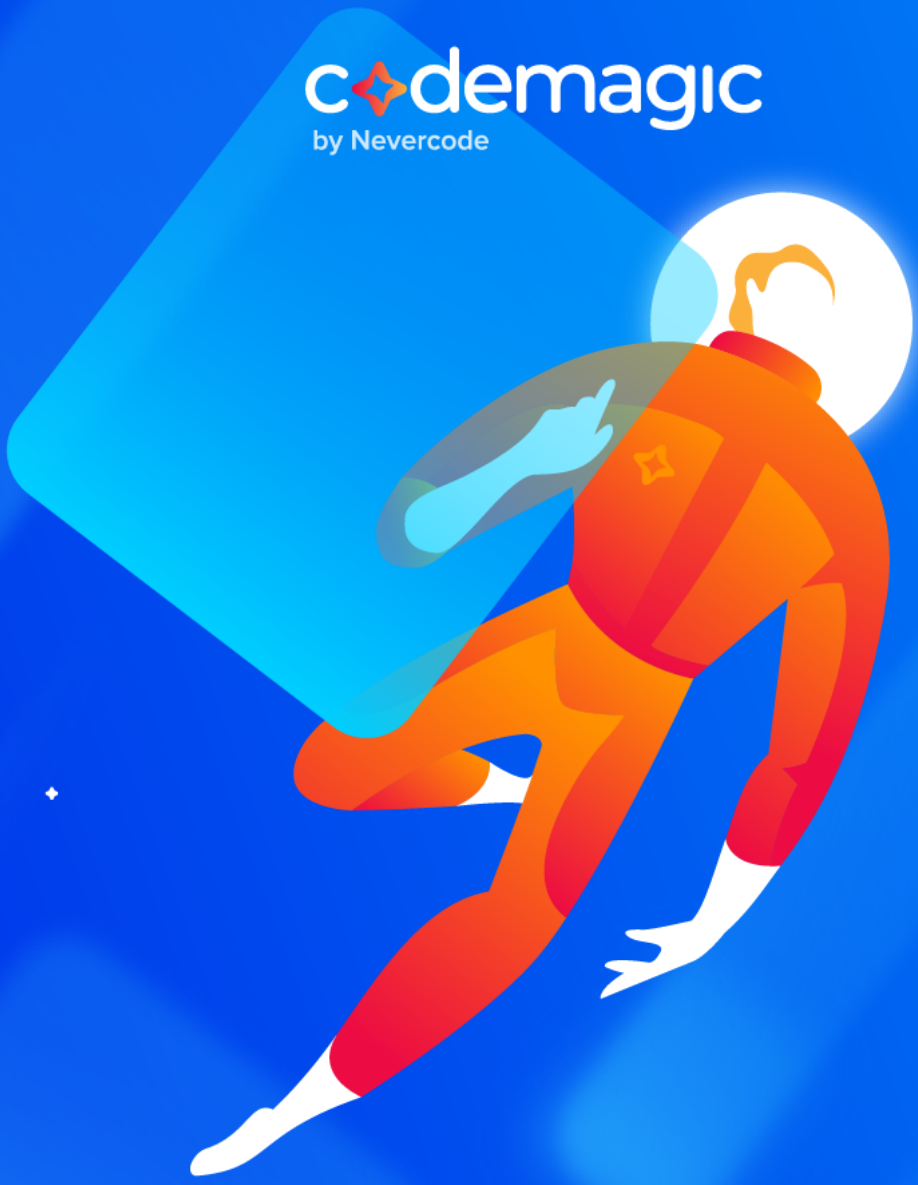
FLUTTER LIBRARIES WE LOVE

codemagic
by Nevercode



Discover 60+ Flutter libraries from 11 different categories

FLUTTER LIBRARIES WE LOVE

60+ must-have Flutter libraries to speed up your work

by **Souvik Biswas** & **Codemagic**

Designed by Kätrin Sibul

Table of contents

Editorial note

It all started with a tweet, like so many things nowadays. Marie Jaksman, the CMO of Codemagic, tweeted a question in March to find out which Flutter Libraries are absolutely necessary for developers.

We got so many great ideas from the answers that it was impossible to squeeze all those libraries into one article. Instead, we decided to create our first-ever ebook.

The idea was to cover a wide range of categories and to showcase different libraries – some that are less known gems but also some that are already quite popular and have the Flutter Favorite label on them.

Unfortunately, we couldn't cover all the libraries from the tweets we got but we tried to find a great balance between famous libraries and hidden treasures.

This book is not a beginners book, although feel free to take a look if you are one. You won't find all "must have" libraries here but you might find an interesting selection of something unique. We tried to pick libraries that might not be popular yet but have both great potential and our wholehearted support. In addition to that, there are also some libraries that we just had to mention because the Flutter community simply loves them.

Introduction

When talking to developers, one question always comes up – what tools should we use? Flutter is relatively new but the selection of Flutter libraries is growing fast.

We focused on 11 different Flutter library categories. After selecting the categories, we made a list of libraries under each category and chose the libraries we wanted to highlight the most.

You will find 11 different Flutter library categories:

1. State management
2. Networking
3. Text and fonts
4. UX/UI
5. Location and connectivity
6. Images and videos
7. Data persistence and file system
8. Animations and transitions
9. Utility
10. Code generator and build tools
11. Testing

Each category has a list of Flutter libraries as well as a highlighted library that we dig deeper into.

In addition to an overall description, all highlighted libraries consist of pros and cons, developer's perspective and real-life code examples.

STATE MANAGEMENT

State management is a crucial aspect when working on a large scale production app. That said, every state management solution is unique and is fit for a certain architecture of the app.
Let's take a look at one of the state management libraries that does a great job of separating the business logic from the UI – Flutter BLoC.

Flutter BLoC

Helps implement BLoC pattern
by Felix Angelov

ANDROID iOS WEB

Flutter BLoC is a predictable state management library that helps to implement the **Business Logic Component (BLoC)** design pattern. It uses reactive programming to handle the flow of data within an app.

> "I made the BLoC library to help developers manage their application state in a predictable, testable, and scalable way. The best thing about it is the awesome community behind it. I will continue developing it and there will be exciting updates in the near future."
>
> **Felix Angelov, the creator of Flutter BLoC**

Why BLoC?

BLoC helps to encapsulate the business logic from the UI. **Events** are fed in as the input to the logic component and **states** are generated as the output. It relies heavily on Streams and is often used in conjunction with the Provider for exposing the BLoC to the UI. Testing BLoC is really easy using the library bloc_test.

Developer's perspective

There are a lot of options while choosing a state management library for Flutter, and every app's structure is unique. So, there is no state management library best for every use case. BLoC is really nice if you are working on a large-scale production app, but due to the amount of boilerplate code, it might not be suitable for smaller apps.

Pros and cons

+ Easy to separate UI and business logic
+ Good tooling support for VS Code and IntelliJ
+ Easy to test
+ Nice documentation

- Creates a lot of boilerplate code
- Not suitable for simple apps

Example

An app for fetching applications using Codemagic API is implemented using the BLoC pattern.

The FetchApplication event will be added to the BLoC to fetch the applications from the server. It will mostly be used right after the initial state when there are no applications fetched yet.

```
// BLoC Event

abstract class ApplicationEvent extends Equatable {
  const ApplicationEvent();
}

class FetchApplication extends ApplicationEvent {
  const FetchApplication();

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

ApplicationState will define all possible states while fetching the applications from the server.

- ApplicationEmpty: initial state when there is no data present
- ApplicationLoading: while fetching applications
- ApplicationLoaded: successfully fetched an application
- ApplicationError: unable to fetch the application

```
// BLoC State

abstract class ApplicationState extends Equatable {
  ApplicationState();

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

class ApplicationEmpty extends ApplicationState {}

class ApplicationLoading extends ApplicationState {}
```

```
class ApplicationLoaded extends ApplicationState {
  final Application application;

  ApplicationLoaded({@required this.application}) :
assert(application != null);

  @override
  List<Object> get props => [application];
}
class ApplicationError extends ApplicationState {}
```

The ApplicationBloc is responsible for receiving the ApplicationEvents and converting them into ApplicationStates.

```
// BLoC

class ApplicationBloc extends Bloc<ApplicationEvent,
ApplicationState> {
  final ApplicationRepository applicationRepository;

  ApplicationBloc({@required this.applicationRepository})
      : assert(applicationRepository != null),
        super(ApplicationEmpty());

  @override
  Stream<ApplicationState> mapEventToState(ApplicationEvent
event) async* {
    if (event is FetchApplication) {
      yield ApplicationLoading();

      try {
        final Application application =
            await applicationRepository.fetchApplication();
```

```
        yield ApplicationLoaded(application: application);
      } catch (_) {
        ApplicationError();
      }
    }
  }
}
```

BlocProvider is used to create and manage an instance of ApplicationBloc.

```
class ApplicationPage extends StatelessWidget {
  final ApplicationRepository repository;

  ApplicationPage({@required this.repository}) :
assert(repository != null);

  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (context) =>
ApplicationBloc(applicationRepository: repository),
      child: ApplicationView(),
    );
  }
}
```

BlocBuilder is used to build the UI based upon the state returned by the ApplicationBloc. When the state is ApplicationEmpty, the FetchApplication event is added to the ApplicationBloc.

```
class ApplicationView extends StatelessWidget {
  final TextStyle _style = TextStyle(fontSize: 18);
  @override
  Widget build(BuildContext context) {
    return BlocBuilder<ApplicationBloc, ApplicationState>(
```

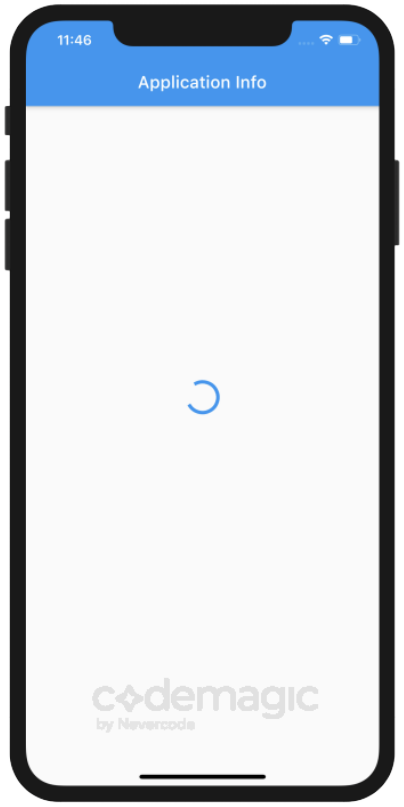
```
      builder: (context, state) {
        if (state is ApplicationEmpty) {
          BlocProvider.of<ApplicationBloc>(context).
add(FetchApplication());
        }
        if (state is ApplicationError) {
          return Center(
            child: Text('Failed to load app info'),
          );
        }
        if (state is ApplicationLoaded) {
          return Center(
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              crossAxisAlignment: CrossAxisAlignment.start,
              children: <Widget>[
                Text('App ID: ${state.application.id}', style:
_style),
                Text('App Name: ${state.application.appName}',
style: _style),
                Text('Last Build Id: ${state.application.
lastBuildId}',
                    style: _style),
              ],
            ),
          );
        }
        return Center(
          child: CircularProgressIndicator(),
        );
      },
    );
  }
}
```

Reference links

Package

Documentation

Sample app



List of state management libraries

Here are some libraries for managing the state of your Flutter app:

- **flutter_mobx**: Flutter integration for **MobX**. It provides a set of Observer widgets that automatically rebuild when the tracked observables change.
- **flutter_redux**: set of utilities that allow you to easily consume a **Redux** Store to build Flutter widgets.
- **provider**: Flutter plugin that acts as a wrapper around the **InheritedWidget** to make them easier to use and more reusable.
- **flutter_modular**: Flutter plugin that helps to deal with problems like dependency injection and route management.
- **get_it**: simple direct Service Locator that allows decoupling the interface from a concrete implementation and accessing it from anywhere in the app.

NETWORKING

Fetching data from the internet is necessary for most apps. Stated below are some Flutter libraries that will help you to make network requests and handle the responses gracefully in your app.

Dio

A powerful HTTP client for Dart

by Flutter China

ANDROID iOS WEB

Dio is a Flutter networking library for handling HTTP requests with ease. It supports interceptors, global configuration, FormData, request cancellation, file downloading, timeout, etc.

> "Dio is a powerful tool for API calls and much more. With customer interceptors with Dio, I have super powers!"
>
> Temi Ajiboye, **mobile app developer**

Why Dio?

Dio is a quite helpful library for anyone working with APIs in their application. It helps with making API calls and provides good control over them. With Dio, you can easily manage uploading and downloading of multiple files. Using Dio Interceptor allows you to intercept and lock/unlock requests for performing some operations in between an API request.

Developer's perspective

There is a similar library for handling HTTP requests in Dart, known as **http**. But it is quite verbose and does not allow much control over the HTTP calls. **Interceptor** is a vital part of Dio that is useful in different scenarios, for example if you want to automatically retry a request when the internet connection is restored. You can even track the download progress of large files easily using Dio.

Pros

+ Make API calls over HTTP
+ Track download/upload progress
+ Ability to intercept requests

Example

```
// Initialize Dio
BaseOptions options = new BaseOptions(
    baseUrl: 'https://api.codemagic.io',
    connectTimeout: 5000,
    receiveTimeout: 3000,
    headers: {
      "Content-Type": "application/json",
      "x-auth-token": _token,
    });
```

```
_dio = new Dio(options);

// POST Data
Response response = await _dio.post(
  “/builds”,
  data: {
    “appId”: _appId,
    “workflowId”: _workflowId,
    “branch”: _branch,
  },
);
if (response.statusCode == 200) {
  print(response.data);
}

// GET Data
Response response = await _dio.get(
  “/builds/$_buildId”,
);
if (response.statusCode == 200) {
  print(response.data);
}
```

Reference links

Package

Documentation

Sample app

List of networking libraries

- **http**: Flutter plugin supporting composable, multi-platform, Future-based API for HTTP requests.
- **url_launcher**: Flutter plugin for launching a URL in the mobile platform. It has support for web, phone, SMS, and email schemes.
- **firebase_auth**: Flutter plugin for Firebase Auth, enabling authentication using passwords, phone numbers and identity providers like Google, Facebook and Twitter.
- **firebase_messaging**: Flutter plugin for Firebase Cloud Messaging, a cross-platform messaging solution that lets you reliably deliver messages.
- **google_sign_in**: Flutter plugin for Google Sign-In, a secure authentication system for signing in with a Google account.

TEXT AND FONTS

Typography is a vital aspect of every popular mobile application. It makes your app more attractive and consistent throughout different screen sizes. Let's take a look at some of the Flutter libraries for improving the text and font of your app.

A Flutter library that adds astonishing animations to any app text is worthy of being highlighted.

Animated Text Kit

Create cool and beautiful text animations
by Ayush Agarwal

ANDROID iOS WEB

Animated Text Kit provides you with easy implementation of beautiful and nice text animations to your Flutter app. It contains 7 types of text animations, including Rotate, Fade, Typer, Typewriter, Scale, Colorize and TextLiquidFill.

> "I started working on this package after Jeremiah and I released the Flutter Spinkit package. I was working on some Flutter and needed an animated text for the UI, but couldn't find any suitable packages or even blogs about this. So I decided to create a Flutter package myself. I am very glad that it became so popular among other Flutter developers."
>
> Ayush Agarwal, **creator of Animated Text Kit**

Why Animated Text Kit?

With **Animated Text Kit**, you can prevent the boilerplate code required to achieve these text animations. There are several customization options, so building your unique design won't be a challenge with this package. You can apply the animations to either a single String or a List of Strings.

Developer's perspective

Animated Text Kit is an impressive text animation package that lets you add subtle but eye-catching texts to your app. This package comes with several types of text animations, but the most distinguishable among these is the TextLiquidFill, which adds a liquid filling-like text animation.

Pros

+ Easy implementation
+ Large number of customization options
+ Support for any TextStyle

Example

```
class LiquidTextView extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return SafeArea(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.start,
        children: <Widget>[
          SizedBox(height: 50),
          Text(
            'TextLiquidFill',
            style: TextStyle(fontSize: 30.0, fontWeight:
FontWeight.bold),
```

```
            ),
            SizedBox(height: 200),
            TextLiquidFill(
              text: 'CODE',
              waveColor: Colors.blueAccent[700],
              boxBackgroundColor: Colors.orange[600],
              textStyle: style,
              waveDuration: Duration(seconds: 4),
              boxHeight: 120,
              boxWidth: 300,
            ),
            TextLiquidFill(
              text: 'MAGIC',
              waveColor: Colors.blueAccent[700],
              boxBackgroundColor: Colors.orange[600],
              textStyle: style,
              waveDuration: Duration(seconds: 2),
              boxHeight: 120,
              boxWidth: 300,
            ),
          ],
        ),
      );
  }
}
```

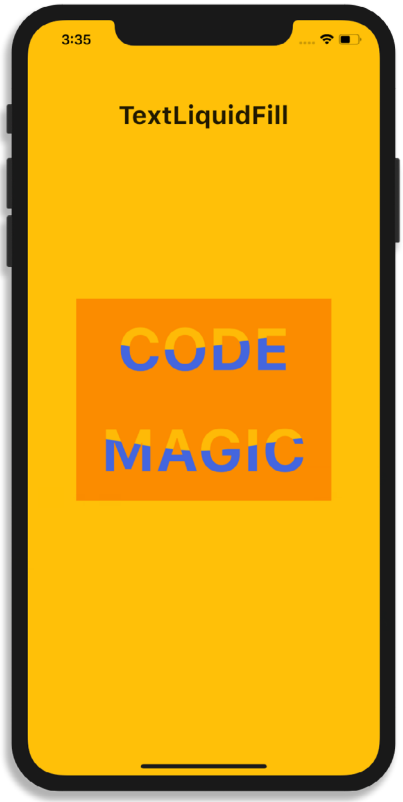
Reference links

Package

Documentation

Wavy Liquid Animation for Text

Sample app



List of text and fonts libraries

- **google_fonts**: Flutter package for accessing the Google Fonts API, allowing you to easily use any of the 977 fonts from fonts.google.com.
- **auto_size_text**: Flutter widget that automatically resizes text to fit perfectly within its bounds.
- **enough_ascii_art**: generates ASCII art using image to ASCII, FIGlet text banner support and emoticon to text conversions.
- **font_awesome_flutter**: The Font Awesome Icon pack available as Flutter Icons. Provides 1500 additional icons to use in your apps.
- **flutter_screenutil**: Flutter plugin for adapting the screen and font size.

UX/UI

Getting **User Interface** and **User Experience** right is a crucial thing in modern applications. Flutter is known as **Google's UI toolkit** for creating beautiful and natively compiled apps.

To further improve your app design, let's take a look at some UI libraries that caught our attention.

The most unique UI library that we want to highlight is VelocityX.

VelocityX

A minimalist UI framework for Flutter

by Pawan Kumar

ANDROID iOS WEB DESKTOP

VelocityX gives you access to all the building blocks you need to create beautiful and responsive UI designs. This package uses **Tailwind CSS**-like property naming and **SwiftUI** style declarative syntax to facilitate rapid development.

> "VelocityX is a minimalist Flutter framework inspired by Tailwind CSS and SwiftUI for building custom designs fast. We decided to make it because most frameworks/ libraries do too much. They come with all sorts of pre-designed components, like buttons, cards, and alerts, that might help you move quickly at first, but cause more pain than gain when it's time to make your app stand out with a custom design."
>
> Pawan Kumar, **creator of VelocityX**

Why VelocityX?

VelocityX can make developers more **productive** because of its declaration style. It uses extension methods to form a chain of properties, rather than using the nested style, which is default in Flutter.

Developer's perspective

Though this package can make you more productive, this property chaining style can be quite intimidating for the beginners.

The best thing about this package is that it makes every widget **responsive**, which is great if you are building Flutter apps for **Web** or **Desktop**.

This makes VelocityX quite popular among everyone who wants to create Flutter apps that are mainly focused on the web and desktop platform. But those who are mostly focused on mobile cross-platform support might prefer the nested style of Flutter. Also, using this package compromises the readability of the code.

Pros and cons

+ Increased **productivity**
+ Highly **responsive UI**
+ **Declarative syntax** similar to SwiftUI

- Loses Flutter's unique declaration style (VelocityX uses **widgets**)
- **Precise control** over styling is **not always possible**
- Reduces **readability** of the code

Example

```
class VelocityDemo extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: SafeArea(
        child: VxBox(
          child: Column(
```

```
            mainAxisAlignment: MainAxisAlignment.start,
            children: <Widget>[
              'VelocityX'.text.orange500.semiBold.size(40).
make().p16(),
              VxZeroList(
                length: 3,
                isDark: true,
                isBottomLinesActive: false,
              ),
              [
                "Velocity 1"
                .text.white.uppercase.size(20).make().box.
rounded.alignCenter.purple600.make().p4(),
                "Velocity 2"
                .text.white.uppercase.size(20).make().box.
rounded.alignCenter.green500.make().p4(),
                "Velocity 3"
                .text.white.uppercase.size(20).make().box.
rounded.alignCenter.orange500.make().p4(),
              ].swiper(enlargeCenterPage: true).py12(),
              'Codemagic'.text.uppercase.red600.bold.
letterSpacing(8).size(40).make().p16(),
            ],
          ),
        ).make(),
      ),
    );
  }
}
```

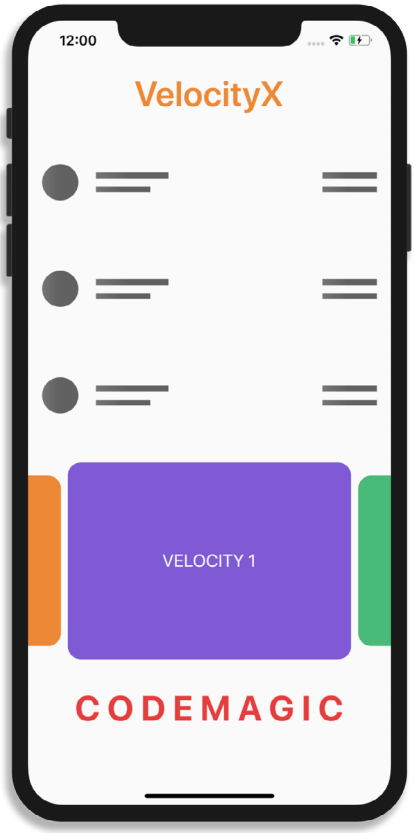
Reference links

Package

Documentation

VelocityX Tutorials on MTECHVIRAL

Sample app



List of UX/UI libraries

- **fl_chart**: powerful Flutter chart library, currently supporting Line Chart, Bar Chart, Pie Chart and Scatter Chart.
- **giffy_dialog**: beautiful and custom alert dialog for Flutter.
- **emoji_picker**: Flutter package that provides an Emoji Keyboard widget with 390 emojis in 8 categories.
- **flutter_webview_plugin**: plugin that allows Flutter to communicate with a native Webview.
- **fluttertoast**: Flutter library for creating toast messages in a single line of code.
- **share**: Flutter plugin to share content from your Flutter app via the platform's share dialog.
- **showcaseview**: Flutter package to showcase/highlight widgets step by step.

LOCATION AND CONNECTIVITY

If you are working with an app that requires you to access some platform-specific services like device location, Bluetooth, WiFi, etc., then you may want a helpful plugin to achieve that. If you are dealing with any of the above, there are some Flutter libraries that may come in handy.

Let us introduce you to a Flutter plugin that allows you to work with locations and gives you easy access to geocoding.

Geolocator

API for generic location services
by Baseflow

ANDROID iOS

Flutter Geolocator plugin provides easy access to the platform-specific location services. It uses FusedLocationProviderClient as default. If this is not available, it uses the LocationManager on **Android** and CLLocationManager on **iOS**.

> "Geolocator is developed by Baseflow. This alone is proof of quality, but in addition to that, it is also one of the plugins that is chosen as a **Flutter Favorite** — and for good reasons. Too often plugins are small frameworks themselves, requiring the user to learn how to use them. This isn't the case with Geolocator, it does one thing, the name is self-explanatory, and it does everything right, with no hassle for the developer."
>
> **Francesco Lapicca, Flutter developer**

Why Geolocator?

Geolocator helps to retrieve the current location of the device comfortably on both the Android and iOS platforms. You can even generate an approximate address based on the coordinates of that location or vice versa. It also allows for fetching the last known location of the device. In addition, this plugin provides an excellent method for determining the distance between two coordinates.

Developer's perspective

Geolocator is an essential library for people dealing with GPS or Maps in their app. This plugin is a perfect fit for the **google_maps_flutter** library, as it often accepts the location of a place in the form of coordinates. Being a quite popular and useful library, it is holding a position in the **Flutter Favorite** package list.

Pros

+ Location addresses can be easily retrieved from the coordinates
+ Background location access is available
+ Distance between two locations can be measured with ease
+ Addresses can be formatted to the specified locale

Example

```
class GeolocationView extends StatefulWidget {
  @override
  _GeolocationViewState createState() => _
GeolocationViewState();
}

class _GeolocationViewState extends State<GeolocationView> {
  TextStyle _style = TextStyle(fontSize: 20);
  final Geolocator _geolocator = Geolocator();
```

```
  Position _currentPosition;
  String _currentAddress;

  // Method for retrieving the current location
  _getCurrentLocation() {
    _geolocator
        .getCurrentPosition(desiredAccuracy: LocationAccuracy.
best)
        .then((Position position) {
      setState(() {
        _currentPosition = position;
      });
      _getAddress();
    }).catchError((e) {
      print(e);
    });
  }

  // Method for retrieving the address
  _getAddress() async {
    try {
      List<Placemark> p = await _geolocator.
placemarkFromCoordinates(
          _currentPosition.latitude, _currentPosition.
longitude);

      Placemark place = p[0];

      setState(() {
        _currentAddress =
            "${place.locality}, ${place.postalCode}, ${place.
country}";
      });
    } catch (e) {
      print(e);
    }
  }
```

```
  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        _currentPosition != null
            ? Column(
                crossAxisAlignment: CrossAxisAlignment.start,
                children: <Widget>[
                  Text(
                    'Latitude:  ${_currentPosition.latitude}',
                    style: _style,
                  ),
                  Text(
                    'Longitude:  ${_currentPosition.
longitude}',
                    style: _style,
                  ),
                  _currentAddress != null
                      ? Text(
                          'Address:  $_currentAddress',
                          style: _style,
                        )
                      : Container(),
                ],
              )
            : Container(),
        Padding(
          padding: const EdgeInsets.only(top: 15.0),
          child: RaisedButton(
            onPressed: _getCurrentLocation,
            child: Text('GET LOCATION', style: _style),
          ),
        )
      ],
    );
  }
}
```

Reference links

Package

Documentation

Sample app

List of location and connectivity libraries

- **google_maps_flutter**: Flutter plugin that provides a Google Maps widget.
- **flutter_polyline_points**: Flutter plugin that decodes a Polyline string into a list of geo-coordinates suitable for showing routes on Maps.
- **connectivity**: Flutter plugin for discovering the state of the network (WiFi and cellular) connectivity.
- **flutter_blue**: Flutter plugin for connecting and communicating with Bluetooth Low Energy devices.

IMAGES AND VIDEOS

One of the most fundamental features for most of the applications are image and video support. Some of the libraries that may help you to integrate images and videos to your Flutter app are listed below. We want to especially highlight an excellent image library that loads images from a given network URL and caches it.

Cached network image

Download, cache and show images in Flutter
by Baseflow

ANDROID iOS WEB MacOS

Cached network image helps to load images from a given network URL and caches them by storing all data in the temporary directory of the app. It uses **sqflite** for storing the image file information, which is later used for retrieving the image from the cache directory if present.

> "Caching and not downloading the same thing again is one of the most important responsibilities of a mobile app developer for both the client side and server side. cached_network_image does this without any additional code. It is also quite customizable and it has a separate image provider, so we can use it completely separately from the widget. Can't even think of developing an app without it."
>
> Serdar Coskun, **Sr. Mobile app developer**

Why cached network image?

Cached network image simplifies the caching process of any image that is to be fetched from the network. You can use it to show any widget while the image is being retrieved. If any error occurs during the retrieval process, it provides a nice property to handle that case as well.

Developer's perspective

This library is very useful for everyone working with **network images** in their app. Most of the well-known apps use this kind of feature to prevent the annoying loading screen from being popped up every time by **caching** commonly used images in the local storage.

Pros and cons

+ Helps to avoid the loading screen
+ In-built property for showing a place-holder widget
+ Handles error cases gracefully

- Takes up local storage space

Example

```
class CachedImageView extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return GridView.builder(
      itemCount: 500,
      gridDelegate: const
SliverGridDelegateWithFixedCrossAxisCount(
        crossAxisCount: 3,
      ),
      itemBuilder: (BuildContext context, int index) =>
```

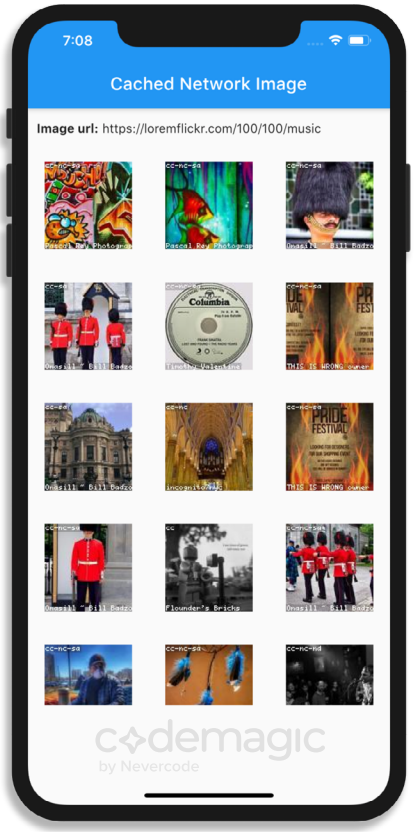
```
CachedNetworkImage(
        imageUrl: 'https://loremflickr.com/100/100/
music?lock=$index',
        placeholder: (context, url) => Center(
          child: CircularProgressIndicator(),
        ),
        errorWidget: (context, url, image) => Center(
          child: Icon(Icons.error),
        ),
      ),
    );
  }
}
```

Reference links

Package

Documentation

Sample app



List of image and video libraries

- **flutter_svg**: SVG rendering and widget library for Flutter that allows painting and displaying Scalable Vector Graphics 1.1 files.
- **image_cropper**: Flutter plugin for Android and iOS that supports cropping images.
- **before_after**: Flutter package that makes it easier to display the difference between two images.
- **image_picker**: Flutter plugin for iOS and Android for picking images from the image library and taking new pictures with the camera.
- **video_player**: Flutter plugin for iOS, Android and Web for playing back video on a widget surface.

DATA PERSISTENCE AND FILE SYSTEM

It is really frustrating for any user of your app if they have to log in every time or if a theme switches back to the default every time they come back to it. To prevent these situations from arising, data persistence and the ability to interact with the device's file system should be an essential part of any app.

Flutter has several libraries to help you with that. Let's take a look at a lightweight and performant pure Dart library that proves to be an excellent solution for the local database.

Hive

Lightweight and blazing fast key-value database
by HiveDB

Hive is a lightweight yet powerful database that is easy to manage and is very performant on every platform. It is a pure Dart package with no native dependencies, so it can even run smoothly on **Flutter Web**.

> "Hive has significantly faster performance than other key–value databases, it is easy to set up and has cross-platform support out of the box. You can also store more complex data, such as lists and maps, very easily. Hive is lightweight for simple Flutter projects and can also be used for large/scalable projects."
>
> **Taha Tesser, Flutter developer & Senior Open Source Support Engineer at Nevercode**

Why Hive?

Hive works seamlessly on all platforms, including mobile, web and desktop. Data is stored in the key-value pair format on the Hive Database. It is strongly encrypted using AES-256 and has a great performance across all the platforms. You can check out the **benchmark** .

Hive supports only regular Dart types out of the box, but behind the scenes it works with binary data. You can define custom types easily using **TypeAdapter** with the help of the **hive_generator** package.

Developer's perspective

Unless you absolutely need to deal with a lot of relations in your database, **Hive** being a pure Dart library is one of the best options out there. If you are not using a heavy-weight state management library, it is very tedious to manually rebuild the UI every time a value changes in the database. In that case, you might appreciate the **hive_flutter** package that monitors for changes and renders the widgets accordingly.

Pros and cons

+ Great performance
+ No native dependencies
+ Simple, powerful and intuitive API
+ Strong encryption
+ Support for TypeAdapters

- Not ideal if you are using a lot of relations in the database

Example

An example showing how to build a simple **Color Generator** using the Hive database.

Initializing Hive:

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();

  // Retrieving the path where to store the Hive database
  final appDocDir = await getApplicationDocumentsDirectory();

  // Initializing Hive
  Hive.init(appDocDir.path);

  // Registering custom adapter
  Hive.registerAdapter(CustomColorAdapter());

  runApp(MyApp());
}
```

Opening a Hive box for storing the key-value pairs in the database:

```
class _HiveViewState extends State<HiveView> {
  @override
  void dispose() {
    // Close all Hive boxes while disposing
    Hive.close();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    // Using a FutureBuilder to open a Hive box with
    // the specified name `colors`
```

```
    return FutureBuilder(
      future: Hive.openBox('colors'),
      builder: (BuildContext context, AsyncSnapshot snapshot)
{
        if (snapshot.connectionState == ConnectionState.done)
{
          if (snapshot.hasError)
            return Text(snapshot.error.toString());
          else
            return ColorPage();
        } else {
          return CircularProgressIndicator();
        }
      },
    );
  }
}
```

UI for showing the ListView of colors and a button for generating random colors:

```
class ColorPage extends StatelessWidget {
  // Defining the Box variable
  final _colorsBox = Hive.box('colors');

  // ...

  @override
  Widget build(BuildContext context) {
    return Column(
      children: <Widget>[
        Expanded(
          // TODO: Show the ListView of colors present
          // in the Box
```

```
          child: Container(),
        ),
        // TODO: Show a button for adding colors to the Box
      ],
    );
  }
}
```

ValueListenableBuilder is for listening to the changes in the Hive database and rendering the widgets accordingly.

```
// For showing the ListView of colors
ValueListenableBuilder(
  valueListenable: _colorsBox.listenable(),
  builder: (context, Box<dynamic> box, _) {
    return ListView.builder(
      itemCount: box.values.length,
      itemBuilder: (BuildContext context, int index) {
        final color = box.getAt(index) as CustomColor;
        return ListTile(
          leading: ClipOval(
            child: Material(
              color: Color(
                  int.parse('0xFF${color.colorHex.
substring(1)}')),
              child: SizedBox(
                width: 50,
                height: 50,
              ),
            ),
          ),
          trailing: Row(
            mainAxisSize: MainAxisSize.min,
            children: <Widget>[
```

```
              // For Updating the color of that index
              IconButton(
                  icon: Icon(Icons.refresh),
                  onPressed: () {
                    // Function generating random colors
                    CustomColor newColor = _generateColor();

                    // Replacing color of that position in
the Hive Box
                    box.putAt(index, newColor);
                  }),

              // For Deleting the color of that index
              IconButton(
                  icon: Icon(Icons.delete),
                  onPressed: () {
                    box.deleteAt(index);
                  }),
            ],
          ),
          title: Text(color.colorName),
          subtitle: Text(color.colorHex),
        );
      },
    );
  }),
```

Button for adding colors to the Hive database:

```
RaisedButton(
  onPressed: () {
    // Function generating random colors
    CustomColor newColor = _generateColor();
```

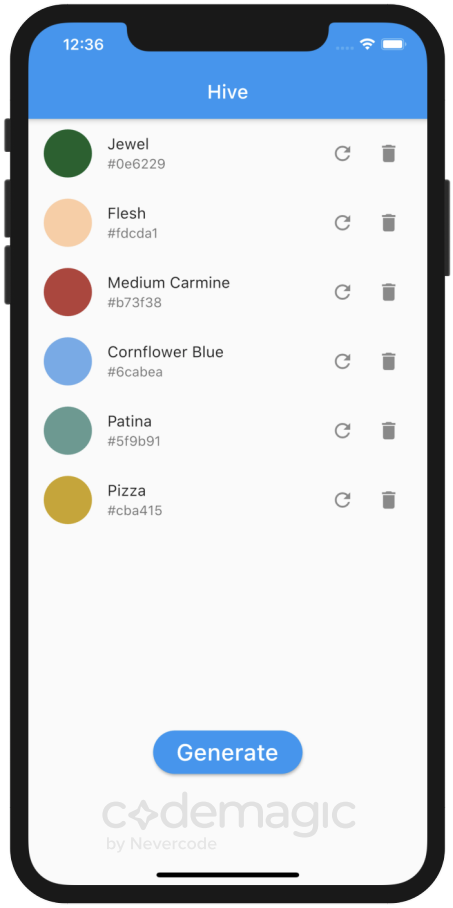
```
      // Adding colors to the Box
      _colorsBox.add(newColor);
    },
    color: Colors.blue,
    child: text,
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(30.0),
    ),
  )
```

Reference links

Package

Documentation

Sample app



List of data persistence and file system libraries

- **shared_preferences**: Flutter plugin for reading and writing simple key-value pairs. Wraps NSUserDefaults on iOS and SharedPreferences on Android.
- **sqflite**: Flutter plugin for SQLite, a self-contained, highly reliable, embedded, SQL database engine.
- **moor**: an easy to use, reactive, type-safe persistence library for Dart and Flutter, built on top of SQLite.
- **path_provider**: Flutter plugin for getting commonly used locations on the Android and iOS file systems.
- **file_access**: Flutter package to handle files on web, desktop and mobile platforms.

ANIMATIONS AND TRANSITIONS

Animations always make your app more attractive and help to enhance the user experience. But overdoing it may also result in a bad UX. Let's take a look at some Flutter packages that can help you add precisely the right amount of animations to your app.

Liquid Swipe

Amazing liquid-like swipe
by Sahdeep Singh

ANDROID iOS

Liquid Swipe is a Flutter package that brings the liquid swipe animation to stacked Container. It adds an interesting liquid animation when swiping between different pages. This is inspired by Cuberto's liquid swipe and IntroViews.

> "Liquid Swipe was all about learning. The thing that makes Liquid Swipe great is its growth, from its initial commit to last, everything was so challenging but fun. Support and encouragement from the community is also a wonderful add-on."
>
> **Sahdeep Singh, creator of Liquid Swipe library**

Why Liquid Swipe?

Liquid Swipe provides great animation without much boilerplate code. The implementation is simple and easy to use with a list of Containers and can be applied to almost any widget within them.

Developer's perspective

Liquid Swipe package would be quite useful for anyone who wants to add some kind of out-of-the-box animation to their app. An animation like this is otherwise quite complex and time-consuming to design but this package makes it simple to implement within a matter of minutes.

Pros and cons

+ Easy to implement
+ Out-of-the-box animation design
+ Can be applied to almost any widget

- Only a few customization options
- Lacks in good user documentation

Example

```
class LiquidSwipeView extends StatelessWidget {
  // List of Containers
  final pages = [
    codemagicPage,
    welcomePage,
  ];

  @override
  Widget build(BuildContext context) {
    return LiquidSwipe(
      pages: pages,
      fullTransitionValue: 200,
      enableSlideIcon: true,
```

```
        enableLoop: true,
        positionSlideIcon: 0.5,
        waveType: WaveType.liquidReveal,
      );
    }
}
```

Reference links

Package

Documentation

Sample app

List of animation and transition libraries

- **animations**: fancy pre-built animations that can easily be integrated into any Flutter application.
- **curved_navigation_bar**: Flutter package for easy implementation of stunning animated curved navigation bar.
- **confetti**: Flutter package for blasting colorful confetti over the screen.
- **intro_slider**: Flutter package for creating a cool intro for your app.
- **flip_panel**: Flutter package for implementing a flip panel animation.
- **flare_flutter**: Flutter package giving you full control of your **Rive** files in the app.
- **flutter_spinkit**: A collection of loading indicators animated with Flutter.

UTILITY

Some assortments of important Flutter libraries have caught our attention. These include libraries that will help you debug your app, get device information, set up authentication, show advertisements and other essential utilities for enhancing your app's functionality. Highlighting a Flutter library that will help you easily add internationalization and localization support to your apps with no tedious setup.

Easy Localization

Easy and fast internationalization

by Aissat

ANDROID iOS WEB MacOS

Easy Localization package simplifies the internationalization and localization process for Flutter apps. For loading translations, it has support for different formats like JSON, CSV, YAML, XML. You can also use the Easy Localization Loader package, which helps to load the translation files quickly.

> "The main reasons for choosing the easy_localization package are the ease of adding translations for many languages and good documentation. There are many additional benefits, like flexible settings, automatic detection of the device locale, options for saving the locale state, pluralization, nested locale keys, RTL locale, etc. All this saves me a lot of time."
>
> **Alexey Z, one of the contributors of easy_localization**

Why Easy Localization?

You can use the official **flutter_localizations** package to internationalize your app but that requires an elaborate setup to get it working. Using Easy Localization, you can not only skip this whole setup process but also get a lot of extra functionalities, which further simplifies the process. It even has a built-in Error widget for missing translations.

Developer's perspective

Though English is widely spoken throughout the world, internationalization is a must if you are releasing your app for a specific region or want to reach a variety of audiences. Easy Localization package not only helps with localization, but it also has support for **plural**, **gender**, **nesting** and **RTL locales**. It supports extension methods on both **Text** and **BuildContext** widgets for easy translation. It is also reactive to locale changes.

Pros

+ Load translations in **any format** (JSON, CSV, YAML, XML)
+ Supports **plural**, **gender**, **nesting**, **RTL locales**
+ Extension methods on **Text** and **BuildContext**
+ Built-in **Error widget**
+ **Code generation** for localization files

Example

Add the EasyLocalization widget inside the main function:

```
void main() {
  runApp(
    EasyLocalization(
      // English, Spanish and Hindi are added as supported
languages
      supportedLocales: [Locale('en'), Locale('es'),
```

```
Locale('hi')],
      // Path to the localization files (JSON)
      path: 'assets/translations',
      fallbackLocale: Locale('en'),
      child: MyApp(),
    ),
  );
}
```

Inside MaterialApp define the following properties:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      localizationsDelegates: context.localizationDelegates,
      supportedLocales: context.supportedLocales,
      locale: context.locale,
      home: HomePage(),
    );
  }
}
```

Then you can easily use this **text** from the translation file (in this case JSON):

```
// `text` is the key for having the translated text for
// each language
Text(
  'text',
  textAlign: TextAlign.center,
  style: _style,
).tr(),
```

To change the locale, you can use the following with the specific language code:

```
// Changing to Spanish
context.locale = Locale('es');
```

Reference links

Package

Documentation

Sample app

List of utility libraries

- **device_preview**: Flutter package to easily preview your app on different devices.
- **fimber**: extensible logging for Flutter — based on **Timber** on Android, using similar method API with same concepts for tree and planting logging tree.
- **flutter_launcher_icons**: package that simplifies updating your Flutter app's launcher icon.
- **intl**: Dart package that provides internationalization and localization facilities, including message translation, plurals and genders, date/number formatting and parsing, and bidirectional text.
- **device_info**: Flutter plugin providing detailed information about the device (make, model, etc.) and the Android or iOS version the app is running on.
- **local_auth**: Flutter plugin for accessing Android and iOS device authentication sensors such as Fingerprint Reader, Touch ID and Face ID.
- **flutter_local_notifications**: plugin for displaying and scheduling local notifications in Flutter applications with the ability to customize for each platform.
- **firebase_admob**: plugin for Flutter that supports loading and displaying banner, interstitial (full-screen), and rewarded video ads using the **Firebase AdMob API**.
- **permission_handler**: Flutter plugin that provides a cross-platform (Android and iOS) API to request permissions and check their status.

CODE GENERATOR AND BUILD TOOLS

Code generation uses the **Dart Build System** builders to generate boilerplate code. It helps to accelerate the development process, prevent errors caused by manually written boilerplate code, and bring apps to production faster. Let's take a look at some libraries that use code generation to create boilerplate code for various functionalities.

There is a native Dart library that helps with the serialization by auto-generating the boilerplate code required for working with JSON in your app.

Json Serializable

Automatically generate code for JSON
by Dart

DART NATIVE

Json Serializable provides Dart Build System builders to generate code for converting to and from JSON by annotating Dart classes. To indicate that a class is serializable you have to annotate it with @JsonSerializable().

> "Whenever you need to generate, read and write JSON, you should use json_serializable. It becomes very powerful when combined with freezed and is a quick solution compared to the built_value. When consuming REST APIs or persisting app state, you really can't go wrong with this package."
>
> **Rody Davis Jr, Developer Advocate at Google**

Why Json Serializable?

Manual serialization is hard to manage as the project grows larger and may lead to errors. **JSON Serializable** library helps you generate code from your model classes. Any typo in hand-written boilerplate code can result in an error during runtime. But you can prevent this by using this library where errors in JSON fields are caught at compile time.

Developer's perspective

If you are dealing with any kind of JSON data retrieved from an API, structuring the data using the model class is very important. But that needs a lot of boilerplate code if it is a large JSON response. You can prevent a huge chunk of this boilerplate using the **JSON Serializable** library. It also provides a number of properties that you can apply to the classes annotated with `@JsonSerializable` and `@JsonKey`. Besides setting arguments on the associated annotation classes, you can also configure code generation by setting values in `build.yaml`.

Pros and cons

+ Generates boilerplate for JSON
+ You can apply properties on annotated classes
+ Prevents Runtime errors due to invalid JSON field
+ There's support for build configuration

- Requires some initial setup
- Might produce visual clutter in the project navigator

Example

The following is a model class for retrieving applications using **Codemagic API**:

```
// application.dart

import 'package:json_annotation/json_annotation.dart';

part 'application.g.dart';

// An annotation for the code generator that this class needs
the
// JSON serialization logic to be generated.
@JsonSerializable()
class Application {
  // To specify that JSON must contain the key,
  // if the key doesn't exist, an exception is thrown.
  // Also as the key name is different than the
  // variable name, so it is specified
  @JsonKey(required: true, name: '_id')
  final String id;
  final String appName;
  final String iconUrl;
  final String lastBuildId;

  Application({
    this.id,
    this.appName,
    this.iconUrl,
    this.lastBuildId,
  });

  factory Application.fromJson(Map<String, dynamic> json) =>
      _$ApplicationFromJson(json);

  Map<String, dynamic> toJson() => _$ApplicationToJson(this);
}
```

The generated code using **Json Serializable** library:

```
// application.g.dart

part of 'application.dart';

Application _$ApplicationFromJson(Map<String, dynamic> json) {
  $checkKeys(json, requiredKeys: const ['_id']);
  return Application(
    id: json['_id'] as String,
    appName: json['appName'] as String,
    iconUrl: json['iconUrl'] as String,
    lastBuildId: json['lastBuildId'] as String,
  );
}

Map<String, dynamic> _$ApplicationToJson(Application instance)
=>
    <String, dynamic>{
      '_id': instance.id,
      'appName': instance.appName,
      'iconUrl': instance.iconUrl,
      'lastBuildId': instance.lastBuildId,
    };
```

Reference links

Package

Documentation

Sample app

List of code generator and build tool libraries

- **build_runner**: build system for Dart code generation and modular compilation.
- **slidy**: CLI package manager and template for Flutter, helps to generate modules, pages, widgets and BLoCs.
- **freezed**: code generator for immutable classes that has a simple syntax/API without compromising on the features.
- **hive_generator**: extension for **Hive** that automatically generates TypeAdapters to store any class.
- **moor_generator**: dev-dependency to generate the table and data-classes together with the **Moor** package.

TESTING

Testing is a must for every app before it gets into production. It helps to prevent bugs and logical errors, which may otherwise result in an unsatisfactory experience for the user. Some Flutter libraries that can make testing easier and faster are listed below.
A testing framework that makes it easy to test classes that depend on live web services or databases is discussed in detail.

Mockito

Mock library for Dart

by Dart

ANDROID iOS WEB

Mockito is a mocking framework written in Dart and inspired by the original Mockito (available in JAVA). It is useful when it comes to unit testing classes that depend on the data fetched from live web services or databases.

> "Whether you are doing test-driven development or you just need to test an API, a repository or any class with a function that returns a value, there is no better tool than Mockito for Dart developer, period. Despite being more of a framework than a package, Mockito is simple to use and makes mocking incredibly easy, allowing you to write clean and professional tests in a fraction of time."
>
> Francesco Lapicca, **Flutter developer**

Why Mockito?

Unit testing classes that depend on **dynamic data**, i.e. data that can change at any time, is quite difficult. Testing this kind of dynamic data with respect to static data defined in your test class may result in an error. Using the Mockito library, you don't have to depend on dynamic data anymore and you can test the logic with fewer errors by mocking the data.

Developer's perspective

Mockito helps to emulate a live web service or database and return specific results depending upon the situation. This allows you to run unit tests faster and reliably. Also, it is quite easy to test all the possible success and failure scenarios using this library.

Pros

+ Prevents errors caused by dynamic data
+ Helps to test all possible scenarios
+ Allows faster test execution

Example

This example shows how to test an **API response** from a live web service using **Mockito** library.

Create a MockClient using the Mock class provided by the **Mockito** package.

```
// fetch_app_test.dart

class MockClient extends Mock implements http.Client {}
```

The fetchApplication() function does one of two things:

1. Returns an application if the HTTP call succeeds
2. Throws an exception if the HTTP call fails

Here, we will test these two cases by mocking them.

```
// fetch_app_test.dart

main() {
  group('fetchApplication', () {
    test('returns an Application if the HTTP call is
successfully', () async {
      final client = MockClient();

      // Use Mockito to return a successful response when it
calls the
      // provided http.Client.
      when(client.get(
        BASE_URL + ENDPOINT,
        headers: {
          "Content-Type": "application/json",
          "x-auth-token": API_TOKEN,
        },
      )).thenAnswer((_) async => http.Response(
            '{"application":{"_id": "1","appName":"sign_in_
flutter","lastBuildId":"123"}}',
            200,
          ));

      expect(await fetchApps(client), isA<Application>());
    });

    test('throws an exception if the HTTP call returns an
error', () {
      final client = MockClient();

      // Use Mockito to return an unsuccessful response when
it calls the
      // provided http.Client.
      when(client.get(
        BASE_URL + ENDPOINT,
```

```
          headers: {
            “Content-Type”: “application/json”,
            “x-auth-token”: API_TOKEN,
          },
        )).thenAnswer((_) async => http.Response(‘Not Found’,
404));

        expect(fetchApps(client), throwsException);
      });
    });
}
```

Reference links

Package

Documentation

Sample app

List of testing libraries

- **e2e**: Flutter package that enables self-driving tests on devices and emulators.
- **bloc_test**: testing library that facilitates testing BLoCs, built to work with **bloc** and **mockito**.
- **angular_test**: test runner and library for **AngularDart**.
- **flutter_gherkin**: Gherkin/Cucumber parser and test runner for Dart and Flutter.

Conclusion

We hope that each and every one of you has found something useful from this ebook. As you can see – there are so many great Flutter libraries! And it's amazing to see how you – Flutter enthusiasts and developers – are creating even more highly useful libraries. Big thank you to all of you!

About the Author

This book was published in cooperation between **Codemagic** and **Souvik Biswas**.

Souvik Biswas is a passionate Mobile App Developer (Android, iOS, and Flutter) with extensive knowledge of UX designs. He has developed several mobile apps throughout his journey, both freelance and personal projects. Apart from mobile apps, he has also worked on platforms like WearOS and VR. As a Technical Writer, he has published over 30+ Flutter articles on Codemagic Blog and Medium – Flutter Community, having over 100K+ views. When not developing apps or writing articles, he is an avid video game player. He is currently pursuing a B.Tech degree in Computer Science and Engineering from the Indian Institute of Information Technology, Kalyani.

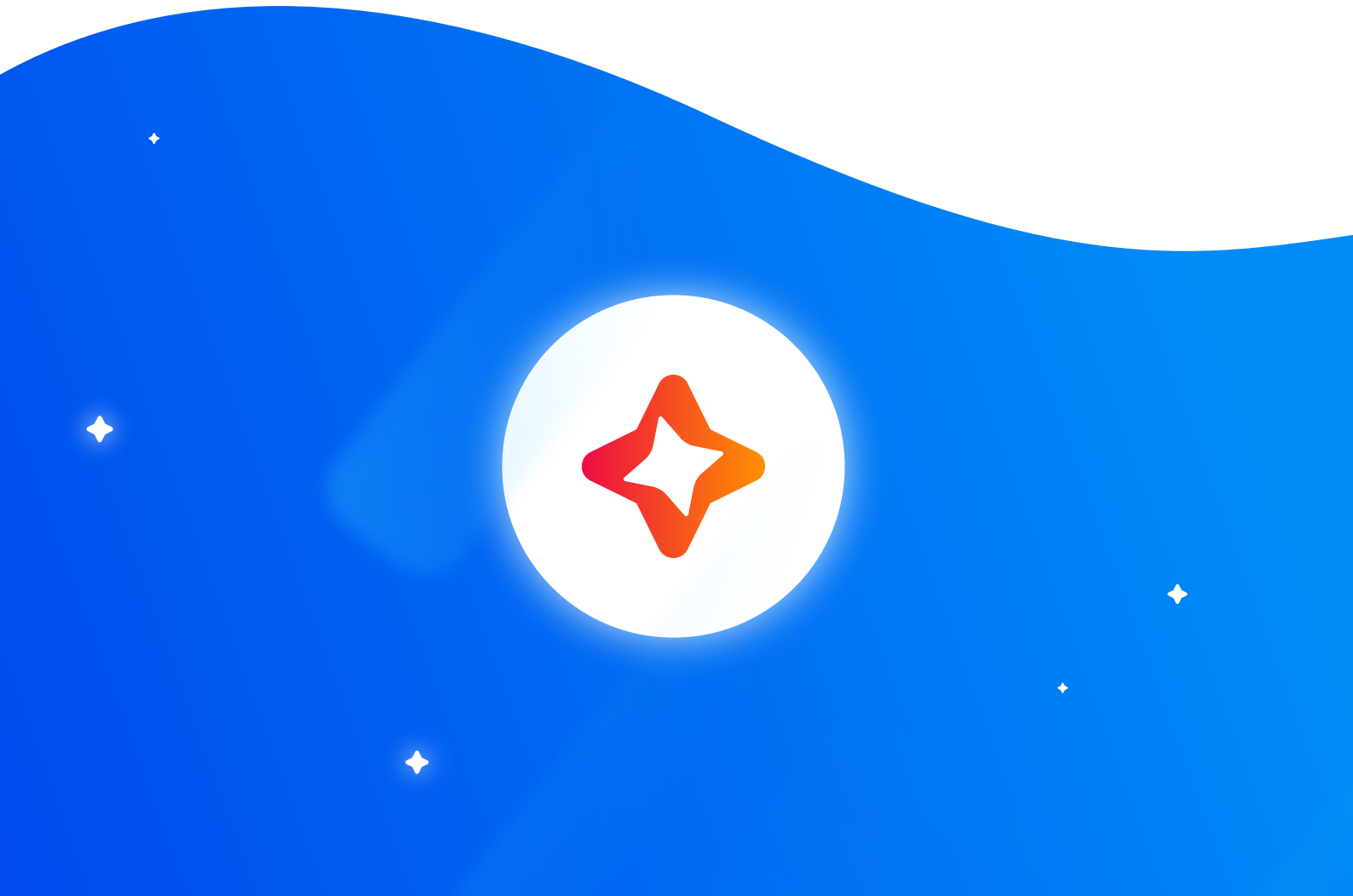
About Codemagic

Codemagic is a productivity tool for professional developers. The idea is to help developers to become more successful and support them with a hassle-free continuous integration platform, so they could concentrate on building awesome apps with shorter time and less errors. Join **Codemagic community on Slack**.

Codemagic offers continuous integration and continuous delivery for Flutter and mobile app projects, including React Native, native iOS and native Android projects.

Build, test and deliver mobile apps in record time with Codemagic CI/CD!

codemagic.io



THANK YOU!

codemagic
by Nevercode