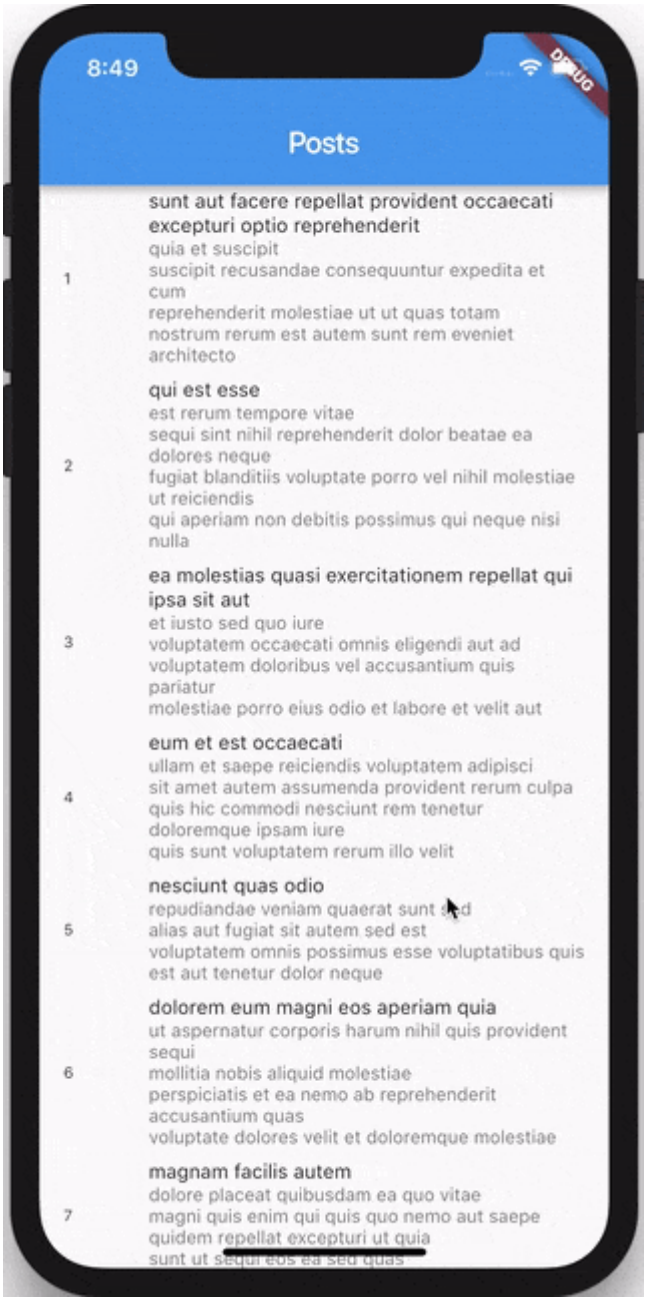
Flutter Infinite List Tutorial

level intermediate

> In this tutorial, we're going to be implementing an app which fetches data over the network and loads it as a user scrolls using Flutter and the bloc library.



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 Cubit instead of Bloc. What's the difference?
- Adding events with context.read.⚡
- Prevent unnecessary rebuilds with Equatable.
- Use the `transformEvents` method with Rx.

Setup

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

```bash
flutter create flutter_infinite_list
```

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

```yaml
name: flutter_infinite_list
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
  rxdart: ^0.23.1

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

and then install all of our dependencies

```bash
flutter packages get
```

REST API

For this demo application, we'll be using **jsonplaceholder** as our data source.

> jsonplaceholder is an online REST API which serves fake data; it's very useful for building prototypes.

Open a new tab in your browser and visit **https://jsonplaceholder.typicode.com/posts?_start=0&_limit=2** to see what the API returns.

```json
[
  {
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat provident occaecati excepturi opti
    "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita
  },
  {
    "userId": 1,
    "id": 2,
    "title": "qui est esse",
    "body": "est rerum tempore vitae\nsequi sint nihil reprehenderit dolo
  }
]
```

> **Note:** in our url we specified the start and limit as query parameters to the GET request.

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

Data Model

Create `post.dart` and let's get to work creating the model of our Post object.

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

class Post extends Equatable {
  final int id;
  final String title;
  final String body;

  const Post({this.id, this.title, this.body});

  @override
  List<Object> get props => [id, title, body];

  @override
  String toString() => 'Post { id: $id }';
}
```

`Post` is just a class with an `id`, `title`, and `body`.

> We override the `toString` function in order to have a custom string representation of our `Post` for later.

> We extend `Equatable` so that we can compare `Posts`; by default, the equality operator returns true if and only if this and other are the same instance.

Now that we have our `Post` object model, let's start working on the Business Logic Component (bloc).

Post Events

Before we dive into the implementation, we need to define what our `PostBloc` is going to be doing.

At a high level, it will be responding to user input (scrolling) and fetching more posts in order for the presentation layer to display them. Let's start by creating our `Event`.

Our `PostBloc` will only be responding to a single event; `PostFetched` which will be added by the presentation layer whenever it needs more Posts to present. Since our `PostFetched` event is a type of `PostEvent` we can create `bloc/post_event.dart` and implement the event like so.

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

abstract class PostEvent extends Equatable {
  @override
  List<Object> get props => [];
}

class PostFetched extends PostEvent {}
```

To recap, our `PostBloc` will be receiving `PostEvents` and converting them to `PostStates`. We have defined all of our `PostEvents` (PostFetched) so next let's define our `PostState`.

Post States

Our presentation layer will need to have several pieces of information in order to properly lay itself out:

- `PostInitial` - will tell the presentation layer it needs to render a loading indicator while the initial batch of posts are loaded
- `PostSuccess` - will tell the presentation layer it has content to render
    - `posts` - will be the `List<Post>` which will be displayed

    - `hasReachedMax` - will tell the presentation layer whether or not it has reached the maximum number of posts

- `PostFailure` - will tell the presentation layer that an error has occurred while fetching posts

We can now create `bloc/post_state.dart` and implement it like so.

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

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

abstract class PostState extends Equatable {
  const PostState();

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

class PostInitial extends PostState {}

class PostFailure extends PostState {}

class PostSuccess extends PostState {
  final List<Post> posts;
  final bool hasReachedMax;

  const PostSuccess({
    this.posts,
    this.hasReachedMax,
  });

  PostSuccess copyWith({
    List<Post> posts,
    bool hasReachedMax,
  }) {
    return PostSuccess(
      posts: posts ?? this.posts,
      hasReachedMax: hasReachedMax ?? this.hasReachedMax,
    );
  }
```

```
  @override
  List<Object> get props => [posts, hasReachedMax];

  @override
  String toString() =>
      'PostSuccess { posts: ${posts.length}, hasReachedMax: $hasReachedMa
}
```

We implemented `copyWith` so that we can copy an instance of `PostSuccess` and update zero or more properties conveniently (this will come in handy later).

Now that we have our `Events` and `States` implemented, we can create our `PostBloc` .

To make it convenient to import our states and events with a single import we can create `bloc/bloc.dart` which exports them all (we'll add our `post_bloc.dart` export in the next section).

dart
```
export './post_event.dart';
export './post_state.dart';
```

Post Bloc

For simplicity, our `PostBloc` will have a direct dependency on an `http client` ; however, in a production application you might want instead inject an api client and use the repository pattern docs.

Let's create `post_bloc.dart` and create our empty `PostBloc` .

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

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

```dart
import 'package:flutter_infinite_list/post.dart';

class PostBloc extends Bloc<PostEvent, PostState> {
  final http.Client httpClient;

  PostBloc({@required this.httpClient}) : super(PostInitial());

  @override
  Stream<PostState> mapEventToState(PostEvent event) async* {
    // TODO: implement mapEventToState
    yield null;
  }
}
```

Note: just from the class declaration we can tell that our PostBloc will be taking PostEvents as input and outputting PostStates.

Next, we need to implement `mapEventToState` which will be fired every time a `PostEvent` is added.

```dart
@override
Stream<PostState> mapEventToState(PostEvent event) async* {
  final currentState = state;
  if (event is PostFetched && !_hasReachedMax(currentState)) {
    try {
      if (currentState is PostInitial) {
        final posts = await _fetchPosts(0, 20);
        yield PostSuccess(posts: posts, hasReachedMax: false);
        return;
      }
      if (currentState is PostSuccess) {
        final posts =
            await _fetchPosts(currentState.posts.length, 20);
        yield posts.isEmpty
            ? currentState.copyWith(hasReachedMax: true)
            : PostSuccess(
                posts: currentState.posts + posts,
                hasReachedMax: false,
              );
      }
```

```
      } catch (_) {
        yield PostFailure();
      }
    }
  }

  bool _hasReachedMax(PostState state) =>
      state is PostSuccess && state.hasReachedMax;

  Future<List<Post>> _fetchPosts(int startIndex, int limit) async {
    final response = await httpClient.get(
        'https://jsonplaceholder.typicode.com/posts?_start=$startIndex&_lim
    if (response.statusCode == 200) {
      final data = json.decode(response.body) as List;
      return data.map((rawPost) {
        return Post(
          id: rawPost['id'],
          title: rawPost['title'],
          body: rawPost['body'],
        );
      }).toList();
    } else {
      throw Exception('error fetching posts');
    }
  }
```

Our `PostBloc` will `yield` whenever there is a new state because it returns a `Stream<PostState>`. Check out core concepts for more information about `Streams` and other core concepts.

Now every time a `PostEvent` is added, if it is a `PostFetched` event and there are more posts to fetch, our `PostBloc` will fetch the next 20 posts.

The API will return an empty array if we try to fetch beyond the maximum number of posts (100), so if we get back an empty array, our bloc will `yield` the currentState except we will set `hasReachedMax` to true.

If we cannot retrieve the posts, we throw an exception and `yield` `PostFailure()`.

If we can retrieve the posts, we return `PostSuccess()` which takes the entire list of posts.

One optimization we can make is to `debounce` the `Events` in order to prevent spamming our API unnecessarily. We can do this by overriding the `transform` method in our `PostBloc`.

> **Note:** Overriding transform allows us to transform the Stream before mapEventToState is called. This allows for operations like distinct(), debounceTime(), etc... to be applied.

dart

```
@override
Stream<Transition<PostEvent, PostState>> transformEvents(
  Stream<PostEvent> events,
  TransitionFunction<PostEvent, PostState> transitionFn,
) {
  return super.transformEvents(
    events.debounceTime(const Duration(milliseconds: 500)),
    transitionFn,
  );
}
```

Our finished `PostBloc` should now look like this:

dart

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

import 'package:meta/meta.dart';
import 'package:rxdart/rxdart.dart';
import 'package:http/http.dart' as http;
import 'package:bloc/bloc.dart';
import 'package:flutter_infinite_list/bloc/bloc.dart';
import 'package:flutter_infinite_list/models/models.dart';

class PostBloc extends Bloc<PostEvent, PostState> {
  final http.Client httpClient;

  PostBloc({@required this.httpClient}) : super(PostInitial());

  @override
  Stream<Transition<PostEvent, PostState>> transformEvents(
```

```
    Stream<PostEvent> events,
    TransitionFunction<PostEvent, PostState> transitionFn,
  ) {
    return super.transformEvents(
      events.debounceTime(const Duration(milliseconds: 500)),
      transitionFn,
    );
  }

  @override
  Stream<PostState> mapEventToState(PostEvent event) async* {
    final currentState = state;
    if (event is PostFetched && !_hasReachedMax(currentState)) {
      try {
        if (currentState is PostInitial) {
          final posts = await _fetchPosts(0, 20);
          yield PostSuccess(posts: posts, hasReachedMax: false);
          return;
        }
        if (currentState is PostSuccess) {
          final posts = await _fetchPosts(currentState.posts.length, 20);
          yield posts.isEmpty
              ? currentState.copyWith(hasReachedMax: true)
              : PostSuccess(
                  posts: currentState.posts + posts,
                  hasReachedMax: false,
                );
        }
      } catch (_) {
        yield PostFailure();
      }
    }
  }

  bool _hasReachedMax(PostState state) =>
      state is PostSuccess && state.hasReachedMax;

  Future<List<Post>> _fetchPosts(int startIndex, int limit) async {
    final response = await httpClient.get(
        'https://jsonplaceholder.typicode.com/posts?_start=$startIndex&_l
    if (response.statusCode == 200) {
      final data = json.decode(response.body) as List;
      return data.map((rawPost) {
```

```
          return Post(
            id: rawPost['id'],
            title: rawPost['title'],
            body: rawPost['body'],
          );
        }).toList();
      } else {
        throw Exception('error fetching posts');
      }
    }
  }
```

Don't forget to update `bloc/bloc.dart` to include our `PostBloc`!

```dart
export './post_bloc.dart';
export './post_event.dart';
export './post_state.dart';
```

Great! Now that we've finished implementing the business logic all that's left to do is implement the presentation layer.

Presentation Layer

In our `main.dart` we can start by implementing our main function and calling `runApp` to render our root widget.

In our `App` widget, we use `BlocProvider` to create and provide an instance of `PostBloc` to the subtree. Also, we add a `PostFetched` event so that when the app loads, it requests the initial batch of Posts.

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

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

void main() {
```

```
  runApp(App());
}

class App extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Infinite Scroll',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Posts'),
        ),
        body: BlocProvider(
          create: (context) =>
              PostBloc(httpClient: http.Client())..add(PostFetched()),
          child: HomePage(),
        ),
      ),
    );
  }
}
```

Next, we need to implement our `HomePage` widget which will present our posts and hook up to our `PostBloc`.

```dart
class HomePage extends StatefulWidget {
  @override
  _HomePageState createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
  final _scrollController = ScrollController();
  final _scrollThreshold = 200.0;
  PostBloc _postBloc;

  @override
  void initState() {
    super.initState();
    _scrollController.addListener(_onScroll);
    _postBloc = BlocProvider.of<PostBloc>(context);
  }
```

```dart
@override
Widget build(BuildContext context) {
  return BlocBuilder<PostBloc, PostState>(
    builder: (context, state) {
      if (state is PostInitial) {
        return Center(
          child: CircularProgressIndicator(),
        );
      }
      if (state is PostFailure) {
        return Center(
          child: Text('failed to fetch posts'),
        );
      }
      if (state is PostSuccess) {
        if (state.posts.isEmpty) {
          return Center(
            child: Text('no posts'),
          );
        }
        return ListView.builder(
          itemBuilder: (BuildContext context, int index) {
            return index >= state.posts.length
                ? BottomLoader()
                : PostWidget(post: state.posts[index]);
          },
          itemCount: state.hasReachedMax
              ? state.posts.length
              : state.posts.length + 1,
          controller: _scrollController,
        );
      }
    },
  );
}

@override
void dispose() {
  _scrollController.dispose();
  super.dispose();
}
```

```
  void _onScroll() {
    final maxScroll = _scrollController.position.maxScrollExtent;
    final currentScroll = _scrollController.position.pixels;
    if (maxScroll - currentScroll <= _scrollThreshold) {
      _postBloc.add(PostFetched());
    }
  }
}
```

`HomePage` is a `StatefulWidget` because it will need to maintain a `ScrollController`. In `initState`, we add a listener to our `ScrollController` so that we can respond to scroll events. We also access our `PostBloc` instance via `BlocProvider.of<PostBloc>(context)`.

Moving along, our build method returns a `BlocBuilder`. `BlocBuilder` is a Flutter widget from the flutter_bloc package which handles building a widget in response to new bloc states. Any time our `PostBloc` state changes, our builder function will be called with the new `PostState`.

> We need to remember to clean up after ourselves and dispose of our `ScrollController` when the StatefulWidget is disposed.

Whenever the user scrolls, we calculate how far away from the bottom of the page they are and if the distance is ≤ our `_scrollThreshold` we add a `PostFetched` event in order to load more posts.

Next, we need to implement our `BottomLoader` widget which will indicate to the user that we are loading more posts.

```dart
class BottomLoader extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      alignment: Alignment.center,
      child: Center(
        child: SizedBox(
          width: 33,
```

```
            height: 33,
            child: CircularProgressIndicator(
              strokeWidth: 1.5,
            ),
          ),
        ),
      );
    }
  }
```

Lastly, we need to implement our `PostWidget` which will render an individual Post.

```dart
class PostWidget extends StatelessWidget {
  final Post post;

  const PostWidget({Key key, @required this.post}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return ListTile(
      leading: Text(
        '${post.id}',
        style: TextStyle(fontSize: 10.0),
      ),
      title: Text(post.title),
      isThreeLine: true,
      subtitle: Text(post.body),
      dense: true,
    );
  }
}
```

At this point, we should be able to run our app and everything should work; however, there's one more thing we can do.

One added bonus of using the bloc library is that we can have access to all `Transitions` in one place.

> The change from one state to another is called a `Transition`.

A `Transition` consists of the current state, the event, and the next state.

Even though in this application we only have one bloc, it's fairly common in larger applications to have many blocs managing different parts of the application's state.

If we want to be able to do something in response to all `Transitions` we can simply create our own `BlocObserver`.

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

class SimpleBlocObserver extends BlocObserver {
  @override
  void onTransition(Bloc bloc, Transition transition) {
    print(transition);
    super.onTransition(bloc, transition);
  }
}
```

All we need to do is extend `BlocObserver` and override the `onTransition` method.

In order to tell Bloc to use our `SimpleBlocObserver`, we just need to tweak our main function.

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

Now when we run our application, every time a Bloc `Transition` occurs we can see the transition printed to the console.

In practice, you can create different `BlocObservers` and because every state change is recorded, we are able to very easily instrument our applications and track all user interactions and state changes in one place!

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

Our `HomePage` has no idea where the `Posts` are coming from or how they are being retrieved. Conversely, our `PostBloc` has no idea how the `State` is being rendered, it simply converts events into states.

The full source for this example can be found here.

PREVIOUS
Timer

NEXT
Login ✨