



GSoC Proposal for

## Bug Logging Tool (BLT)

**YASH SOMALKAR**

**INDIAN INSTITUTE OF TECHNOLOGY (BHU),  
VARANASI**

[Github : Yashsomalkar](#)

SLACK: YashSomalkar

[RESUME](#) | Email :

[yashsomalkar@gmail.com](mailto:yashsomalkar@gmail.com)

Phone No : 7057956369 |

[LinkedIn](#)

Timezone : ( GMT + 05:30 ) Asia / Kolkata - IST

## Table of contents :

<b>1. Personal Introduction</b>	<b>3</b>
1.1 Work Experience	3
1.2 Previous Projects	4
1.3 Why do I wish to take part in GSoC 23 ?	4
1.4 Why OWASP ?	5
<b>2. Start with app internationalization and localization (l10n)</b>	<b>6</b>
2.1 File and Code changes.	6
Localization in the iOS apps	8
<b>3. Adding Design for Dark Theme of the App.</b>	<b>12</b>
3.1 Implement a generic theme manager for future themes.	13
Making Custom Themes	14
Adding the Bloc	16
ThemeEvent	16
ThemeState	17
ThemeBloc	17
Changing Themes	18
<b>4. Still editing and adding Ideas.</b>	<b>22</b>

# 1. Personal Introduction

I am Yash Somalkar , a 2nd year undergrad at **Indian Institute of Technology (BHU), Varanasi**. I have been pursuing my Android for the past **1.5 years** and have done internships regarding this field. Android has always been a part of my life even before I started its development, I am always curious about how things work and what happens under the cover. I also have very keen interest in cybersec

## 1.1 Work Experience

These are my responsibilities in my internship period.

1. **Shrusti (NGO) : ([link](#))** I recently had the opportunity to complete a remote internship with a nonprofit organization. During my time there, I was tasked with creating and implementing a **Google App script** for their **Kotlin-based** NGO app. One of the main features I focused on was implementing case-insensitive login validation, which allowed users to log in using either upper or lower case letters in their username and password. In addition to this feature, I also worked on **publishing the app** on the **Google Play store**. This required me to ensure that the app met all of the necessary guidelines and requirements for publishing. It was a challenging but rewarding process, and I was able to see the fruits of my labor when the app was finally made available for download on the Play store. Overall, this internship was a great learning experience and allowed me to develop valuable skills in app development and publishing.

TECHNOLOGIES :   

2. **Open Source Investigator(OSINT) at Saptang Labs :** ([certificate](#)) As an **OSINT** Analyst, I was responsible for finding and **creating APIs** and data points for publicly available information. This involved using a variety of tools and techniques to extract data from online sources such as social media, news articles, and government websites. I worked closely with software developers to identify the key data elements that would be most useful for our organization and **developed scripts** and code to extract this data accurately and efficiently(**Web Scraping**). Additionally, I collaborated with other analysts to analyze the data and identify trends and insights, which helped guide our organization's decision-making processes based on publicly available information.

TECHNOLOGIES :  



## 1.2 Previous Projects

These are some of my previous projects :

1. **Unt(startup work) at UnTangle** : ([link](#)) It is a team project and I am in the core tech team. Utilized Agora's Voice SDK, Firebase Cloud Messaging, and Flutter framework to develop two mobile applications with voice call functionality. Implemented Real-time communication through the integration of **Agora's Voice SDK** and utilized **Firebase Cloud Messaging** for push notifications. Utilized Flutter framework to create visually appealing and intuitive user interfaces.

TECHNOLOGIES :   

2. **Prastuti App for Department TechFest (Prastuti)** : ([Github-link](#)) {[PlayStore-Link](#)} Utilized Flutter to design and develop the user interface for a tech-fest information and event joining platform. Implemented features such as event registration and notification system through the use of Flutter widgets. Utilized design principles to create a visually appealing and intuitive user experience. It has over **250+ downloads** on PlayStore.

TECHNOLOGIES :  

## 1.3 Why do I wish to take part in GSoC 23 ?

I have been pursuing my Android development journey since my 12th standard and have been very enthusiastic to know about new emerging technology related to Android development. In my second year of college, I started to work in startups and service-based companies in order to gain industrial exposure on how to scale code and make it efficient with such a big user base. It also taught me how to write clean, readable code and collaborate with a team while contributing to a large codebase.

From the start of this journey, I had a special place for open source software as all the things I learnt are directly or indirectly related to open source. Be it Gradle or be it Android Studio, everything is open source. Now I want to come back to the roots and apply my knowledge to where I started from. For this open-source journey, I started by participating in Hacktoberfest 22 and successfully completed it. I am now applying for GSoC 23 in order to know more about the open-source community and to apply my knowledge to contribute something back to the open-source community. By this opportunity, I also want to enhance my current knowledge and learn about more efficient workflow strategies related to Android development.

## 1.4 Why OWASP ?

I am thrilled to share that I have embarked on a cybersec journey at the beginning of my first year, fueled by a longstanding fascination with hackers and all things tech. As an avid player of CTF (Capture the Flag), I was ecstatic to be part of a team that secured **3rd position in the CSAW CTF 23** regionally, a prestigious event organized by **NYU (New York University)**.

As my interest in technology continued to grow, I found myself drawn to Android, particularly its Linux-based architecture, which never fails to amaze me. Given my passion for cybersec and Android, it was only natural that I discovered OWASP, an organization that perfectly aligns with my interests and goals.

With all this in mind, I am incredibly excited and fully committed to contributing to the OWASP community through GSoC 23. I believe this program offers a fantastic opportunity for me to develop my skills further, learn from experts in the field, and ultimately make a meaningful impact in the realm of cybersec.

## 2. Start with app internationalization and localization (l10n)

Internationalization is done in a Flutter app to create a localized version for each language or region that you want to support, providing a better user experience for users who prefer to use your app in their native language. This helps to improve the user satisfaction and engagement with your app, and can also help you reach a wider audience and increase the popularity and adoption of your app in different countries or regions.

### 2.1 File and Code changes.

#### Part 1: Setting up your Flutter app

##### Step 1: Add dependencies

First, let's add necessary dependencies into the `pubspec.yaml` file.

```
dependencies:  
  # Other dependencies...  
  flutter_localizations:  
    sdk: flutter  
  intl: ^0.17.0
```

##### Step 2: Enable generation of localization files

To enable automatic generation of localization files, update the `flutter` section of the `pubspec.yaml` file.

```
flutter:  
  generate: true  
  # Other config...
```

### Step 3: Configure localization tool

Create a new `l10n.yaml` file in the root of the Flutter project. This file is going to hold the configuration for the `gen_l10n` tool. You can find the full list of config options in [this document](#). However, in this guide, we will use just a few of them:

- `arb-dir` - the path of the directory that contains the translation files.
- `template-arb-file` - the name of the template arb file that will be used as the basis for generating the Dart localization files.
- `output-localization-file` - the name of the file for the output localization and localizations delegate classes.

Below, you can find the content of the `l10n.yaml` file used in this guide.

```
1  arb-dir: lib/l10n
2  template-arb-file: intl_en.arb
3  output-localization-file: app_localizations.dart
```

### Step 4: Add translation files

add the `l10n` directory with three [ARB](#) files: `intl_ar.arb`, `intl_en.arb`, and `intl_es.arb`. These files are going to hold translations for Arabic, English, and Spanish, respectively. Below, you can see the project structure after adding these files and their content.

```
FLUTTER_PROJECT
|-- ...
|-- android
|-- ios
|-- lib
|   |-- src
|       |-- l10n
|           |-- intl_de.arb
|           |-- intl_fr.arb
|           |-- intl_hi.arb
|           |-- intl_ja.arb
|           |-- intl_ru.arb
|       |-- main.dart
|-- pubspec.yaml
|-- ...
```

The `intl_ar.arb` file:

```
{
  "@@locale": "ar",
  "helloWorld": "مرحبا بالعالم!",
}
```

Similarly other files contain keys and values.

## Localization in the iOS apps

According to the [official documentation](#), for iOS apps, it is necessary to update the `Info.plist` file with the list of supported locales. However, there are [some indications](#) that this could be automated soon. Until that happens, you should manually add the snippet shown below to support localization in iOS apps.

The `ios/Runner/Info.plist` file :

```
<key>CFBundleLocalizations</key>
<array>
  <string>ar</string>
  <string>en</string>
  <string>es</string>
</array>
```

## Step 5: Run the app to trigger code generation

To trigger the generation of the localization files, you need to run the app. After that, you will be able to see the generated code under the `.dart_tool` folder.



```
FLUTTER_PROJECT
|-- .dart_tool
|   |-- ...
|   |-- flutter_gen
|       |-- gen_l10n
|           |-- app_localizations.dart
|           |-- app_localizations_de.dart
|           |-- app_localizations_fr.dart
|           |-- app_localizations_en.dart
|           |-- app_localizations_hi.dart
|           |-- app_localizations_ja.dart
|           |-- app_localizations_ru.dart
|           |-- ...
|-- android
|-- ios
|-- lib
|-- ...
```

By default, the `gen_l10n` tool generates localization files as a synthetic package. Therefore these files are not checked into the version control system (i.e. Git). However, if for some reason you want to track this code with the version control system, you will need to update your `l10n.yaml` file with `synthetic-package` and `output-dir` config params.

the `l10n.yaml` file:

```
# Other config...
synthetic-package: false
output-dir: lib/l10n
```

## Step 6: Update the app

Next, update the `MaterialApp` widget with the `localizationsDelegates` and `supportedLocales` props. For that, import the generated `app_localizations.dart` file and pass needed values.

The `AppLocalizations.localizationsDelegates` represents the list of:

- generated localizations delegate — provides localized messages from ARB files,
- `GlobalMaterialLocalizations.delegate` - provides localized messages for the Material widgets,
- `GlobalCupertinoLocalizations.delegate` - provides localized messages for Cupertino widgets,
- `GlobalWidgetsLocalizations.delegate` - provides text direction for widgets.

These delegates allow us to use different languages throughout the app without much hassle. Moreover, they localize Material's and Cupertino's widgets into ~78 languages (e.g. Material's date picker) and set up text directions within the app.

The `AppLocalizations.supportedLocales` represents the list of supported locales.

The `main.dart` file:

```
import 'package:flutter/material.dart';
import 'package:flutter_gen/gen_l10n/app_localizations.dart';

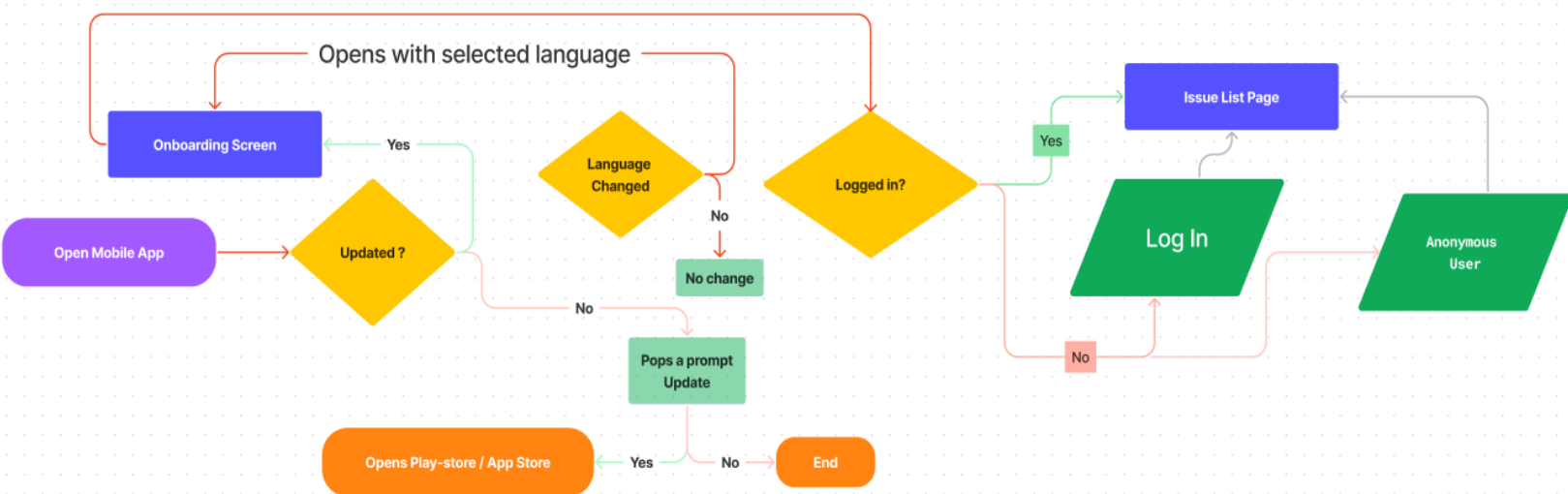
void main() {
  runApp(const MyApp());
}

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

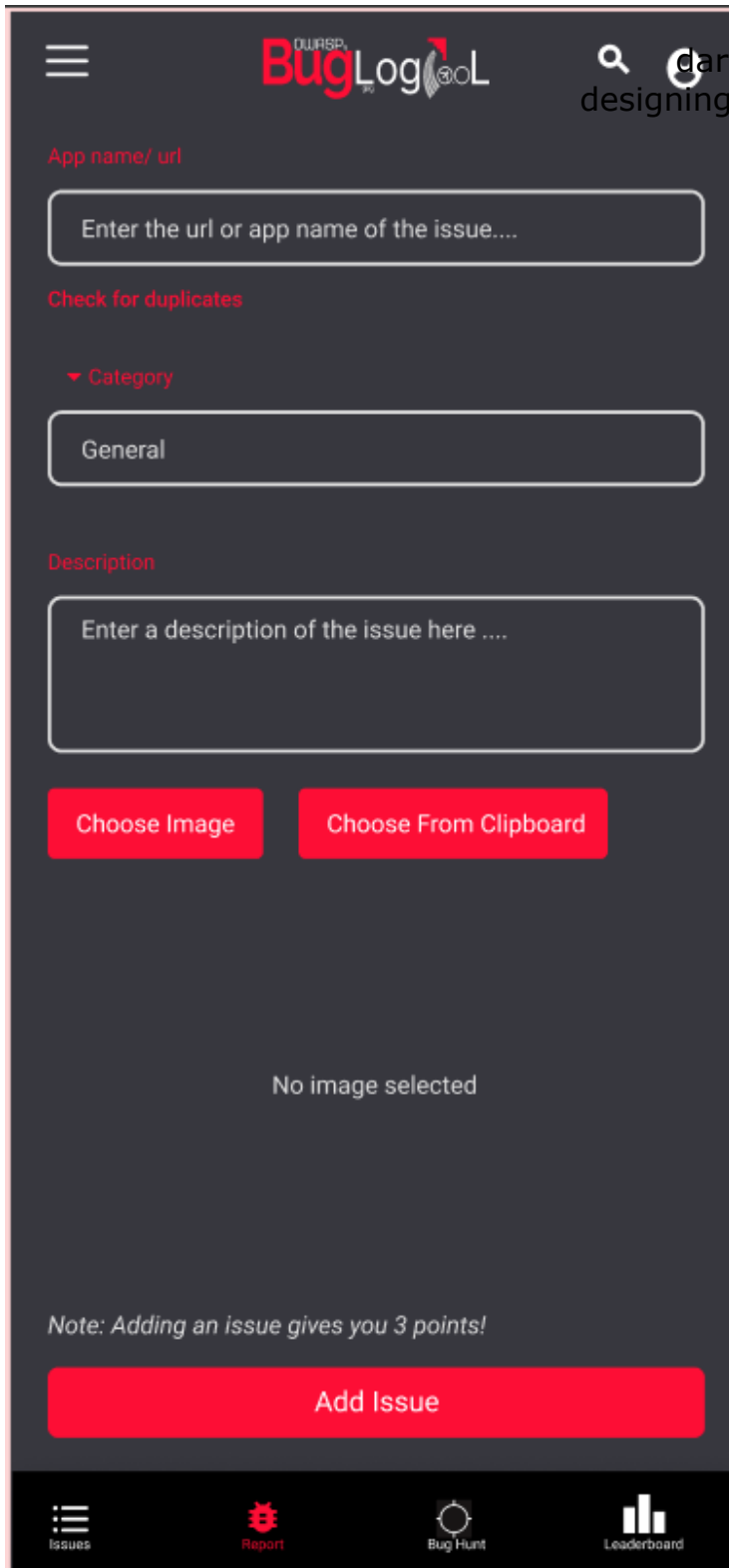
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      // ...
      localizationsDelegates: AppLocalizations.localizationsDelegates,
      supportedLocales: AppLocalizations.supportedLocales,
      // ...
    );
  }
}
```

This is the Example video : [Video](#)

# USER FLOW



### 3. Adding Design for Dark Theme of the App.



The image shows a mobile app interface for BugLog30L in dark mode. At the top, there is a hamburger menu icon, the app logo, a search icon, and a settings icon. Below the header, the form includes a label 'App name/ url' above a text input field with placeholder text 'Enter the url or app name of the issue....'. This is followed by a 'Check for duplicates' label. Then, a 'Category' dropdown menu is shown with 'General' selected. Below that is a 'Description' label above a larger text input field with placeholder text 'Enter a description of the issue here ....'. Two red buttons, 'Choose Image' and 'Choose From Clipboard', are positioned below the description field. Underneath these buttons, the text 'No image selected' is displayed. A note at the bottom of the form states 'Note: Adding an issue gives you 3 points!'. At the very bottom of the form is a large red 'Add Issue' button. The bottom navigation bar contains four icons: 'Issues', 'Report', 'Bug Hunt', and 'Leaderboard'.

Here is the example Design for dark mode, which I would be designing in the course of GSoC

: [Dark-Mode](#)

Based on the sample design I have shared, it is evident that my understanding of how to create a Dark Mode design in Figma is solid. My design showcases consistent use of color, typography, and spacing, which are all essential elements of creating visually appealing and functional Dark Mode designs.

My use of dark background colors, contrasting text colors, and appropriate shadows and highlights demonstrates my understanding of designing for low-light environments.

Furthermore, the seamless transition from a light mode design to a dark mode design is a testament to my proficiency in Figma. I also use this tool : [Colors](#)

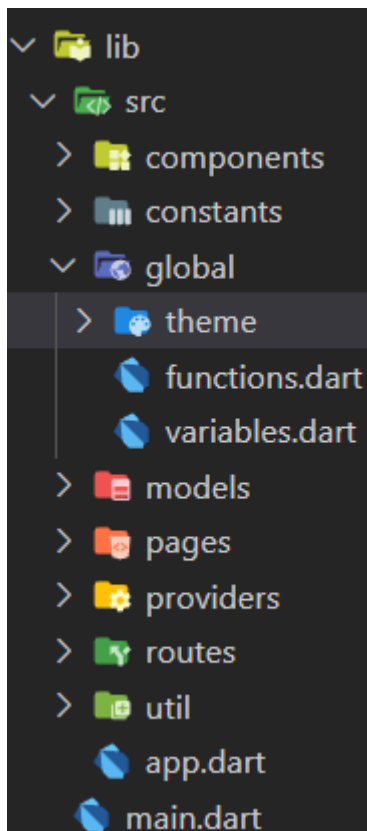
Overall, my Dark Mode design in Figma shows my skills in design, attention to detail, and understanding of the principles of Dark Mode design. With this level of proficiency, I have no doubt that I will be able to create impressive and functional Dark Mode designs for any project I work on.

### 3.1 Implement a generic theme manager for future themes.

We obviously need `flutter_bloc` and we're also going to add `equatable` - this is optional, but should we need value equality, this package will do the repetitive work for us.

pubspec.yaml

```
...
dependencies:
  flutter:
    sdk: flutter
  flutter_bloc: ^0.20.0
  equatable: ^0.4.0
...
```



Create the basic scaffolding of this project. The most important part is the **global/theme** folder - that's where the ThemeBloc and other related code will be located. After all, themes should be applied globally.

## Making Custom Themes

*Here I am demonstrating an example.*

Before doing anything else, we first have to decide on the themes our app will use. Let's keep it simple with only 4 themes - green and blue, both with light and dark variants. Create a new file `app_themes.dart` inside the theme folder.

Flutter uses a class `ThemeData` to, well, store theme data. Since we want to configure 4 distinct instances of `ThemeData`, we will need a simple way to access them. The best option is to use a map. We could use strings for the keys.

### `app_themes.dart`

```
final appThemeData = {  
  "Green Light": ThemeData(  
    brightness: Brightness.light,  
    primaryColor: Colors.green,  
  ),  
  ...  
};
```

As you can imagine, as soon as we add multiple themes, strings will become cumbersome to use. Instead, let's create an **enum** `AppTheme`. The final code will look like this:

***Here I am demonstrating an example, to show the understanding of concepts.***

#### app\_themes.dart

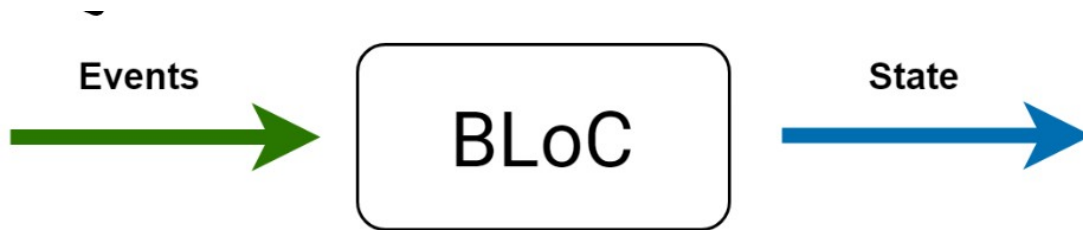
```
import 'package:flutter/material.dart';

enum AppTheme {
  GreenLight,
  GreenDark,
  BlueLight,
  BlueDark,
}

final appThemeData = {
  AppTheme.GreenLight: ThemeData(
    brightness: Brightness.light,
    primaryColor: Colors.green,
  ),
  AppTheme.GreenDark: ThemeData(
    brightness: Brightness.dark,
    primaryColor: Colors.green[700],
  ),
  AppTheme.BlueLight: ThemeData(
    brightness: Brightness.light,
    primaryColor: Colors.blue,
  ),
  AppTheme.BlueDark: ThemeData(
    brightness: Brightness.dark,
    primaryColor: Colors.blue[700],
  ),
};
```

## Adding the Bloc

Having custom themes created, we can move on to create a way for the UI to initiate and also listen to theme changes. This will all happen inside a ThemeBloc which will receive ThemeChanged events, figure out which theme should be displayed, and then output ThemeState containing the proper ThemeData pulled from the Map created above.



Events (e.g user input values) come from one side, then the Bloc (*the pipe*) runs some code in reaction to the event coming in (retrieve weather data, in this case). Finally, the Bloc outputs State (*package containing data*).

## ThemeEvent

There will be only one event ThemeChanged which pass the selected AppTheme enum value to the Bloc.

theme\_event.dart

```

import 'package:equatable/equatable.dart';
import 'package:meta/meta.dart';

import '../app_themes.dart';

@immutable
abstract class ThemeEvent extends Equatable {
  // Passing class fields in a list to the Equatable super class
  ThemeEvent([List props = const []]) : super(props);
}

class ThemeChanged extends ThemeEvent {
  final AppTheme theme;

  ThemeChanged({
    @required this.theme,
  }) : super([theme]);
}
  
```



## ThemeState

Similar to the event, there will be only a single **ThemeState**. We will actually make the generated class concrete (not abstract), instead of creating a new subclass. There can logically be only one theme in the app. On the other hand, there can be multiple events which cause the theme to change.

This state will hold a **ThemeData** object which can be used by the **MaterialApp**.

theme\_state.dart

```
import 'package:equatable/equatable.dart';
import 'package:flutter/material.dart';
import 'package:meta/meta.dart';

@immutable
class ThemeState extends Equatable {
  final ThemeData themeData;

  ThemeState({
    @required this.themeData,
  }) : super([themeData]);
}
```

## ThemeBloc

Bloc is the glue which connects events and states together with logic. Because of the way we've set up the app theme data, **ThemeBloc** will take up only a few simple lines of code.

### theme\_bloc.dart

```
import 'dart:async';
import 'package:bloc/bloc.dart';
import '../app_themes.dart';
import './bloc.dart';

class ThemeBloc extends Bloc<ThemeEvent, ThemeState> {
  @override
  ThemeState get initialState =>
    // Everything is accessible from the appThemeData Map.
    ThemeState(themeData: appThemeData[AppTheme.GreenLight]);

  @override
  Stream<ThemeState> mapEventToState(
    ThemeEvent event,
  ) async* {
    if (event is ThemeChanged) {
      yield ThemeState(themeData: appThemeData[event.theme]);
    }
  }
}
```

## Changing Themes

To apply a theme to the whole app, we have to change the theme property on the root **MaterialApp**. The **ThemeBloc** will also have to be available throughout the whole app. After all, we need to use its state in the aforementioned **MaterialApp** and also dispatch events from the **PreferencePage**, which we are yet to create.

Let's wrap the root widget of our app as a **BlocProvider** and while we're at it, also add a **BlocBuilder** which will rebuild the UI on every state change. Since we're operating with the **ThemeBloc**, the whole UI will be rebuilt when a new **ThemeState** is outputted.

**main.dart**

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

import 'ui/global/theme/bloc/bloc.dart';
import 'ui/home/home_page.dart';

void main() ⇒ runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      builder: (context) ⇒ ThemeBloc(),
      child: BlocBuilder<ThemeBloc, ThemeState>(
        builder: _buildWithTheme,
      ),
    );
  }

  Widget _buildWithTheme(BuildContext context, ThemeState state) {
    return MaterialApp(
      title: 'Material App',
      home: HomePage(),
      theme: state.themeData,
    );
  }
}
```

Of course, the UI won't ever be rebuilt just yet. For that we need to dispatch the **ThemeChanged** event to the **ThemeBloc**. Users will select their preferred theme in the PreferencePage, but first, to make the UI a bit more realistic, let's add a **dummy HomePage**.

home\_page.dart

```
import 'package:flutter/material.dart';
import '../preference/preference_page.dart';

class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Home'),
        actions: <Widget>[
          IconButton(
            icon: Icon(Icons.settings),
            onPressed: () {
              // Navigate to the PreferencePage
              Navigator.of(context).push(MaterialPageRoute(
                builder: (context) => PreferencePage(),
              ));
            },
          ),
        ],
      ),
      body: Center(
        child: Container(
          child: Text(
            'Home',
            style: Theme.of(context).textTheme.display1,
          ),
        ),
      ),
    );
  }
}
```

**PreferencePage** is where the **ThemeChanged** events will be dispatched. Again, because of the way we can access all of the app themes, implementing the UI will be as simple as accessing the appThemeData map in a ListView.

## preference\_page.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:theme_switching_prep/ui/global/theme/app_themes.dart';
import 'package:theme_switching_prep/ui/global/theme/bloc/bloc.dart';

class PreferencePage extends StatelessWidget {
  const PreferencePage({Key key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Preferences'),
      ),
      body: ListView.builder(
        padding: EdgeInsets.all(8),
        itemCount: AppTheme.values.length,
        itemBuilder: (context, index) {
          // Enums expose their values as a list - perfect for ListView
          // Store the theme for the current ListView item
          final itemAppTheme = AppTheme.values[index];
          return Card(
            // Style the cards with the to-be-selected theme colors
            color: appThemeData[itemAppTheme].primaryColor,
            child: ListTile(
              title: Text(
                itemAppTheme.toString(),
                // To show light text with the dark variants...
                style: appThemeData[itemAppTheme].textTheme.body1,
              ),
              onTap: () {
                // This will make the Bloc output a new ThemeState,
                // which will rebuild the UI because of the BlocBuilder in main.dart
                BlocProvider.of<ThemeBloc>(context)
                  .dispatch(ThemeChanged(theme: itemAppTheme));
              },
            ),
          );
        },
      ),
    );
  }
}
```

```
    },  
    );  
  },  
  ),  
  );  
}  
}
```

For the implementation of a generic theme manager in the Flutter App, we can utilize this particular method .

## **4. Still editing and adding Ideas.**