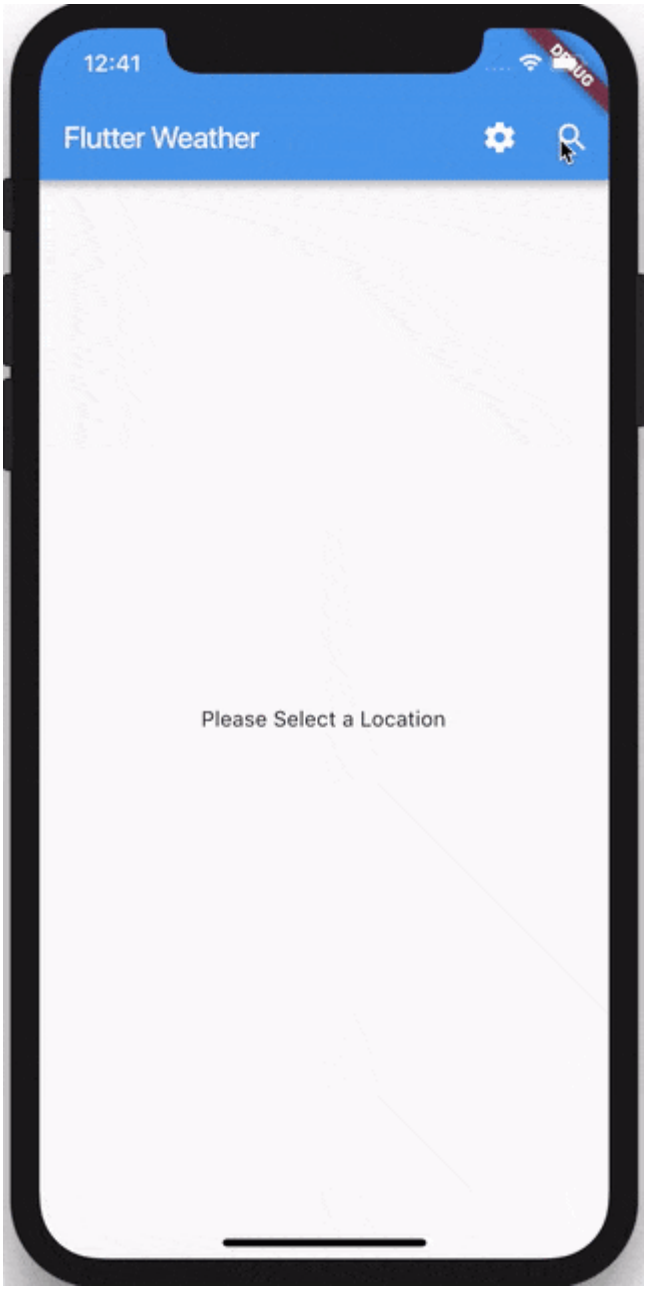
Flutter Weather Tutorial

level advanced

> In the following tutorial, we're going to build a Weather app in Flutter which demonstrates how to manage multiple blocs to implement dynamic theming, pull-to-refresh, and much more. Our weather app will pull real data from an API and demonstrate how to separate our application into three layers (data, business logic, and presentation).



Key Topics

- Observe state changes with BlocObserver.
- BlocProvider, Flutter widget which provides a bloc to its children.
- BlocBuilder, Flutter widget that handles building the widget in response to new states.
- Using Bloc instead of Cubit. What's the difference?
- Prevent unnecessary rebuilds with Equatable.
- RepositoryProvider, a Flutter widget which provides a repository to its children.
- BlocListener, a Flutter widget which invokes the listener code in response to state changes in the bloc.
- MultiBlocProvider, a Flutter widget that merges multiple BlocProvider widgets into one.
- BlocConsumer, exposes a builder and listener in order to react to new states.

Setup

We'll start off by creating a brand new Flutter project

```bash
flutter create flutter_weather
```

We can then go ahead and replace the contents of pubspec.yaml with

```yaml
name: flutter_weather
description: A new Flutter project.

version: 1.0.0+1

environment:
  sdk: ">=2.6.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter
  flutter_bloc: ^5.0.0
  http: ^0.12.0
  equatable: ^1.0.0

dev_dependencies:
  flutter_test:
    sdk: flutter
  mockito: ^4.0.0

flutter:
  uses-material-design: true
  assets:
    - assets/
```

Note: We are going to add some assets (icons for weather types) in our app, so we need to include the assets folder in the pubspec.yaml. Please go ahead and create an *assets* folder in the root of the project.

and then install all of our dependencies

```bash
flutter packages get
```

REST API

For this application we'll be hitting the metaweather API.

We'll be focusing on two endpoints:

- `/api/location/search/?query=$city` to get a locationId for a given city name
- `/api/location/$locationId` to get the weather for a given locationId

Open https://www.metaweather.com/api/location/search/?query=london in your browser and you'll see the following response

```json
[
  {
    "title": "London",
    "location_type": "City",
    "woeid": 44418,
    "latt_long": "51.506321,-0.12714"
  }
]
```

We can then get the where-on-earth-id (woeid) and use it to hit the location api.

Navigate to https://www.metaweather.com/api/location/44418 in your browser and you'll see the response for weather in London. It should look something like this:

```json
{
  "consolidated_weather": [
    {
      "id": 5565095488782336,
      "weather_state_name": "Showers",
      "weather_state_abbr": "s",
      "wind_direction_compass": "WNW",
```

```
      "created": "2019-02-10T19:55:02.434940Z",
      "applicable_date": "2019-02-10",
      "min_temp": 3.75,
      "max_temp": 6.883333333333333,
      "the_temp": 6.885,
      "wind_speed": 10.251177687940428,
      "wind_direction": 288.4087075064449,
      "air_pressure": 998.9649999999999,
      "humidity": 79,
      "visibility": 8.241867493835997,
      "predictability": 73
    },
    {
      "id": 5039805855432704,
      "weather_state_name": "Light Cloud",
      "weather_state_abbr": "lc",
      "wind_direction_compass": "NW",
      "created": "2019-02-10T19:55:02.537745Z",
      "applicable_date": "2019-02-11",
      "min_temp": 1.7699999999999998,
      "max_temp": 8.986666666666666,
      "the_temp": 8.105,
      "wind_speed": 5.198548786091227,
      "wind_direction": 319.24869874195554,
      "air_pressure": 1027.4,
      "humidity": 75,
      "visibility": 11.027785234232084,
      "predictability": 70
    },
    {
      "id": 6214207016009728,
      "weather_state_name": "Heavy Cloud",
      "weather_state_abbr": "hc",
      "wind_direction_compass": "SW",
      "created": "2019-02-10T19:55:02.736577Z",
      "applicable_date": "2019-02-12",
      "min_temp": 3.2699999999999996,
      "max_temp": 11.783333333333333,
      "the_temp": 10.425,
      "wind_speed": 6.291005350509027,
      "wind_direction": 225.7496998927606,
      "air_pressure": 1034.9099999999999,
      "humidity": 77,
```

```
      "visibility": 9.639331305177762,
      "predictability": 71
    },
    {
      "id": 6548160117735424,
      "weather_state_name": "Heavy Cloud",
      "weather_state_abbr": "hc",
      "wind_direction_compass": "SSW",
      "created": "2019-02-10T19:55:02.687267Z",
      "applicable_date": "2019-02-13",
      "min_temp": 3.526666666666667,
      "max_temp": 11.476666666666667,
      "the_temp": 10.695,
      "wind_speed": 6.524550068392587,
      "wind_direction": 203.1296143014564,
      "air_pressure": 1035.775,
      "humidity": 76,
      "visibility": 12.940987135130836,
      "predictability": 71
    },
    {
      "id": 4957149578919936,
      "weather_state_name": "Light Cloud",
      "weather_state_abbr": "lc",
      "wind_direction_compass": "SSE",
      "created": "2019-02-10T19:55:03.487370Z",
      "applicable_date": "2019-02-14",
      "min_temp": 3.4500000000000006,
      "max_temp": 12.540000000000001,
      "the_temp": 12.16,
      "wind_speed": 5.990352212916568,
      "wind_direction": 154.1901674720193,
      "air_pressure": 1035.53,
      "humidity": 71,
      "visibility": 13.873665294679075,
      "predictability": 70
    },
    {
      "id": 5277694765826048,
      "weather_state_name": "Light Cloud",
      "weather_state_abbr": "lc",
      "wind_direction_compass": "S",
      "created": "2019-02-10T19:55:04.800837Z",
```

```
      "applicable_date": "2019-02-15",
      "min_temp": 3.4,
      "max_temp": 12.986666666666666,
      "the_temp": 12.39,
      "wind_speed": 5.359238182348418,
      "wind_direction": 176.84978678797177,
      "air_pressure": 1030.96,
      "humidity": 77,
      "visibility": 9.997862483098704,
      "predictability": 70
    }
  ],
  "time": "2019-02-10T21:49:37.574260Z",
  "sun_rise": "2019-02-10T07:24:19.235049Z",
  "sun_set": "2019-02-10T17:05:51.151342Z",
  "timezone_name": "LMT",
  "parent": {
    "title": "England",
    "location_type": "Region / State / Province",
    "woeid": 24554868,
    "latt_long": "52.883560,-1.974060"
  },
  "sources": [
    {
      "title": "BBC",
      "slug": "bbc",
      "url": "http://www.bbc.co.uk/weather/",
      "crawl_rate": 180
    },
    {
      "title": "Forecast.io",
      "slug": "forecast-io",
      "url": "http://forecast.io/",
      "crawl_rate": 480
    },
    {
      "title": "HAMweather",
      "slug": "hamweather",
      "url": "http://www.hamweather.com/",
      "crawl_rate": 360
    },
    {
      "title": "Met Office",
```

```
        "slug": "met-office",
        "url": "http://www.metoffice.gov.uk/",
        "crawl_rate": 180
      },
      {
        "title": "OpenWeatherMap",
        "slug": "openweathermap",
        "url": "http://openweathermap.org/",
        "crawl_rate": 360
      },
      {
        "title": "Weather Underground",
        "slug": "wunderground",
        "url": "https://www.wunderground.com/?apiref=fc30dc3cd224e19b",
        "crawl_rate": 720
      },
      {
        "title": "World Weather Online",
        "slug": "world-weather-online",
        "url": "http://www.worldweatheronline.com/",
        "crawl_rate": 360
      },
      {
        "title": "Yahoo",
        "slug": "yahoo",
        "url": "http://weather.yahoo.com/",
        "crawl_rate": 180
      }
    ],
    "title": "London",
    "location_type": "City",
    "woeid": 44418,
    "latt_long": "51.506321,-0.12714",
    "timezone": "Europe/London"
  }
```


Great, now that we know what our data is going to look like, let's create the necessary data models.

Creating Our Weather Data Model

Even though the weather API returns weather for multiple days, for simplicity, we're only going to worry about today's weather.

Let's start off by creating a folder for our models `lib/models` and create a file in there called `weather.dart` which will hold our data model for our `Weather` class. Next inside of `lib/models` create a file called `models.dart` which is our barrel file where we export all models from.

Imports

First off we need to import our dependencies for our class. At the top of `weather.dart` go ahead and add:

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

- `equatable` : Package that allows comparisons between objects without having to override the `==` operator

Create WeatherCondition Enum

Next we are going to create an enumerator for all our possible weather conditions. On the next line, let's add the enum.

*These conditions come from the definition of the **metaweather API***

```dart
enum WeatherCondition {
  snow,
  sleet,
  hail,
  thunderstorm,
  heavyRain,
  lightRain,
  showers,
  heavyCloud,
  lightCloud,
  clear,
  unknown
}
```

Create Weather Model

Next we need to create a class to be our defined data model for the weather object returned from the API. We are going to extract a subset of the data from the API and create a `Weather` model. Go ahead and add this to the `weather.dart` file below the `WeatherCondition` enum.

```dart
class Weather extends Equatable {
  final WeatherCondition condition;
  final String formattedCondition;
  final double minTemp;
  final double temp;
  final double maxTemp;
  final int locationId;
  final String created;
  final DateTime lastUpdated;
  final String location;

  const Weather({
    this.condition,
    this.formattedCondition,
    this.minTemp,
    this.temp,
    this.maxTemp,
    this.locationId,
    this.created,
    this.lastUpdated,
    this.location,
  });

  @override
  List<Object> get props => [
        condition,
        formattedCondition,
        minTemp,
        temp,
        maxTemp,
        locationId,
        created,
        lastUpdated,
        location,
      ];
```

```
static Weather fromJson(dynamic json) {
  final consolidatedWeather = json['consolidated_weather'][0];
  return Weather(
    condition: _mapStringToWeatherCondition(
        consolidatedWeather['weather_state_abbr']),
    formattedCondition: consolidatedWeather['weather_state_name'],
    minTemp: consolidatedWeather['min_temp'] as double,
    temp: consolidatedWeather['the_temp'] as double,
    maxTemp: consolidatedWeather['max_temp'] as double,
    locationId: json['woeid'] as int,
    created: consolidatedWeather['created'],
    lastUpdated: DateTime.now(),
    location: json['title'],
  );
}

static WeatherCondition _mapStringToWeatherCondition(String input) {
  WeatherCondition state;
  switch (input) {
    case 'sn':
      state = WeatherCondition.snow;
      break;
    case 'sl':
      state = WeatherCondition.sleet;
      break;
    case 'h':
      state = WeatherCondition.hail;
      break;
    case 't':
      state = WeatherCondition.thunderstorm;
      break;
    case 'hr':
      state = WeatherCondition.heavyRain;
      break;
    case 'lr':
      state = WeatherCondition.lightRain;
      break;
    case 's':
      state = WeatherCondition.showers;
      break;
    case 'hc':
      state = WeatherCondition.heavyCloud;
```

```
          break;
        case 'lc':
          state = WeatherCondition.lightCloud;
          break;
        case 'c':
          state = WeatherCondition.clear;
          break;
        default:
          state = WeatherCondition.unknown;
      }
      return state;
    }
  }
```

We extend `Equatable` so that we can compare `Weather` instances. By default, the equality operator returns true if and only if this and other are the same instance.

There's not much happening here; we are just defining our `Weather` data model and implementing a `fromJson` method so that we can create a `Weather` instance from the API response body and creating a method that maps the raw string to a `WeatherCondition` in our enum.

Export in Barrel

Now we need to export this class in our barrel file. Open up `lib/models/models.dart` and add the following line of code:

```dart
export 'weather.dart';
```

Data Provider

Next, we need to build our `WeatherApiClient` which will be responsible for making http requests to the weather API.

> The `WeatherApiClient` is the lowest layer in our application architecture (the data provider). It's only responsibility is to fetch data directly from our API.

As we mentioned earlier, we are going to be hitting two endpoints so our `WeatherApiClient` needs to expose two public methods:

- `getLocationId(String city)`
- `fetchWeather(int locationId)`

Creating our Weather API Client

This layer of our application is called the repository layer, so let's go ahead and create a folder for our repositories. Inside of `lib/` create a folder called `repositories` and then create a file called `weather_api_client.dart`.

Adding a Barrel

Same as we did with our models, let's create a barrel file for our repositories. Inside of `lib/repositories` go ahead and add a file called `repositories.dart` and leave it blank for now.

- `models`: Lastly, we import our `Weather` model we created earlier.

Create Our WeatherApiClient class

Let's create a class. Go ahead and add this:

```dart
class WeatherApiClient {
  static const baseUrl = 'https://www.metaweather.com';
  final http.Client httpClient;

  WeatherApiClient({
    @required this.httpClient,
  }) : assert(httpClient != null);
}
```

Here we are creating a constant for our base URL and instantiating our http client. Then we are creating our Constructor and requiring that we inject an instance of

httpClient. You'll see some missing dependencies. Let's go ahead and add them to the top of the file:

```dart
import 'package:meta/meta.dart';
import 'package:http/http.dart' as http;
```

- `meta` : Defines annotations that can be used by the tools that are shipped with the Dart SDK.
- `http` : A composable, Future-based library for making HTTP requests.

Add getLocationId Method

Now let's add our first public method, which will get the locationId for a given city. Below the constructor, go ahead and add:

```dart
Future<int> getLocationId(String city) async {
  final locationUrl = '$baseUrl/api/location/search/?query=$city';
  final locationResponse = await this.httpClient.get(locationUrl);
  if (locationResponse.statusCode != 200) {
    throw Exception('error getting locationId for city');
  }

  final locationJson = jsonDecode(locationResponse.body) as List;
  return (locationJson.first)['woeid'];
}
```

Here we are just making a simple HTTP request and then decoding the response as a list. Speaking of decoding, you'll see `jsonDecode` is a function from a dependency we need to import. So let's go ahead and do that now. At the top of the file by the other imports go ahead and add:

```dart
import 'dart:convert';
```

- `dart:convert` : Encoder/Decoder for converting between different data representations, including JSON and UTF-8.

Add fetchWeather Method

Next up let's add our other method to hit the metaweather API. This one will get the weather for a city given it's locationId. Below the `getLocationId` method we just implemented, let's go ahead and add this:

```dart
Future<Weather> fetchWeather(int locationId) async {
  final weatherUrl = '$baseUrl/api/location/$locationId';
  final weatherResponse = await this.httpClient.get(weatherUrl);

  if (weatherResponse.statusCode != 200) {
    throw Exception('error getting weather for location');
  }

  final weatherJson = jsonDecode(weatherResponse.body);
  return Weather.fromJson(weatherJson);
}
```

Here again we are just making a simple HTTP request and decoding the response into JSON. You'll notice we again need to import a dependency, this time our `Weather` model. At the top of the file, go ahead and import it like so:

```dart
import 'package:flutter_weather/models/models.dart';
```

Export WeatherApiClient

Now that we have our class created with our two methods, let's go ahead and export it in the barrel file. Inside of `repositories.dart` go ahead and add:

```dart
export 'weather_api_client.dart';
```

What next

We've got our `DataProvider` done so it's time to move up to the next layer of our app's architecture: the **repository layer**.

Repository

> The `WeatherRepository` serves as an abstraction between the client code and the data provider so that as a developer working on features, you don't have to know where the data is coming from. Our `WeatherRepository` will have a dependency on our `WeatherApiClient` that we just created and it will expose a single public method called, you guessed it, `getWeather(String city)`. No one needs to know that under the hood we need to make two API calls (one for locationId and one for weather) because no one really cares. All we care about is getting the `Weather` for a given city.

Creating Our Weather Repository

This file can live in our repository folder. So go ahead and create a file called `weather_repository.dart` and open it up.

Our `WeatherRepository` is quite simple and should look something like this:

```dart
import 'dart:async';

import 'package:meta/meta.dart';

import 'package:flutter_weather/repositories/weather_api_client.dart';
import 'package:flutter_weather/models/models.dart';

class WeatherRepository {
  final WeatherApiClient weatherApiClient;

  WeatherRepository({@required this.weatherApiClient})
      : assert(weatherApiClient != null);

  Future<Weather> getWeather(String city) async {
    final int locationId = await weatherApiClient.getLocationId(city);
    return weatherApiClient.fetchWeather(locationId);
  }
}
```

Export WeatherRepository in Barrel

Go ahead and open up `repositories.dart` and export this like so:

```dart
export 'weather_repository.dart';
```

Awesome! We are now ready to move up to the business logic layer and start building our `WeatherBloc`.

Business Logic (Bloc)

> Our `WeatherBloc` is responsible for receiving `WeatherEvents` and converting them into `WeatherStates`. It will have a dependency on `WeatherRepository` so that it can retrieve the `Weather` when a user inputs a city of their choice.

Creating Our First Bloc

We will create a few Blocs during this tutorial, so lets create a folder inside of `lib` called `blocs`. Again since we will have multiple blocs, let's first create a barrel file called `blocs.dart` inside our `blocs` folder.

Before jumping into the Bloc we need to define what events our `WeatherBloc` will be handling as well as how we are going to represent our `WeatherState`. To keep our files small, we will separate `event` `state` and `bloc` into three files.

Weather Event

Let's create a file called `weather_event.dart` inside of the `blocs` folder. For simplicity, we're going to start off by having a single event called `WeatherRequested`.

We can define it like:

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

abstract class WeatherEvent extends Equatable {
  const WeatherEvent();
```

```
}

class WeatherRequested extends WeatherEvent {
  final String city;

  const WeatherRequested({@required this.city}) : assert(city != null);

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

Whenever a user inputs a city, we will `add` a `WeatherRequested` event with the given city and our bloc will be responsible for figuring out what the weather is there and returning a new `WeatherState`.

Then let's export the class in our barrel file. Inside of `blocs.dart` please add:

```dart
export 'weather_event.dart';
```

Weather State

Next up let's create our `state` file. Inside of the `blocs` folder go ahead and create a file named `weather_state.dart` where our `weatherState` will live.

For the current application, we will have 4 possible states:

- `WeatherInitial` - our initial state which will have no weather data because the user has not yet selected a city
- `WeatherLoadInProgress` - a state which will occur while we are fetching the weather for a given city
- `WeatherLoadSuccess` - a state which will occur if we were able to successfully fetch weather for a given city.
- `WeatherLoadFailure` - a state which will occur if we were unable to fetch weather for a given city.

We can represent these states like so:

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

import 'package:flutter_weather/models/models.dart';

abstract class WeatherState extends Equatable {
  const WeatherState();

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

class WeatherInitial extends WeatherState {}

class WeatherLoadInProgress extends WeatherState {}

class WeatherLoadSuccess extends WeatherState {
  final Weather weather;

  const WeatherLoadSuccess({@required this.weather}) : assert(weather !=

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

class WeatherLoadFailure extends WeatherState {}
```

Then let's export this class in our barrel file. Inside of `blocs.dart` go ahead and add:

```dart
export 'weather_state.dart';
```

Now that we have our `Events` and our `States` defined and implemented we are ready to make our `WeatherBloc`.

Weather Bloc

> Our `WeatherBloc` is very straightforward. To recap, it converts `WeatherEvents` into `WeatherStates` and has a dependency on the `WeatherRepository`.

Tip: Check out the Bloc VSCode Extension in order to take advantage of the bloc snippets and even further improve your efficiency and development speed.

Go ahead and create a file inside of the `blocs` folder called `weather_bloc.dart` and add the following:

```dart
import 'package:meta/meta.dart';
import 'package:bloc/bloc.dart';

import 'package:flutter_weather/repositories/repositories.dart';
import 'package:flutter_weather/models/models.dart';
import 'package:flutter_weather/blocs/blocs.dart';

class WeatherBloc extends Bloc<WeatherEvent, WeatherState> {
  final WeatherRepository weatherRepository;

  WeatherBloc({@required this.weatherRepository})
      : assert(weatherRepository != null),
      super(WeatherInitial());

  @override
  Stream<WeatherState> mapEventToState(WeatherEvent event) async* {
    if (event is WeatherRequested) {
      yield WeatherLoadInProgress();
      try {
        final Weather weather = await weatherRepository.getWeather(event.
        yield WeatherLoadSuccess(weather: weather);
      } catch (_) {
        yield WeatherLoadFailure();
      }
    }
  }
}
```

We set our initial state to `WeatherInitial` since initially, the user has not selected a city. Then, all that's left is to implement `mapEventToState`.

Since we are only handling the `WeatherRequested` event all we need to do is `yield` our `WeatherLoadInProgress` state when we get a `WeatherRequested` event and then try to get the weather from the `WeatherRepository`.

If we are able to successfully retrieve the weather we then `yield` a `WeatherLoadSuccess` state and if we are unable to retrieve the weather, we `yield` a `WeatherLoadFailure` state.

Now export this class in `blocs.dart`:

```dart
export 'weather_bloc.dart';
```

That's all there is to it! Now we're ready to move on to the final layer: the presentation layer.

Presentation

Setup

As you've probably already seen in other tutorials, we're going to create a `SimpleBlocObserver` so that we can see all state transitions in our application. Let's go ahead and create `simple_bloc_observer.dart` and create our own custom observer.

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

class SimpleBlocObserver extends BlocObserver {
  @override
  void onEvent(Bloc bloc, Object? event) {
    super.onEvent(bloc, event);
    print('onEvent $event');
  }

  @override
```

```
    onTransition(Bloc bloc, Transition transition) {
      super.onTransition(bloc, transition);
      print('onTransition $transition');
    }

    @override
    void onError(BlocBase bloc, Object error, StackTrace stackTrace) {
      print('onError $error');
      super.onError(bloc, error, stackTrace);
    }
  }
```

We can then import it into `main.dart` file and set our observer like so:

```dart
import 'package:flutter_weather/simple_bloc_observer.dart';

void main() {
  Bloc.observer = SimpleBlocObserver();
  runApp(App());
}
```

Lastly, we need to create our `WeatherRepository` and inject it into our `App` widget (which we will create in the next step).

```dart
import 'package:flutter_weather/repositories/repositories.dart';
import 'package:http/http.dart' as http;

void main() {
  Bloc.observer = SimpleBlocObserver();

  final WeatherRepository weatherRepository = WeatherRepository(
    weatherApiClient: WeatherApiClient(
      httpClient: http.Client(),
    ),
  );

  runApp(App(weatherRepository: weatherRepository));
}
```

App Widget

Our `App` widget is going to start off as a `StatelessWidget` which has the `WeatherRepository` injected and builds the `MaterialApp` with our `Weather` widget (which we will create in the next step). We are using the `BlocProvider` widget to create an instance of our `WeatherBloc` and make it available to the `Weather` widget and its children. In addition, the `BlocProvider` manages building and closing the `WeatherBloc`.

```dart
class App extends StatelessWidget {
  final WeatherRepository weatherRepository;

  App({Key key, @required this.weatherRepository})
      : assert(weatherRepository != null),
        super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Weather',
      home: BlocProvider(
        create: (context) =>
            WeatherBloc(weatherRepository: weatherRepository),
        child: Weather(),
      ),
    );
  }
}
```

Weather

Now we need to create our `Weather` Widget. Go ahead and make a folder called `widgets` inside of `lib` and create a barrel file inside called `widgets.dart`. Next create a file called `weather.dart`.

> Our Weather Widget will be a `StatelessWidget` responsible for rendering the various weather data.

Creating Our Stateless Widget

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

import 'package:flutter_bloc/flutter_bloc.dart';

import 'package:flutter_weather/widgets/widgets.dart';
import 'package:flutter_weather/blocs/blocs.dart';

class Weather extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Flutter Weather'),
        actions: <Widget>[
          IconButton(
            icon: Icon(Icons.search),
            onPressed: () async {
              final city = await Navigator.push(
                context,
                MaterialPageRoute(
                  builder: (context) => CitySelection(),
                ),
              );
              if (city != null) {
                BlocProvider.of<WeatherBloc>(context)
                    .add(WeatherRequested(city: city));
              }
            },
          )
        ],
      ),
      body: Center(
        child: BlocBuilder<WeatherBloc, WeatherState>(
          builder: (context, state) {
            if (state is WeatherInitial) {
              return Center(child: Text('Please Select a Location'));
            }
            if (state is WeatherLoadInProgress) {
              return Center(child: CircularProgressIndicator());
            }
```

```
            if (state is WeatherLoadSuccess) {
              final weather = state.weather;

              return ListView(
                children: <Widget>[
                  Padding(
                    padding: EdgeInsets.only(top: 100.0),
                    child: Center(
                      child: Location(location: weather.location),
                    ),
                  ),
                  Center(
                    child: LastUpdated(dateTime: weather.lastUpdated),
                  ),
                  Padding(
                    padding: EdgeInsets.symmetric(vertical: 50.0),
                    child: Center(
                      child: CombinedWeatherTemperature(
                        weather: weather,
                      ),
                    ),
                  ),
                ],
              );
            }
            if (state is WeatherLoadFailure) {
              return Text(
                'Something went wrong!',
                style: TextStyle(color: Colors.red),
              );
            }
          },
        ),
      ),
    );
  }
}
```

All that's happening in this widget is we're using `BlocBuilder` with our `WeatherBloc` in order to rebuild our UI based on state changes in our `WeatherBloc`.

Go ahead and export `Weather` in the `widgets.dart` file.

You'll notice that we are referencing a `CitySelection`, `Location`, `LastUpdated`, and `CombinedWeatherTemperature` widget which we will create in the following sections.

Location Widget

Go ahead and create a file called `location.dart` inside of the `widgets` folder.

> Our `Location` widget is simple; it displays the current location.

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

class Location extends StatelessWidget {
  final String location;

  Location({Key key, @required this.location})
      : assert(location != null),
        super(key: key);

  @override
  Widget build(BuildContext context) {
    return Text(
      location,
      style: TextStyle(
        fontSize: 30,
        fontWeight: FontWeight.bold,
        color: Colors.white,
      ),
    );
  }
}
```

Make sure to export this in the `widgets.dart` file.

Last Updated

Next up create a `last_updated.dart` file inside the `widgets` folder.

> Our `LastUpdated` widget is also super simple; it displays the last updated time so that users know how fresh the weather data is.

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

class LastUpdated extends StatelessWidget {
  final DateTime dateTime;

  LastUpdated({Key key, @required this.dateTime})
      : assert(dateTime != null),
        super(key: key);

  @override
  Widget build(BuildContext context) {
    return Text(
      'Updated: ${TimeOfDay.fromDateTime(dateTime).format(context)}',
      style: TextStyle(
        fontSize: 18,
        fontWeight: FontWeight.w200,
        color: Colors.white,
      ),
    );
  }
}
```

Make sure to export this in the `widgets.dart` file.

Note: We are using `TimeOfDay` to format the `DateTime` into a more human-readable format.

Combined Weather Temperature

Next up create a `combined_weather_temperature.dart` file inside the `widgets` folder.

> The `CombinedWeatherTemperature` widget is a compositional widget which displays the current weather along with the temperature. We are still going to modularize the `Temperature` and `WeatherConditions` widgets so that they can all be reused.

dart

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

import 'package:flutter_weather/models/models.dart' as model;
import 'package:flutter_weather/widgets/widgets.dart';

class CombinedWeatherTemperature extends StatelessWidget {
  final model.Weather weather;

  CombinedWeatherTemperature({
    Key key,
    @required this.weather,
  })  : assert(weather != null),
        super(key: key);

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Row(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Padding(
              padding: EdgeInsets.all(20.0),
              child: WeatherConditions(condition: weather.condition),
            ),
            Padding(
              padding: EdgeInsets.all(20.0),
              child: Temperature(
                temperature: weather.temp,
                high: weather.maxTemp,
                low: weather.minTemp,
              ),
            ),
          ],
        ),
        Center(
```

```
            child: Text(
              weather.formattedCondition,
              style: TextStyle(
                fontSize: 30,
                fontWeight: FontWeight.w200,
                color: Colors.white,
              ),
            ),
          ),
        ],
      );
    }
  }
```

Make sure to export this in the `widgets.dart` file.

Note: We are using two unimplemented widgets: `WeatherConditions` and `Temperature` which we will create next.

Weather Conditions

Next up create a `weather_conditions.dart` file inside the `widgets` folder.

> Our `WeatherConditions` widget will be responsible for displaying the current weather conditions (clear, showers, thunderstorms, etc...) along with a matching icon.

dart

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

import 'package:flutter_weather/models/models.dart';

class WeatherConditions extends StatelessWidget {
  final WeatherCondition condition;

  WeatherConditions({Key key, @required this.condition})
      : assert(condition != null),
        super(key: key);
```

```
    @override
    Widget build(BuildContext context) => _mapConditionToImage(condition);

    Image _mapConditionToImage(WeatherCondition condition) {
      Image image;
      switch (condition) {
        case WeatherCondition.clear:
        case WeatherCondition.lightCloud:
          image = Image.asset('assets/clear.png');
          break;
        case WeatherCondition.hail:
        case WeatherCondition.snow:
        case WeatherCondition.sleet:
          image = Image.asset('assets/snow.png');
          break;
        case WeatherCondition.heavyCloud:
          image = Image.asset('assets/cloudy.png');
          break;
        case WeatherCondition.heavyRain:
        case WeatherCondition.lightRain:
        case WeatherCondition.showers:
          image = Image.asset('assets/rainy.png');
          break;
        case WeatherCondition.thunderstorm:
          image = Image.asset('assets/thunderstorm.png');
          break;
        case WeatherCondition.unknown:
          image = Image.asset('assets/clear.png');
          break;
      }
      return image;
    }
  }
```

Make sure to export this in the `widgets.dart` file.

Here you can see we are using some assets. Please download them from here and add them to the `assets/` directory we created at the beginning of the project.

> **Tip:** Check out icons8 for the assets used in this tutorial.

Temperature

Next up create a `temperature.dart` file inside the `widgets` folder.

> Our `Temperature` widget will be responsible for displaying the average, min, and max temperatures.

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

class Temperature extends StatelessWidget {
  final double temperature;
  final double low;
  final double high;

  Temperature({
    Key key,
    this.temperature,
    this.low,
    this.high,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Row(
      children: [
        Padding(
          padding: EdgeInsets.only(right: 20.0),
          child: Text(
            '${_formattedTemperature(temperature)}°',
            style: TextStyle(
              fontSize: 32,
              fontWeight: FontWeight.w600,
              color: Colors.white,
            ),
          ),
        ),
        Column(
          children: [
            Text(
              'max: ${_formattedTemperature(high)}°',
              style: TextStyle(
```

```
                fontSize: 16,
                fontWeight: FontWeight.w100,
                color: Colors.white,
              ),
            ),
            Text(
              'min: ${_formattedTemperature(low)}°',
              style: TextStyle(
                fontSize: 16,
                fontWeight: FontWeight.w100,
                color: Colors.white,
              ),
            )
          ],
        )
      ],
    );
  }

  int _formattedTemperature(double t) => t.round();
}
```

Make sure to export this in the `widgets.dart` file.

City Selection

The last thing we need to implement to have a functional app is out `CitySelection` widget which allows users to type in the name a city. Go ahead and create a `city_selection.dart` file inside the `widgets` folder.

> The `CitySelection` widget will allow users to input a city name and pass the selected city back to the `App` widget.

dart

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

class CitySelection extends StatefulWidget {
  @override
  State<CitySelection> createState() => _CitySelectionState();
}
```

```
class _CitySelectionState extends State<CitySelection> {
  final TextEditingController _textController = TextEditingController();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('City'),
      ),
      body: Form(
        child: Row(
          children: [
            Expanded(
              child: Padding(
                padding: EdgeInsets.only(left: 10.0),
                child: TextFormField(
                  controller: _textController,
                  decoration: InputDecoration(
                    labelText: 'City',
                    hintText: 'Chicago',
                  ),
                ),
              ),
            ),
            IconButton(
              icon: Icon(Icons.search),
              onPressed: () {
                Navigator.pop(context, _textController.text);
              },
            )
          ],
        ),
      ),
    );
  }
}
```

It needs to be a `StatefulWidget` because it has to maintain a `TextController`.

Note: When we press the search button we use `Navigator.pop` and pass the current text from our `TextController` back to the previous view.

Make sure to export this in the `widgets.dart` file.

Run the App

Now that we have created all our widgets, let's go back to the `main.dart` file. You'll see we need to import our `Weather` widget, so go ahead and add this line up top.

```dart
import 'package:flutter_weather/widgets/widgets.dart';
```

Then you can go ahead and run the app with `flutter run` in the terminal. Go ahead and select a city and you'll notice it has a few problems:

- The background is white and so is the text making it very hard to read
- We have no way to refresh the weather data after it is fetched
- The UI is very plain
- Everything is in Celsius and we have no way to change the units

Let's address these problems and take our Weather App to the next level!

Pull-To-Refresh

> In order to support pull-to-refresh we will need to update our `WeatherEvent` to handle a second event: `WeatherRefreshRequested`. Go ahead and add the following code to `blocs/weather_event.dart`

```dart
class WeatherRefreshRequested extends WeatherEvent {
  final String city;

  const WeatherRefreshRequested({@required this.city}) : assert(city != n
```

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

Next, we need to update our `mapEventToState` inside of `weather_bloc.dart` to handle a `WeatherRefreshRequested` event. Go ahead and add this `if` statement below the existing one.

```dart
if (event is WeatherRefreshRequested) {
  try {
    final Weather weather = await weatherRepository.getWeather(event.city
    yield WeatherLoadSuccess(weather: weather);
  } catch (_) {}
}
```

Here we are just creating a new event that will ask our weatherRepository to make an API call to get the weather for the city.

We can refactor `mapEventToState` to use some private helper functions in order to keep the code organized and easy to follow:

```dart
@override
Stream<WeatherState> mapEventToState(WeatherEvent event) async* {
  if (event is WeatherRequested) {
    yield* _mapWeatherRequestedToState(event);
  } else if (event is WeatherRefreshRequested) {
    yield* _mapWeatherRefreshRequestedToState(event);
  }
}

Stream<WeatherState> _mapWeatherRequestedToState(WeatherRequested event) 
  yield WeatherLoadInProgress();
  try {
    final Weather weather = await weatherRepository.getWeather(event.city
    yield WeatherLoadSuccess(weather: weather);
  } catch (_) {
    yield WeatherLoadFailure();
```

```
    }
  }

  Stream<WeatherState> _mapWeatherRefreshRequestedToState(WeatherRefreshReq
    try {
      final Weather weather = await weatherRepository.getWeather(event.city
      yield WeatherLoadSuccess(weather: weather);
    } catch (_) {}
  }
```

Lastly, we need to update our presentation layer to use a `RefreshIndicator` widget. Let's go ahead and modify our `Weather` widget in `widgets/weather.dart`. There are a few things we need to do.

- Import `async` to the `weather.dart` file to handle `Future`

```dart
import 'dart:async';
```

- Add a Completer

```dart
class Weather extends StatefulWidget {
  State<Weather> createState() => _WeatherState();
}

class _WeatherState extends State<Weather> {
  Completer<void> _refreshCompleter;

  @override
  void initState() {
    super.initState();
    _refreshCompleter = Completer<void>();
  }

  @override
  Widget build(BuildContext) {
    ...
  }
```

Since our `Weather` widget will need to maintain an instance of a `Completer`, we need to refactor it to be a `StatefulWidget`. Then, we can initialize the `Completer` in `initState`.

- Inside the widgets `build` method, let's wrap the `ListView` in a `RefreshIndicator` widget like so. Then return the `_refreshCompleter.future;` when the `onRefresh` callback happens.

```dart
return RefreshIndicator(
  onRefresh: () {
    BlocProvider.of<WeatherBloc>(context).add(
      WeatherRefreshRequested(city: state.weather.location),
    );
    return _refreshCompleter.future;
  },
  child: ListView(
    children: <Widget>[
      Padding(
        padding: EdgeInsets.only(top: 100.0),
        child: Center(
          child: Location(location: weather.location),
        ),
      ),
      Center(
        child: LastUpdated(dateTime: weather.lastUpdated),
      ),
      Padding(
        padding: EdgeInsets.symmetric(vertical: 50.0),
        child: Center(
          child: CombinedWeatherTemperature(
            weather: weather,
          ),
        ),
      ),
    ],
  ),
);
```

In order to use the `RefreshIndicator` we had to create a `Completer` which allows us to produce a `Future` which we can complete at a later time.

The last thing we need to do is complete the `Completer` when we receive a `WeatherLoadSuccess` state in order to dismiss the loading indicator once the weather has been updated.

```dart
class _WeatherState extends State<Weather> {
  ...
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: ...
      body: Center(
        child: BlocConsumer<WeatherBloc, WeatherState>(
          listener: (context, state) {
            if (state is WeatherLoadSuccess) {
              _refreshCompleter?.complete();
              _refreshCompleter = Completer();
            }
          },
          builder: (context, state) {
            ...
          }
        ),
      )
    )
  }
}
```

We converted our `BlocBuilder` into a `BlocConsumer` because we need to handle both rebuilding the UI based on state changes as well as performing side-effects (completing the `Completer`).

Note: `BlocConsumer` is identical to having a nested `BlocBuilder` within a `BlocListener`.

That's it! We now have solved problem #1 and users can refresh the weather by pulling down. Feel free to run `flutter run` again and try refreshing the weather.

Next, let's tackle the plain looking UI by creating a `ThemeBloc`.

Dynamic Theming

> Our `ThemeBloc` is going to be responsible for converting `ThemeEvents` into `ThemeStates`.

Our `ThemeEvents` are going to consist of a single event called `WeatherChanged` which will be added whenever the weather conditions we are displaying have changed.

```dart
abstract class ThemeEvent extends Equatable {
  const ThemeEvent();
}

class WeatherChanged extends ThemeEvent {
  final WeatherCondition condition;

  const WeatherChanged({@required this.condition}) : assert(condition !=

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

Our `ThemeState` will consist of a `ThemeData` and a `MaterialColor` which we will use to enhance our UI.

```dart
class ThemeState extends Equatable {
  final ThemeData theme;
  final MaterialColor color;

  const ThemeState({@required this.theme, @required this.color})
      : assert(theme != null),
        assert(color != null);

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

Now, we can implement our `ThemeBloc` which should look like:

```dart
class ThemeBloc extends Bloc<ThemeEvent, ThemeState> {
  ThemeBloc()
      : super(ThemeState(theme: ThemeData.light(), color: Colors.lightBlu

  @override
  Stream<ThemeState> mapEventToState(ThemeEvent event) async* {
    if (event is WeatherChanged) {
      yield _mapWeatherConditionToThemeData(event.condition);
    }
  }

  ThemeState _mapWeatherConditionToThemeData(WeatherCondition condition)
    ThemeState theme;
    switch (condition) {
      case WeatherCondition.clear:
      case WeatherCondition.lightCloud:
        theme = ThemeState(
          theme: ThemeData(
            primaryColor: Colors.orangeAccent,
          ),
          color: Colors.yellow,
        );
        break;
      case WeatherCondition.hail:
      case WeatherCondition.snow:
      case WeatherCondition.sleet:
        theme = ThemeState(
          theme: ThemeData(
            primaryColor: Colors.lightBlueAccent,
          ),
          color: Colors.lightBlue,
        );
        break;
      case WeatherCondition.heavyCloud:
        theme = ThemeState(
          theme: ThemeData(
            primaryColor: Colors.blueGrey,
          ),
          color: Colors.grey,
        );
```

```
          break;
        case WeatherCondition.heavyRain:
        case WeatherCondition.lightRain:
        case WeatherCondition.showers:
          theme = ThemeState(
            theme: ThemeData(
              primaryColor: Colors.indigoAccent,
            ),
            color: Colors.indigo,
          );
          break;
        case WeatherCondition.thunderstorm:
          theme = ThemeState(
            theme: ThemeData(
              primaryColor: Colors.deepPurpleAccent,
            ),
            color: Colors.deepPurple,
          );
          break;
        case WeatherCondition.unknown:
          theme = ThemeState(
            theme: ThemeData.light(),
            color: Colors.lightBlue,
          );
          break;
      }
      return theme;
    }
  }
```

Even though it's a lot of code, the only thing in here is logic to convert a `WeatherCondition` to a new `ThemeState`.

We can now update our `main` a `ThemeBloc` provide it to our `App`.

```dart
void main() {
  final WeatherRepository weatherRepository = WeatherRepository(
    weatherApiClient: WeatherApiClient(
      httpClient: http.Client(),
    ),
```

```
  );
  Bloc.observer = SimpleBlocObserver();
  runApp(
    BlocProvider<ThemeBloc>(
      create: (context) => ThemeBloc(),
      child: App(weatherRepository: weatherRepository),
    ),
  );
}
```

Our `App` widget can then use `BlocBuilder` to react to changes in `ThemeState`.

dart

```
class App extends StatelessWidget {
  final WeatherRepository weatherRepository;

  App({Key key, @required this.weatherRepository})
      : assert(weatherRepository != null),
        super(key: key);

  @override
  Widget build(BuildContext context) {
    return BlocBuilder<ThemeBloc, ThemeState>(
      builder: (context, themeState) {
        return MaterialApp(
          title: 'Flutter Weather',
          theme: themeState.theme,
          home: BlocProvider(
            create: (context) =>
                WeatherBloc(weatherRepository: weatherRepository),
            child: Weather(),
          ),
        );
      },
    );
  }
}
```

Note: We are using `BlocProvider` to make our `ThemeBloc` globally available using `BlocProvider.of<ThemeBloc>(context)`.

The last thing we need to do is create a cool `GradientContainer` widget which will color our background with respect to the current weather conditions.

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

import 'package:meta/meta.dart';

class GradientContainer extends StatelessWidget {
  final Widget child;
  final MaterialColor color;

  const GradientContainer({
    Key key,
    @required this.color,
    @required this.child,
  })  : assert(color != null, child != null),
        super(key: key);

  @override
  Widget build(BuildContext context) {
    return Container(
      decoration: BoxDecoration(
        gradient: LinearGradient(
          begin: Alignment.topCenter,
          end: Alignment.bottomCenter,
          stops: [0.6, 0.8, 1.0],
          colors: [
            color[700],
            color[500],
            color[300],
          ],
        ),
      ),
      child: child,
    );
  }
}
```

Now we can use our `GradientContainer` in our `Weather` widget like so:

```dart
import 'dart:async';

import 'package:flutter/material.dart';

import 'package:flutter_bloc/flutter_bloc.dart';

import 'package:flutter_weather/widgets/widgets.dart';
import 'package:flutter_weather/repositories/repositories.dart';
import 'package:flutter_weather/blocs/blocs.dart';

class Weather extends StatefulWidget {
  @override
  State<Weather> createState() => _WeatherState();
}

class _WeatherState extends State<Weather> {
  Completer<void> _refreshCompleter;

  @override
  void initState() {
    super.initState();
    _refreshCompleter = Completer<void>();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Flutter Weather'),
        actions: <Widget>[
          IconButton(
            icon: Icon(Icons.search),
            onPressed: () async {
              final city = await Navigator.push(
                context,
                MaterialPageRoute(
                  builder: (context) => CitySelection(),
                ),
              );
              if (city != null) {
                BlocProvider.of<WeatherBloc>(context)
                    .add(WeatherRequested(city: city));
```

```
            }
          },
        )
      ],
    ),
    body: Center(
      child: BlocConsumer<WeatherBloc, WeatherState>(
        listener: (context, state) {
          if (state is WeatherLoadSuccess) {
            BlocProvider.of<ThemeBloc>(context).add(
              WeatherChanged(condition: state.weather.condition),
            );
            _refreshCompleter?.complete();
            _refreshCompleter = Completer();
          }
        },
        builder: (context, state) {
          if (state is WeatherInitial) {
            return Center(child: Text('Please Select a Location'));
          }
          if (state is WeatherLoadInProgress) {
            return Center(child: CircularProgressIndicator());
          }
          if (state is WeatherLoadSuccess) {
            final weather = state.weather;

            return BlocBuilder<ThemeBloc, ThemeState>(
              builder: (context, themeState) {
                return GradientContainer(
                  color: themeState.color,
                  child: RefreshIndicator(
                    onRefresh: () {
                      BlocProvider.of<WeatherBloc>(context).add(
                        WeatherRefreshRequested(city: weather.location)
                      );
                      return _refreshCompleter.future;
                    },
                    child: ListView(
                      children: <Widget>[
                        Padding(
                          padding: EdgeInsets.only(top: 100.0),
                          child: Center(
                            child: Location(location: weather.location)
```

```
                          ),
                        ),
                        Center(
                          child: LastUpdated(dateTime: weather.lastUpda
                        ),
                        Padding(
                          padding: EdgeInsets.symmetric(vertical: 50.0)
                          child: Center(
                            child: CombinedWeatherTemperature(
                              weather: weather,
                            ),
                          ),
                        ),
                      ],
                    ),
                  ),
                );
              },
            );
          }
          if (state is WeatherLoadFailure) {
            return Text(
              'Something went wrong!',
              style: TextStyle(color: Colors.red),
            );
          }
        },
      ),
    ),
  );
 }
}
```

Since we want to "do something" in response to state changes in our `WeatherBloc`, we are using `BlocListener`. In this case, we are completing and resetting the `Completer` and are also adding the `WeatherChanged` event to the `ThemeBloc`.

> **Tip:** Check out the SnackBar Recipe for more information about the `BlocListener` widget.

We are accessing our `ThemeBloc` via `BlocProvider.of<ThemeBloc>(context)` and are then adding a `WeatherChanged` event on each `WeatherLoad`.

We also wrapped our `GradientContainer` widget with a `BlocBuilder` of `ThemeBloc` so that we can rebuild the `GradientContainer` and it's children in response to `ThemeState` changes.

Awesome! We now have an app that looks way nicer (in my opinion :P) and have tackled problem #2.

All that's left is to handle unit conversion between celsius and fahrenheit. To do that we'll create a `Settings` widget and a `SettingsBloc`.

Unit Conversion

We'll start off by creating our `SettingsBloc` which will convert `SettingsEvents` into `SettingsStates`.

Our `SettingsEvents` will consist of a single event: `TemperatureUnitsToggled`.

```dart
abstract class SettingsEvent extends Equatable {}

class TemperatureUnitsToggled extends SettingsEvent {
  @override
  List<Object> get props => [];
}
```

Our `SettingsState` will simply consist of the current `TemperatureUnits`.

```dart
enum TemperatureUnits { fahrenheit, celsius }

class SettingsState extends Equatable {
  final TemperatureUnits temperatureUnits;

  const SettingsState({@required this.temperatureUnits})
      : assert(temperatureUnits != null);

  @override
```

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

Lastly, we need to create our `SettingsBloc`:

```dart
class SettingsBloc extends Bloc<SettingsEvent, SettingsState> {
  SettingsBloc() : super(SettingsState(temperatureUnits: TemperatureUnits

  @override
  Stream<SettingsState> mapEventToState(SettingsEvent event) async* {
    if (event is TemperatureUnitsToggled) {
      yield SettingsState(
        temperatureUnits: state.temperatureUnits == TemperatureUnits.cels
            ? TemperatureUnits.fahrenheit
            : TemperatureUnits.celsius,
      );
    }
  }
}
```

All we're doing is using `fahrenheit` if `TemperatureUnitsToggled` is added and the current units are `celsius` and vice versa.

Now we need to provide our `SettingsBloc` to our `App` widget in `main.dart`.

```dart
void main() {
  final WeatherRepository weatherRepository = WeatherRepository(
    weatherApiClient: WeatherApiClient(
      httpClient: http.Client(),
    ),
  );
  Bloc.observer = SimpleBlocObserver();
  runApp(
    MultiBlocProvider(
      providers: [
        BlocProvider<ThemeBloc>(
          create: (context) => ThemeBloc(),
        ),
```

```
        BlocProvider<SettingsBloc>(
          create: (context) => SettingsBloc(),
        ),
      ],
      child: App(weatherRepository: weatherRepository),
    ),
  );
}
```

Again, we're making `SettingsBloc` globally accessible using `BlocProvider` and we are also closing it in the `close` callback. This time, however, since we are exposing more than one Bloc using `BlocProvider` at the same level we can eliminate some nesting by using the `MultiBlocProvider` widget.

Now we need to create our `Settings` widget from which users can toggle the units.

dart

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

import 'package:flutter_bloc/flutter_bloc.dart';

import 'package:flutter_weather/blocs/blocs.dart';

class Settings extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Settings')),
      body: ListView(
        children: <Widget>[
          BlocBuilder<SettingsBloc, SettingsState>(
              builder: (context, state) {
                return ListTile(
                  title: Text(
                    'Temperature Units',
                  ),
                  isThreeLine: true,
                  subtitle:
                      Text('Use metric measurements for temperature units
                  trailing: Switch(
                    value: state.temperatureUnits == TemperatureUnits.cel
```

```
                    onChanged: (_) => BlocProvider.of<SettingsBloc>(conte
                        .add(TemperatureUnitsToggled()),
                  ),
                );
              }),
        ],
      ),
    );
  }
}
```

We're using `BlocProvider` to access the `SettingsBloc` via the `BuildContext` and then using `BlocBuilder` to rebuild our UI based on `SettingsState` changed.

Our UI consists of a `ListView` with a single `ListTile` which contains a `Switch` that users can toggle to select celsius vs. fahrenheit.

> **Note:** In the switch's `onChanged` method we add a `TemperatureUnitsToggled` event to notify the `SettingsBloc` that the temperature units have changed.

Next, we need to allow users to get to the `Settings` widget from our `Weather` widget.

We can do that by adding a new `IconButton` in our `AppBar`.

dart

```dart
import 'dart:async';

import 'package:flutter/material.dart';

import 'package:flutter_bloc/flutter_bloc.dart';

import 'package:flutter_weather/widgets/widgets.dart';
import 'package:flutter_weather/repositories/repositories.dart';
import 'package:flutter_weather/blocs/blocs.dart';

class Weather extends StatefulWidget {
  @override
  State<Weather> createState() => _WeatherState();
}
```

```dart
class _WeatherState extends State<Weather> {
  Completer<void> _refreshCompleter;

  @override
  void initState() {
    super.initState();
    _refreshCompleter = Completer<void>();
  }

  @override
  Widget build(BuildContext context) {
    final weatherBloc = BlocProvider.of<WeatherBloc>(context);
    return Scaffold(
      appBar: AppBar(
        title: Text('Flutter Weather'),
        actions: <Widget>[
          IconButton(
            icon: Icon(Icons.settings),
            onPressed: () {
              Navigator.push(
                context,
                MaterialPageRoute(
                  builder: (context) => Settings(),
                ),
              );
            },
          ),
          IconButton(
            icon: Icon(Icons.search),
            onPressed: () async {
              final city = await Navigator.push(
                context,
                MaterialPageRoute(
                  builder: (context) => CitySelection(),
                ),
              );
              if (city != null) {
                BlocProvider.of<WeatherBloc>(context)
                    .add(WeatherRequested(city: city));
              }
            },
          )
```

```
        ],
      ),
      body: Center(
        child: BlocConsumer<WeatherBloc, WeatherState>(
          listener: (context, state) {
            if (state is WeatherLoadSuccess) {
              BlocProvider.of<ThemeBloc>(context).add(
                WeatherChanged(condition: state.weather.condition),
              );
              _refreshCompleter?.complete();
              _refreshCompleter = Completer();
            }
          },
          builder: (context, state) {
            if (state is WeatherInitial) {
              return Center(child: Text('Please Select a Location'));
            }
            if (state is WeatherLoadInProgress) {
              return Center(child: CircularProgressIndicator());
            }
            if (state is WeatherLoadSuccess) {
              final weather = state.weather;

              return BlocBuilder<ThemeBloc, ThemeState>(
                builder: (context, themeState) {
                  return GradientContainer(
                    color: themeState.color,
                    child: RefreshIndicator(
                      onRefresh: () {
                        BlocProvider.of<WeatherBloc>(context).add(
                          WeatherRefreshRequested(city: weather.location)
                        );
                        return _refreshCompleter.future;
                      },
                      child: ListView(
                        children: <Widget>[
                          Padding(
                            padding: EdgeInsets.only(top: 100.0),
                            child: Center(
                              child: Location(location: weather.location)
                            ),
                          ),
                          Center(
```

```
                            child: LastUpdated(dateTime: weather.lastUpda
                          ),
                          Padding(
                            padding: EdgeInsets.symmetric(vertical: 50.0)
                            child: Center(
                              child: CombinedWeatherTemperature(
                                weather: weather,
                              ),
                            ),
                          ),
                        ],
                      ),
                    ),
                  );
                },
              );
            }
            if (state is WeatherLoadFailure) {
              return Text(
                'Something went wrong!',
                style: TextStyle(color: Colors.red),
              );
            }
          },
        ),
      ),
    );
  }
}
```

We're almost done! We just need to update our `Temperature` widget to respond to the current units.

dart

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

import 'package:flutter_weather/blocs/blocs.dart';

class Temperature extends StatelessWidget {
  final double temperature;
  final double low;
```

```
  final double high;
  final TemperatureUnits units;

  Temperature({
    Key key,
    this.temperature,
    this.low,
    this.high,
    this.units,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Row(
      children: [
        Padding(
          padding: EdgeInsets.only(right: 20.0),
          child: Text(
            '${_formattedTemperature(temperature)}°',
            style: TextStyle(
              fontSize: 32,
              fontWeight: FontWeight.w600,
              color: Colors.white,
            ),
          ),
        ),
        Column(
          children: [
            Text(
              'max: ${_formattedTemperature(high)}°',
              style: TextStyle(
                fontSize: 16,
                fontWeight: FontWeight.w100,
                color: Colors.white,
              ),
            ),
            Text(
              'min: ${_formattedTemperature(low)}°',
              style: TextStyle(
                fontSize: 16,
                fontWeight: FontWeight.w100,
                color: Colors.white,
              ),
```

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

  int _toFahrenheit(double celsius) => ((celsius * 9 / 5) + 32).round();

  int _formattedTemperature(double t) =>
      units == TemperatureUnits.fahrenheit ? _toFahrenheit(t) : t.round()
}
```

And lastly, we need to inject the `TemperatureUnits` into the `Temperature` widget.

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

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

import 'package:flutter_weather/blocs/blocs.dart';
import 'package:flutter_weather/models/models.dart' as model;
import 'package:flutter_weather/widgets/widgets.dart';

class CombinedWeatherTemperature extends StatelessWidget {
  final model.Weather weather;

  CombinedWeatherTemperature({
    Key key,
    @required this.weather,
  })  : assert(weather != null),
        super(key: key);

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Row(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
```

```
              Padding(
                padding: EdgeInsets.all(20.0),
                child: WeatherConditions(condition: weather.condition),
              ),
              Padding(
                padding: EdgeInsets.all(20.0),
                child: BlocBuilder<SettingsBloc, SettingsState>(
                  builder: (context, state) {
                    return Temperature(
                      temperature: weather.temp,
                      high: weather.maxTemp,
                      low: weather.minTemp,
                      units: state.temperatureUnits,
                    );
                  },
                ),
              ),
            ],
          ),
          Center(
            child: Text(
              weather.formattedCondition,
              style: TextStyle(
                fontSize: 30,
                fontWeight: FontWeight.w200,
                color: Colors.white,
              ),
            ),
          ),
        ],
      );
    }
  }
```

That's all there is to it! We've now successfully implemented a weather app in flutter using the bloc and flutter_bloc packages and we've successfully separated our presentation layer from our business logic.

The full source for this example can be found here.

PREVIOUS
Login ✨

NEXT
Todos