted by calling baseUrl/products/fetch-product this endpoint by passing the **product id** at params (i.e baseUrl/products/fetch-product/603ecf1aa5970c30304e827a). When a user tap on a product to see its details it will called the getProductDetails method which need a required parameter of product id from product_controller.dart and from here called the detailsService method by passing this product id to fetch the details by calling http GET request. And set json data to a data model named **Products**.

4. **Account:**
   a. User-profile will be displayed from the baseUrl/user endpoint. User-details api fetching are contained within auth_controller and auth_service.

5. **Order-History:**
    a. Using our order-history (baseUrl/order/myorders) endpoint we can view all our orders. This integration has been managed within our service and controller class respectively.

6. **Checkout:**
    a. **Make payment:** we've used the stripe_payment | Flutter Package to make the payment successful. We can find the installation and use guidelines from this package which we used in our project. We create a class named **PaymentService** and implement the transaction service here. Secret key and publishableKey is important which we can find from here by creating an account.
        i. We need to pass the total amount and currency type here.
    b. **Order-placed:** After making the successful payment we need to call the checkout api endpoint from the Checkout service class within our service > api folder. Endpoint: baseUrl/order/add-order-info