Flutter Cookbook

This cookbook contains recipes that demonstrate how to solve common problems while writing Flutter apps. Each recipe is self-contained and can be used as reference to help you build up an application.

Version: 2018-03-14

Design basics

- Using Themes to share colors and font styles

Images

- Display images from the internet
- Fade in images with a placeholder
- Working with cached images

Lists

- Create a basic list
- Make a horizontal list
- Working with long lists
- Creating lists with different types of items
- Creating a grid List

Handling Gestures

- Handling Taps
- Adding Material Touch ripples
- Implement Swipe to Dismiss

Navigation

- Navigate to a new screen and back
- Send data to a new screen
- Return data from a screen

Networking

- Fetch data from the internet
- Making authenticated requests
- Working with WebSockets

Using Themes to share colors and font styles

In order to share colors and font styles throughout our app, we can take advantage of themes. There are two ways to define themes: App-wide or using `Theme` Widgets that define the colors and font styles for a particular part of our application. In fact, app-wide themes are just `Theme` Widgets created at the root of our apps by the `MaterialApp`!

After we define a Theme, we can use it within our own Widgets. In addition, the Material Widgets provided by Flutter will use our Theme to set the background colors and and font styles for AppBars, Buttons, Checkboxes, and more.

Creating an app theme

In order to share a Theme containing colors and font styles across our entire app, we can provide `ThemeData` to the `MaterialApp` constructor.

If no `theme` is provided, Flutter will create a fallback theme under the hood.

```
new MaterialApp(
  title: title,
  theme: new ThemeData(
    brightness: Brightness.dark,
    primaryColor: Colors.lightBlue[800],
    accentColor: Colors.cyan[600],
  ),
);
```

Please see the ThemeData documentation to see all of the colors and fonts you can define.

Themes for part of an application

If we want to override the app-wide theme in part of our application, we can wrap a section of our app in a `Theme` Widget.

There are two ways to approach this: creating unique `ThemeData`, or extending the parent theme.

Creating unique `ThemeData`

If we don't want to inherit any application colors or font styles, we can create a `new ThemeData()` instance and pass that to the `Theme` Widget.

```
new Theme(
  // Create a unique theme with "new ThemeData"
  data: new ThemeData(
    accentColor: Colors.yellow,
  ),
  child: new FloatingActionButton(
    onPressed: () {},
    child: new Icon(Icons.add),
  ),
);
```

Extending the parent theme

Rather than overriding everything, it often makes sense to extend the parent theme. We can achieve this by using the `copyWith` method.

```
new Theme(
  // Find and Extend the parent theme using "copyWith". Please see the next
  // section for more info on `Theme.of`.
  data: Theme.of(context).copyWith(accentColor: Colors.yellow),
  child: new FloatingActionButton(
    onPressed: null,
    child: new Icon(Icons.add),
  ),
);
```

Using a Theme

Now that we've defined a theme, we can use it within our Widget `build` methods by using the `Theme.of(context)` function!

`Theme.of(context)` will look up the Widget tree and return the nearest `Theme` in the tree. If we have a stand-alone `Theme` defined above our Widget, it returns that. If not, it returns the App theme.

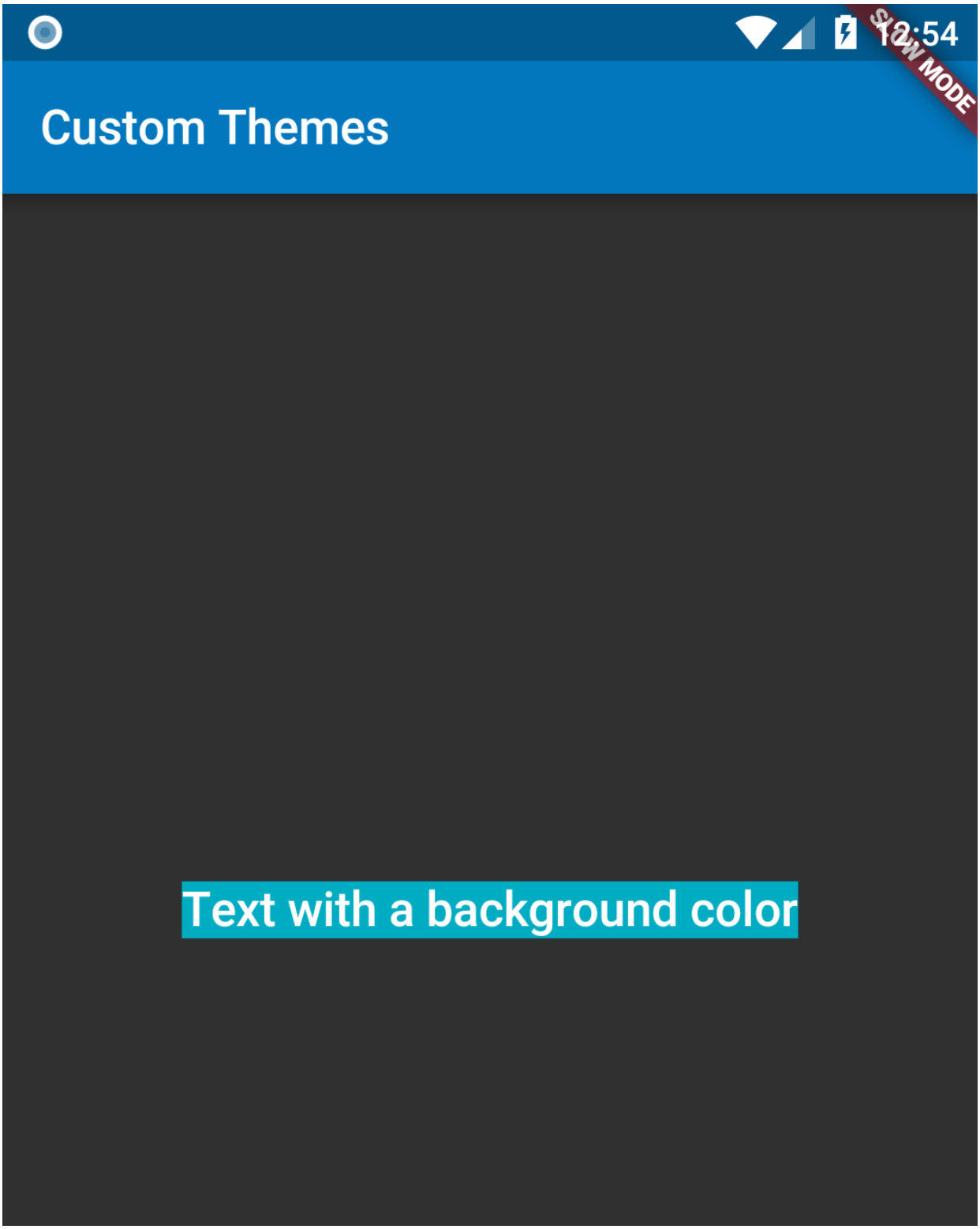
In fact, the `FloatingActionButton` uses this exact technique to find the `accentColor`!

```
new Container(
  color: Theme.of(context).accentColor,
  child: new Text(
    'Text with a background color',
    style: Theme.of(context).textTheme.title,
  ),
);
```

Complete Example

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final appName = 'Custom Themes';

    return new MaterialApp(
      title: appName,
      theme: new ThemeData(
        brightness: Brightness.dark,
        primaryColor: Colors.lightBlue[800],
        accentColor: Colors.cyan[600],
      ),
      home: new MyHomePage(
        title: appName,
      ),
    );
  }
}

class MyHomePage extends StatelessWidget {
  final String title;

  MyHomePage({Key key, @required this.title}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(title),
      ),
      body: new Center(
        child: new Container(
          color: Theme.of(context).accentColor,
          child: new Text(
            'Text with a background color',
            style: Theme.of(context).textTheme.title,
          ),
        ),
      ),
```

```
      floatingActionButton: new Theme(
        data: Theme.of(context).copyWith(accentColor: Colors.yellow),
        child: new FloatingActionButton(
          onPressed: null,
          child: new Icon(Icons.add),
        ),
      ),
    );
  }
}
```



Display images from the internet

Displaying images is fundamental for most mobile apps. Flutter provides the `Image` Widget to display different types of images.

In order to work with images from a URL, use the `Image.network` constructor.

```
new Image.network(
  'https://raw.githubusercontent.com/flutter/website/master/_includes/code/layout/lakes/images/lake.jpg',
)
```

Bonus: Animated Gifs

One amazing thing about the `Image` Widget: It also supports animated gifs out of the box!

```
new Image.network(
  'https://github.com/flutter/plugins/raw/master/packages/video_player/doc/demo_ipod.gif?raw=true',
);
```

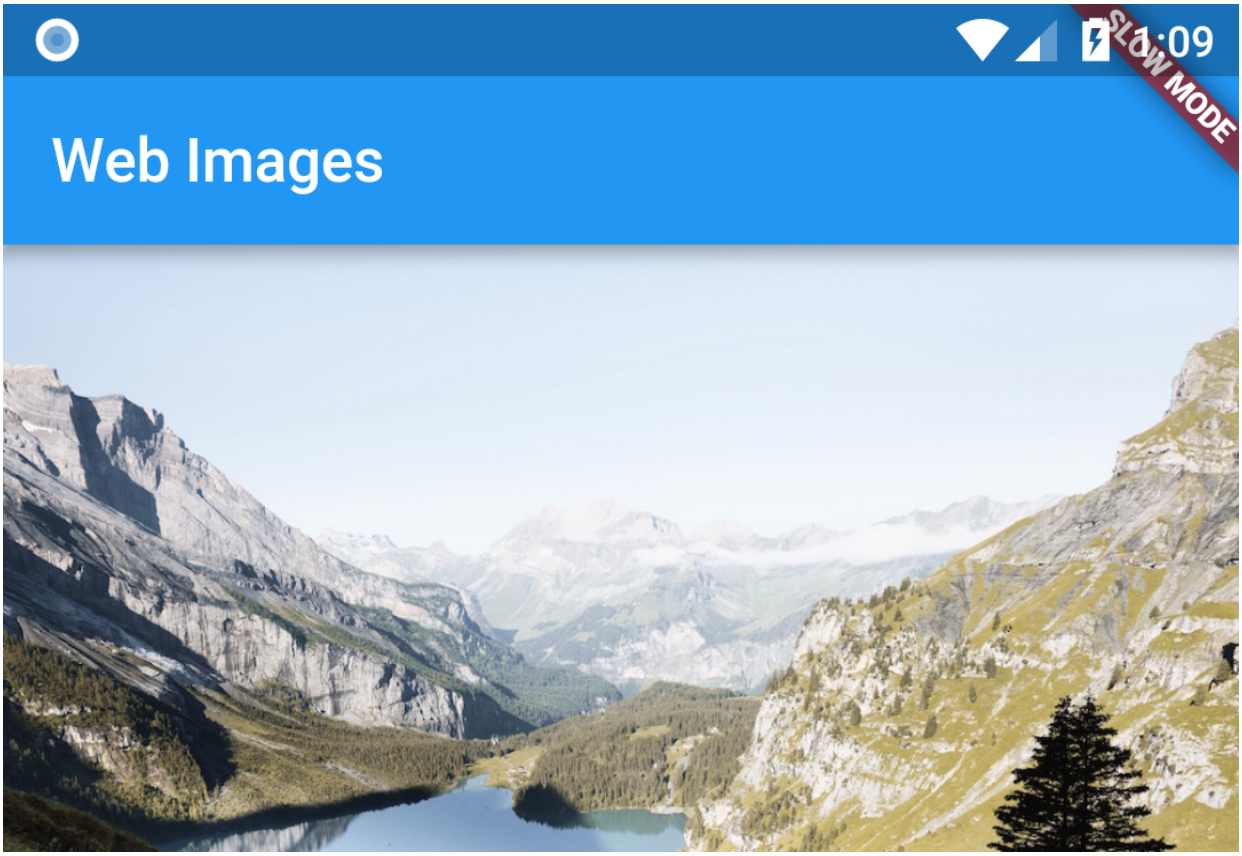
Placeholders and Caching

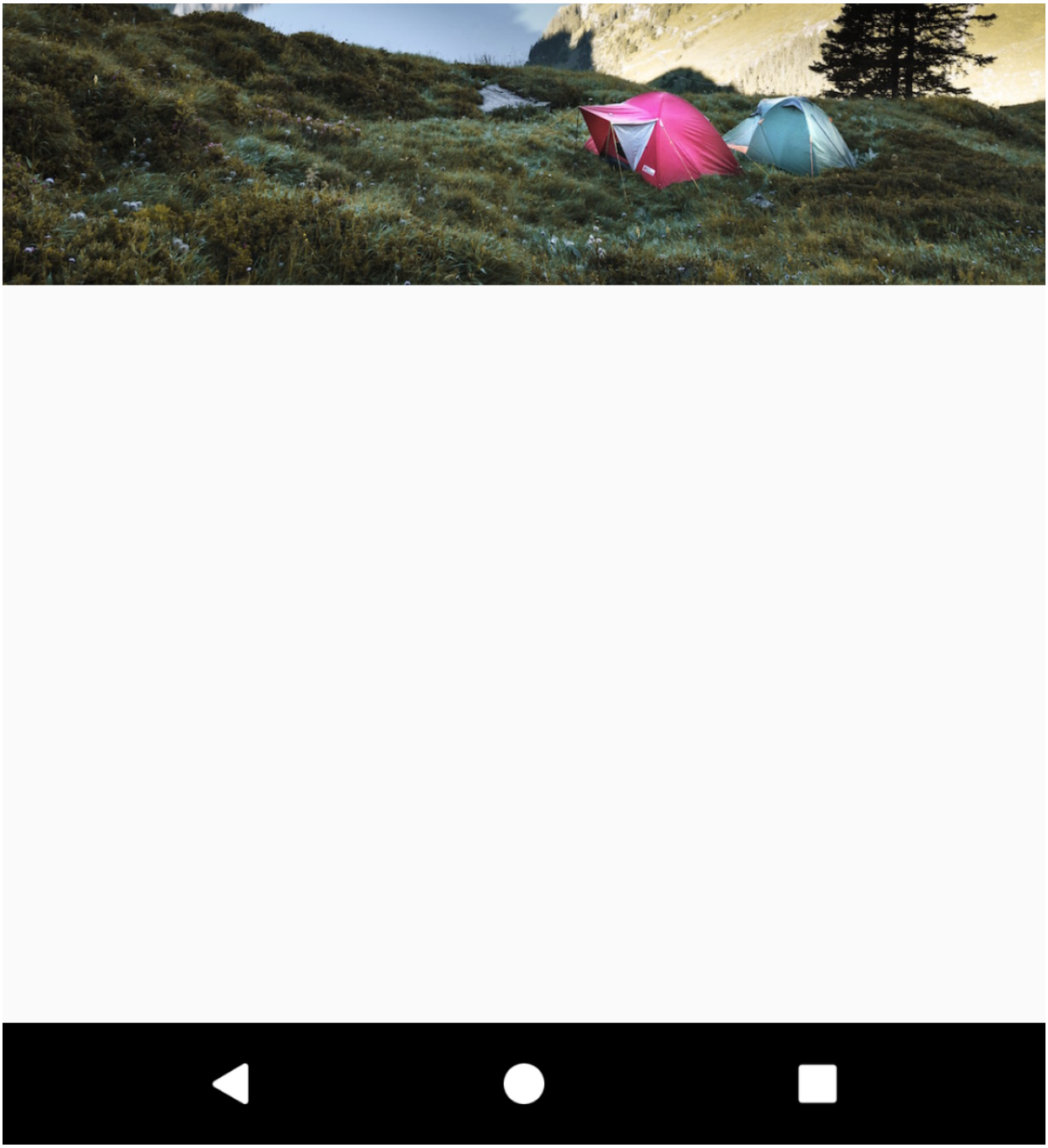
The default `Image.network` constructor does not handle more advanced functionality, such as fading images in after loading or caching images to the device after they're downloaded. To achieve these tasks, please see the following recipes:

- Fade in images with a placeholder
- Working with cached images

Complete Example

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

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    var title = 'Web Images';

    return new MaterialApp(
      title: title,
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text(title),
        ),
        body: new Image.network(

 'https://github.com/flutter/website/blob/master/_includes/code/layout/lakes/images/lake.jpg?raw=true',
        ),
      ),
    );
  }
}
```





Fade in images with a placeholder

When displaying images using the default `Image` widget, you might notice they simply pop onto the screen as they're loaded. This might feel visually jarring to your users.

Instead, wouldn't it be nice if you could display a placeholder at first, and images would fade in as they're loaded? We can use the `FadeInImage` Widget packaged with Flutter for exactly this purpose!

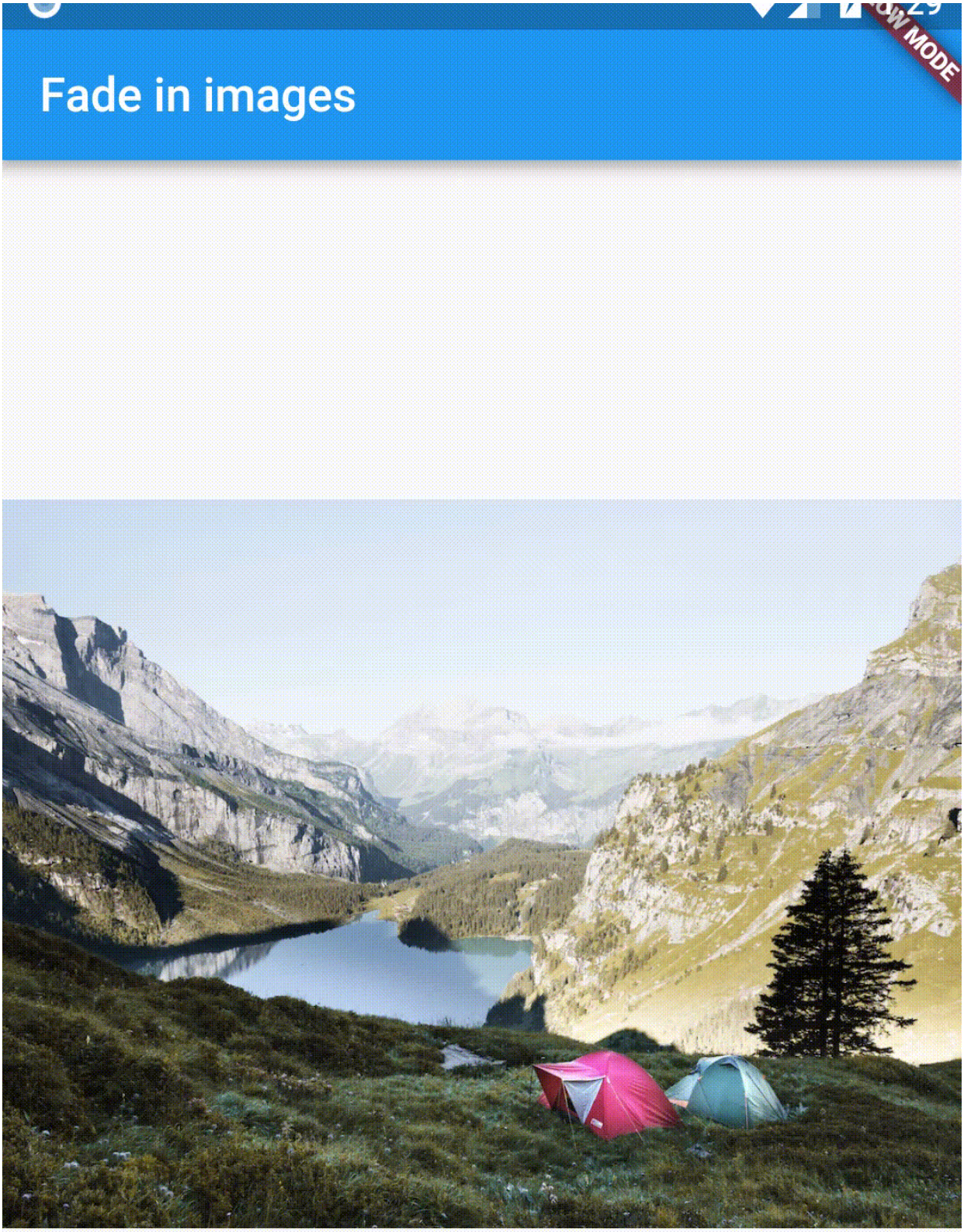
`FadeInImage` works with images of any type: in-memory, local assets, or images from the internet.

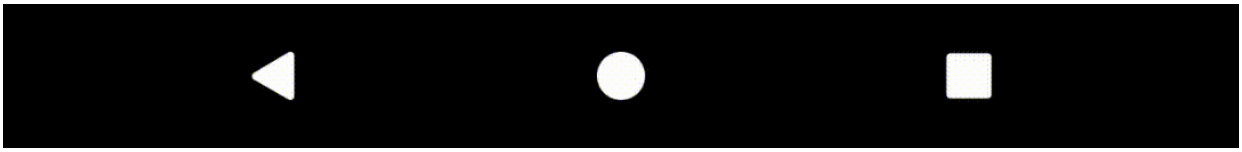
In this example, we'll use the transparent_image package for a simple transparent placeholder. You can also consider using local assets for placeholders by following the Assets and Images guide.

```
new FadeInImage.memoryNetwork(
  placeholder: kTransparentImage,
  image:
'https://github.com/flutter/website/blob/master/_includes/code/layout/lakes
/images/lake.jpg?raw=true',
);
```

Complete Example

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'Fade in images';

    return new MaterialApp(
      title: title,
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text(title),
        ),
        body: new Stack(
          children: <Widget>[
            new Center(child: new CircularProgressIndicator()),
            new Center(
              child: new FadeInImage.memoryNetwork(
                placeholder: kTransparentImage,
                image:

 'https://github.com/flutter/website/blob/master/_includes/code/layout/lake
s/images/lake.jpg?raw=true',
              ),
            ),
          ],
        ),
      ),
    );
  }
}
```


Fade in images
SW MODE



Working with cached images

In some cases, it can be handy to cache images as they're downloaded from the web so they can be used offline. For this purpose, we'll employ the `cached_network_image` package.

In addition to caching, the cached_image_network package also supports placeholders and fading images in as they're loaded!

```
new CachedNetworkImage(
  imageUrl:
'https://github.com/flutter/website/blob/master/_includes/code/layout/lakes
/images/lake.jpg?raw=true',
);
```

Adding a placeholder

The `cached_network_image` package allows us to use any Widget as a placeholder! In this example, we'll display a spinner while the image loads.

```
new CachedNetworkImage(
  placeholder: new CircularProgressIndicator(),
  imageUrl:
'https://github.com/flutter/website/blob/master/_includes/code/layout/lakes
/images/lake.jpg?raw=true',
);
```

Complete Example

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'Cached Images';

    return new MaterialApp(
```

```
      title: title,
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text(title),
        ),
        body: new Center(
          child: new CachedNetworkImage(
            placeholder: new CircularProgressIndicator(),
            imageUrl:

 'https://github.com/flutter/website/blob/master/_includes/code/layout/lakes/images/lake.jpg?raw=true',
          ),
        ),
      ),
    );
  }
}
```

Basic List

Displaying lists of data is a fundamental pattern for mobile apps. Flutter includes the `ListView` Widget to make working with lists a breeze!

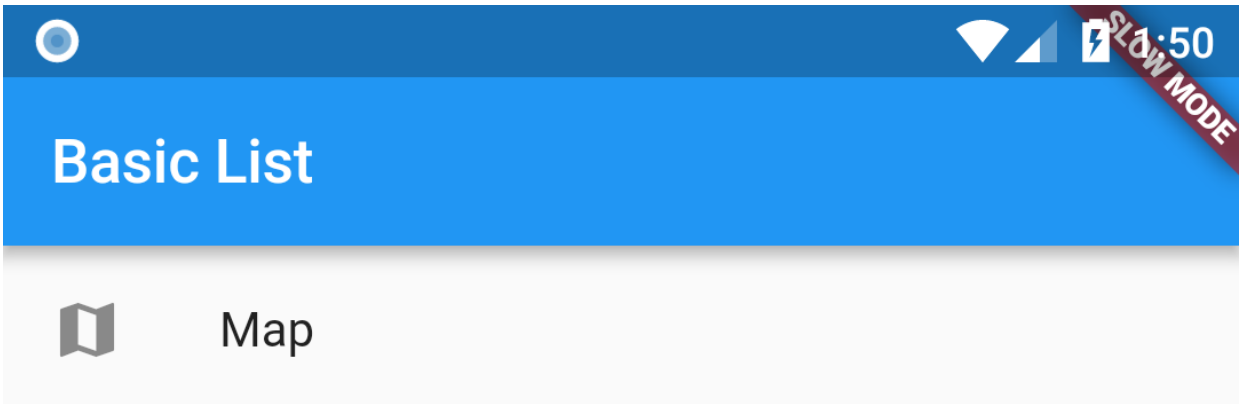
Create a ListView

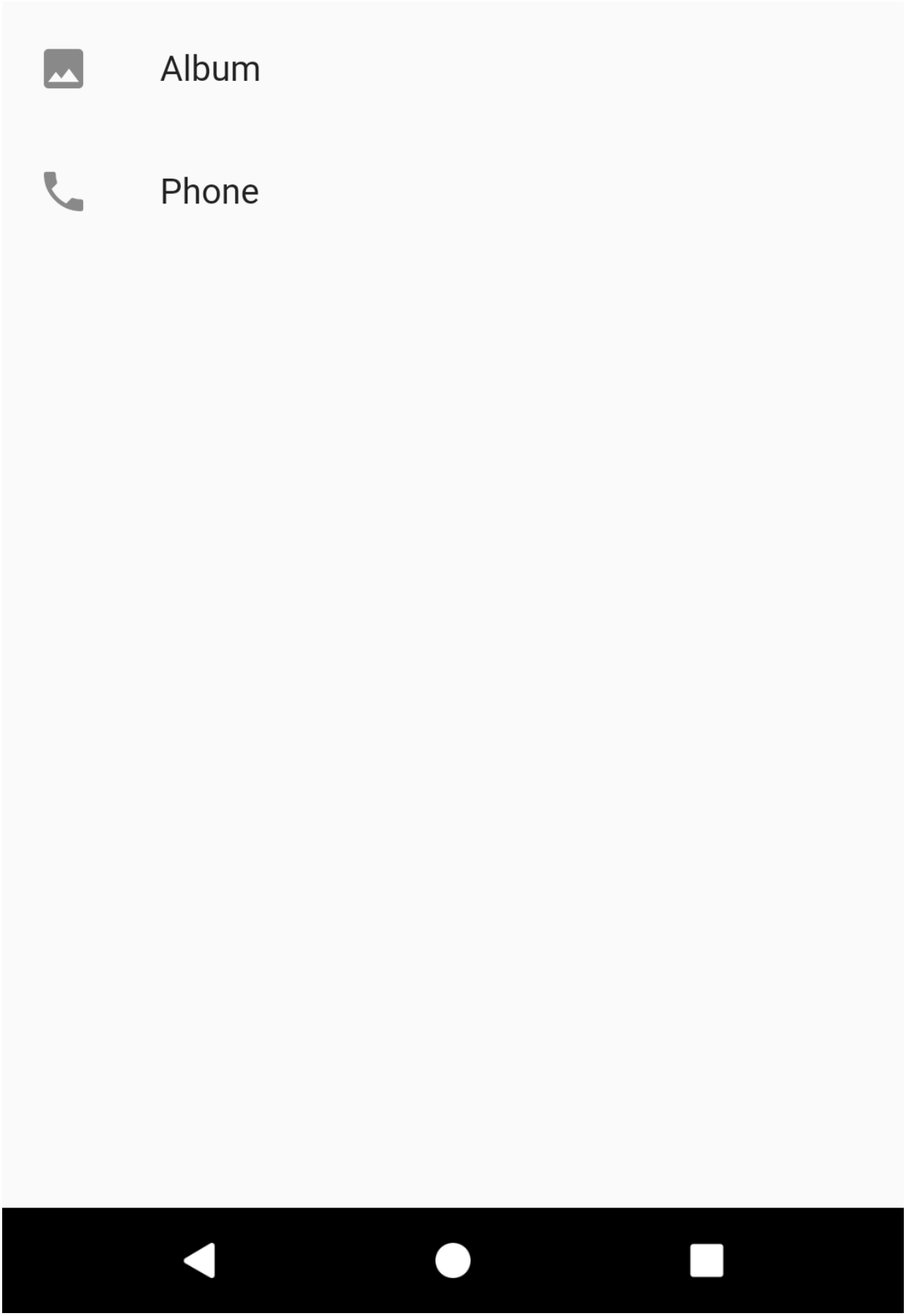
Using the standard `ListView` constructor is perfect for lists that contain only a few items. We will also employ the built-in `ListTile` Widget to give our items a visual structure.

```
new ListView(
  children: <Widget>[
    new ListTile(
      leading: new Icon(Icons.map),
      title: new Text('Maps'),
    ),
    new ListTile(
      leading: new Icon(Icons.photo_album),
      title: new Text('Album'),
    ),
    new ListTile(
      leading: new Icon(Icons.phone),
      title: new Text('Phone'),
    ),
  ],
);
```

Complete Example

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

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'Basic List';

    return new MaterialApp(
      title: title,
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text(title),
        ),
        body: new ListView(
          children: <Widget>[
            new ListTile(
              leading: new Icon(Icons.map),
              title: new Text('Map'),
            ),
            new ListTile(
              leading: new Icon(Icons.photo),
              title: new Text('Album'),
            ),
            new ListTile(
              leading: new Icon(Icons.phone),
              title: new Text('Phone'),
            ),
          ],
        ),
      ),
    );
  }
}
```





Create a horizontal list

At times, you may want to create a List that scrolls horizontally rather than vertically. The `ListView` Widget supports horizontal lists out of the box.

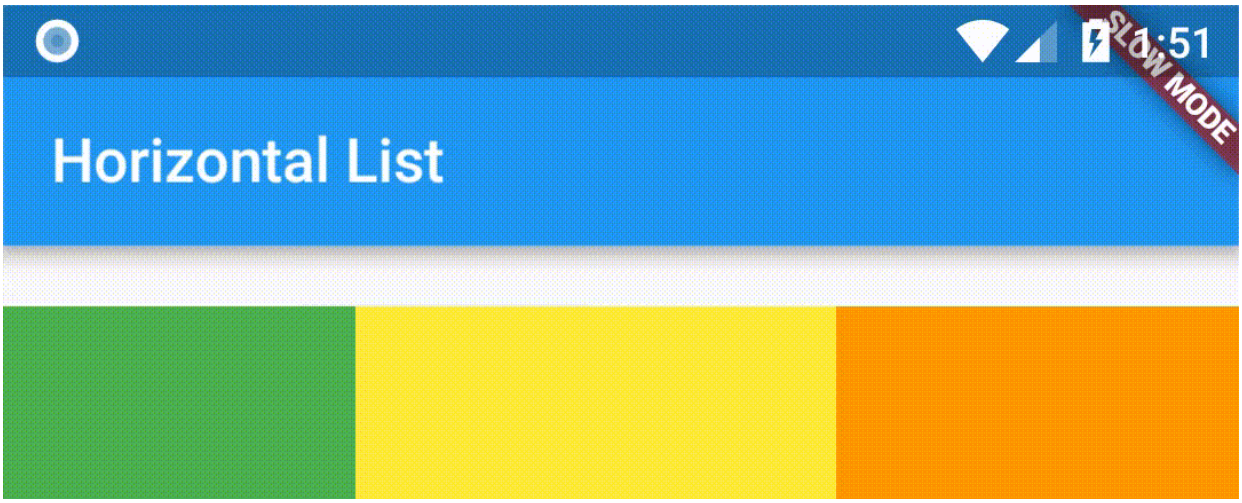
We'll use the standard `ListView` constructor, passing through a horizontal `scrollDirection`, which will override the default vertical direction.

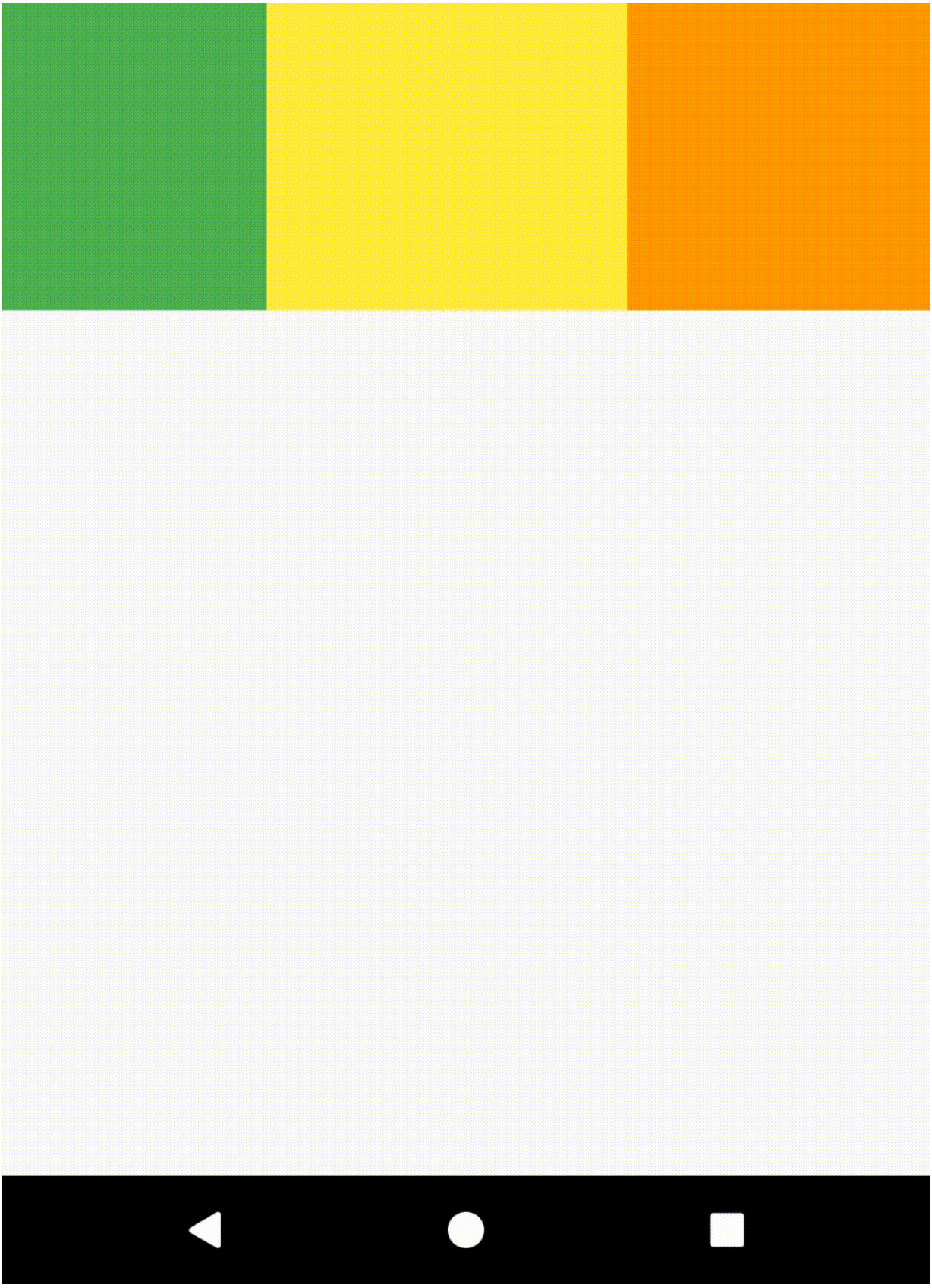
```
new ListView(
  // This next line does the trick.
  scrollDirection: Axis.horizontal,
  children: <Widget>[
    new Container(
      width: 160.0,
      color: Colors.red,
    ),
    new Container(
      width: 160.0,
      color: Colors.blue,
    ),
    new Container(
      width: 160.0,
      color: Colors.green,
    ),
    new Container(
      width: 160.0,
      color: Colors.yellow,
    ),
    new Container(
      width: 160.0,
      color: Colors.orange,
    ),
  ],
)
```

Complete Example

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

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'Horizontal List';

    return new MaterialApp(
      title: title,
      home: new Scaffold(
```

```
        appBar: new AppBar(
          title: new Text(title),
        ),
        body: new Container(
          margin: new EdgeInsets.symmetric(vertical: 20.0),
          height: 200.0,
          child: new ListView(
            scrollDirection: Axis.horizontal,
            children: <Widget>[
              new Container(
                width: 160.0,
                color: Colors.red,
              ),
              new Container(
                width: 160.0,
                color: Colors.blue,
              ),
              new Container(
                width: 160.0,
                color: Colors.green,
              ),
              new Container(
                width: 160.0,
                color: Colors.yellow,
              ),
              new Container(
                width: 160.0,
                color: Colors.orange,
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```





Working with long lists

The standard `ListView` constructor works well for small lists. In order to work with lists that contain a large number of items, it's best to use the `ListView.builder` constructor.

Whereas the default `ListView` constructor requires us to create all items at once, the `ListView.builder` constructor will create items as they are scrolled onto the screen.

1. Create a data source

First, we'll need a data source to work with. For example, your data source might be a list of messages, search results, or products in a store. Most of the time, this data will come from the internet or a database.

For this example, we'll generate a list of 10000 Strings using the `List.generate` constructor.

```
final items = new List<String>.generate(10000, (i) => "Item $i");
```

2. Convert the data source into Widgets

In order to display our List of Strings, we'll need to render each String as a Widget!

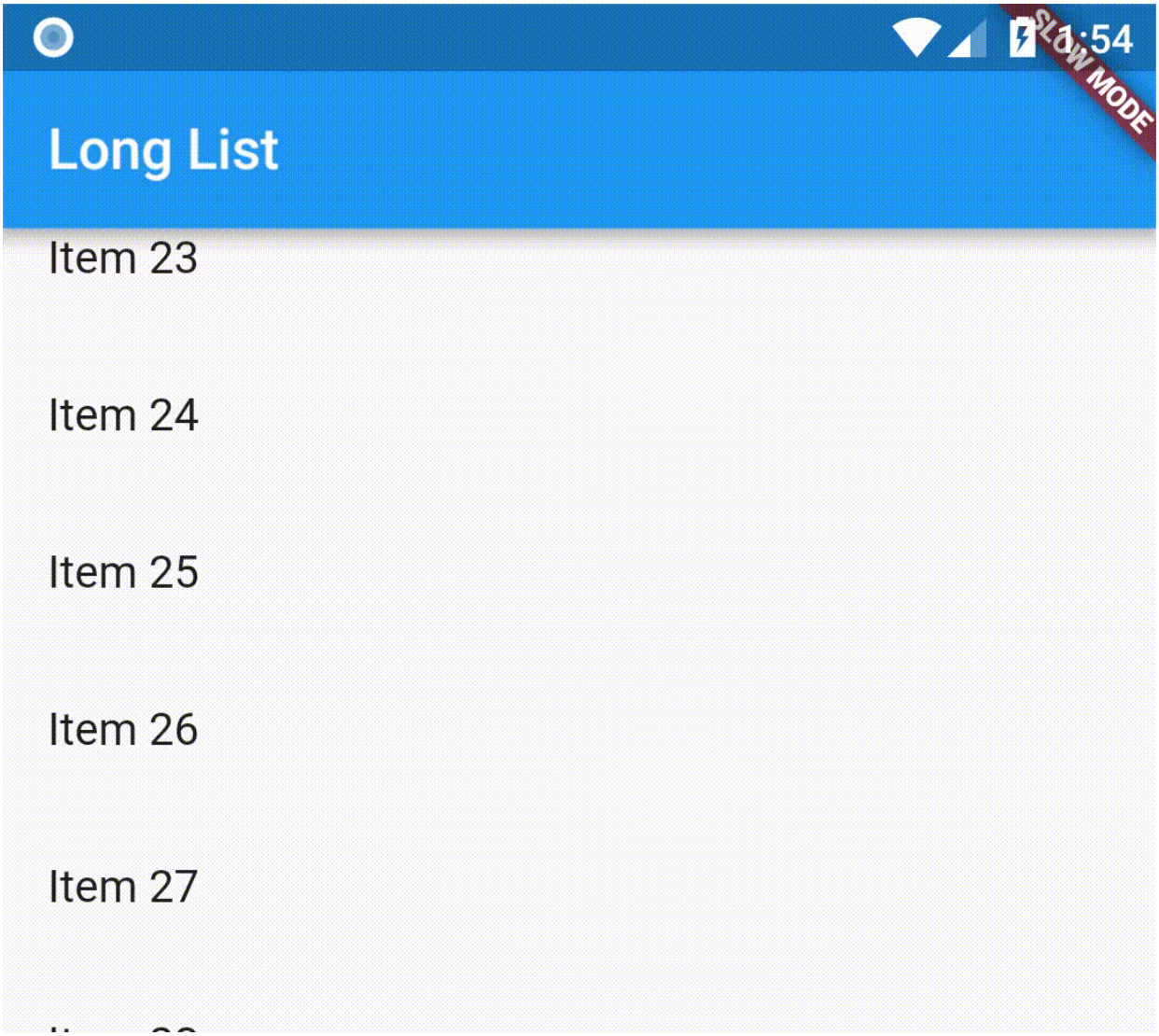
This is where the `ListView.builder` will come into play. In our case, we'll display each String on it's own line.

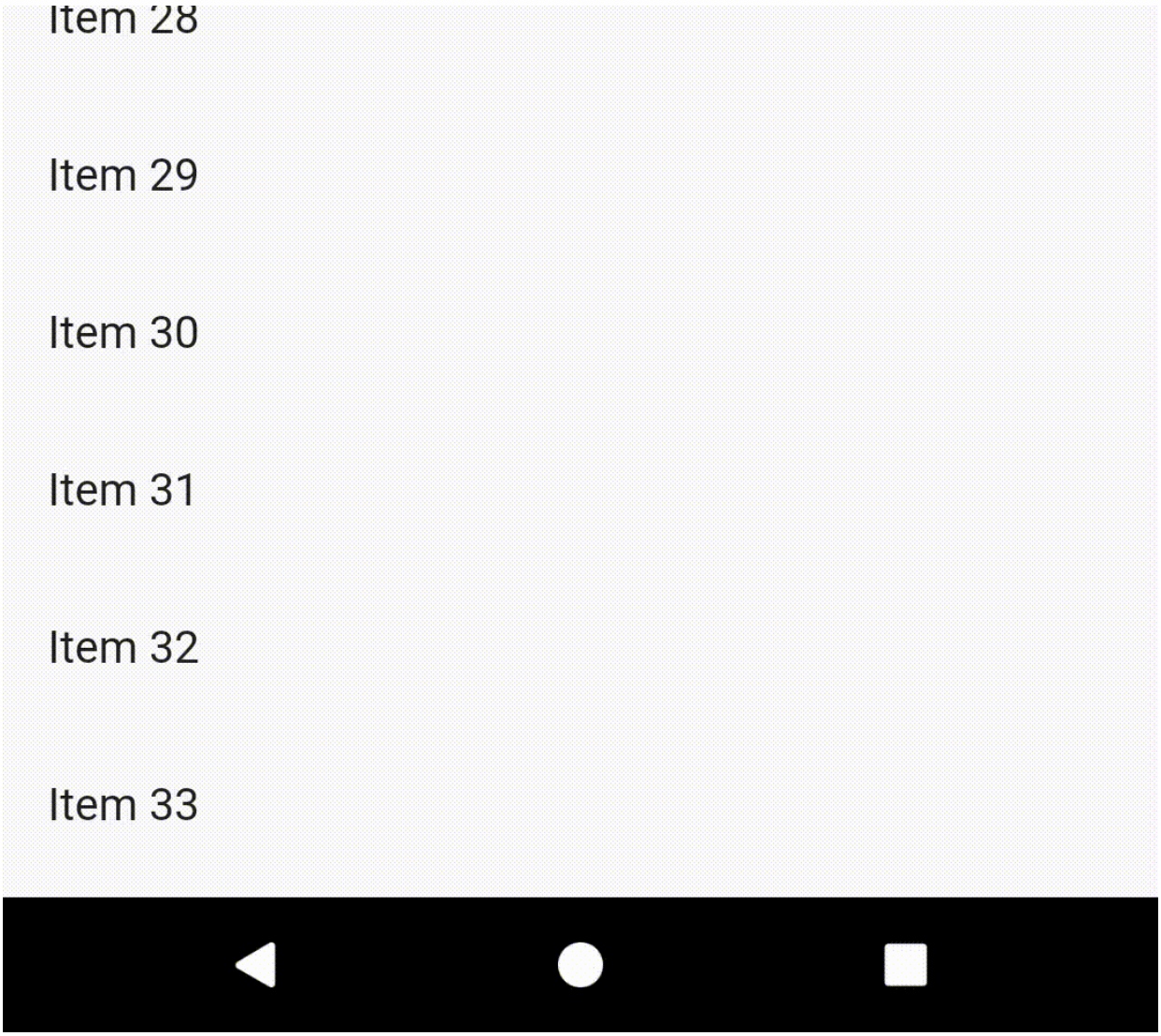
```
new ListView.builder(
  itemCount: items.length,
  itemBuilder: (context, index) {
    return new ListTile(
      title: new Text('${items[index]}'),
    );
  },
);
```

Complete Example

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

void main() {
  runApp(new MyApp(
    items: new List<String>.generate(10000, (i) => "Item $i"),
  ));
}

class MyApp extends StatelessWidget {
  final List<String> items;

  MyApp({Key key, @required this.items}) : super(key: key);

  @override
```

```
  Widget build(BuildContext context) {
    final title = 'Long List';

    return new MaterialApp(
      title: title,
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text(title),
        ),
        body: new ListView.builder(
          itemCount: items.length,
          itemBuilder: (context, index) {
            return new ListTile(
              title: new Text('${items[index]}'),
            );
          },
        ),
      ),
    );
  }
}
```





Creating lists with different types of items

We often need to create lists that display different types of content. For example, we might be working on a List that shows a heading followed by a few items related to the heading, followed by another heading, and so on.

How would we create such a structure with Flutter?

Directions

1. Create a data source with different types of items
2. Convert the data source into a List of Widgets

1. Create a data source with different types of item

Types of Items

In order to represent different types of items in a List, we'll need to define a class for each type of item.

In this example, we'll work on an app that shows a header followed by five messages. Therefore, we'll create three classes: `ListItem`, `HeadingItem`, and `MessageItem`.

```
// The base class for the different types of items the List can contain
abstract class ListItem {}

// A ListItem that contains data to display a heading
class HeadingItem implements ListItem {
  final String heading;

  HeadingItem(this.heading);
}

// A ListItem that contains data to display a message
class MessageItem implements ListItem {
  final String sender;
  final String body;

  MessageItem(this.sender, this.body);
}
```

Create a List of Items

Most of the time, we'd fetch data from the internet or a local database and convert that data into a list of items.

For this example, we'll generate a list of items to work with. The list will contain a header followed by five messages. Rinse, repeat.

```
final items = new List<ListItem>.generate(
  1200,
  (i) => i % 6 == 0
      ? new HeadingItem("Heading $i")
      : new MessageItem("Sender $i", "Message body $i"),
);
```

2. Convert the data source into a List of Widgets

In order to handle converting each item into a Widget, we'll employ the `ListView.builder` constructor.

In general, we'll want to provide a `builder` function that checks for what type of item we're dealing with, and returns the appropriate Widget for that type of item.

In this example, using the `is` keyword to check the type of item we're dealing with can be handy. It's fast, and will automatically cast each item to the appropriate type. However, there are different ways to approach this problem if you prefer another pattern!

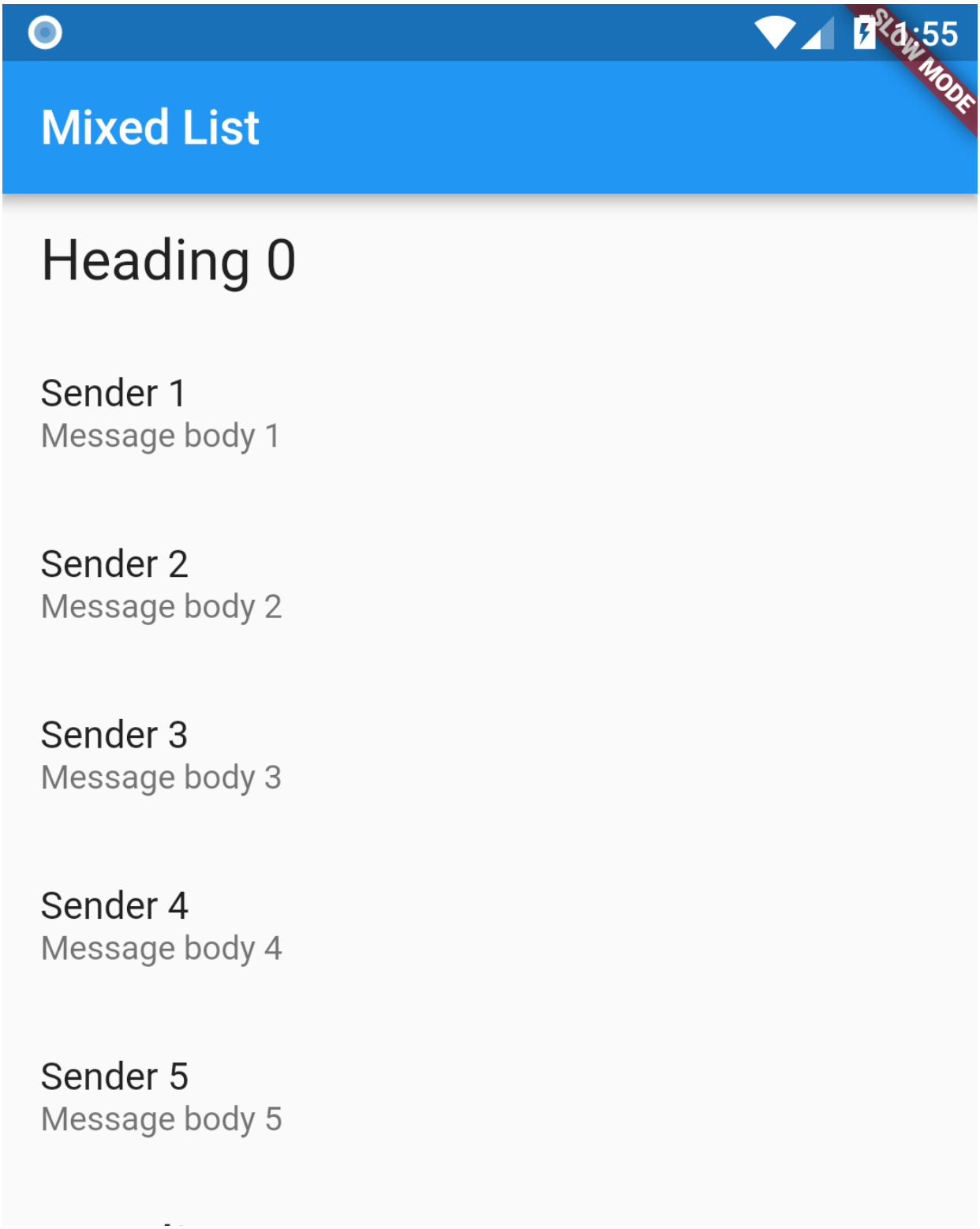
```
new ListView.builder(
  // Let the ListView know how many items it needs to build
  itemCount: items.length,
  // Provide a builder function. This is where the magic happens! We'll
  // convert each item into a Widget based on the type of item it is.
  itemBuilder: (context, index) {
    final item = items[index];

    if (item is HeadingItem) {
      return new ListTile(
        title: new Text(
          item.heading,
          style: Theme.of(context).textTheme.headline,
        ),
      );
    } else if (item is MessageItem) {
      return new ListTile(
        title: new Text(item.sender),
        subtitle: new Text(item.body),
      );
    }
  },
);
```

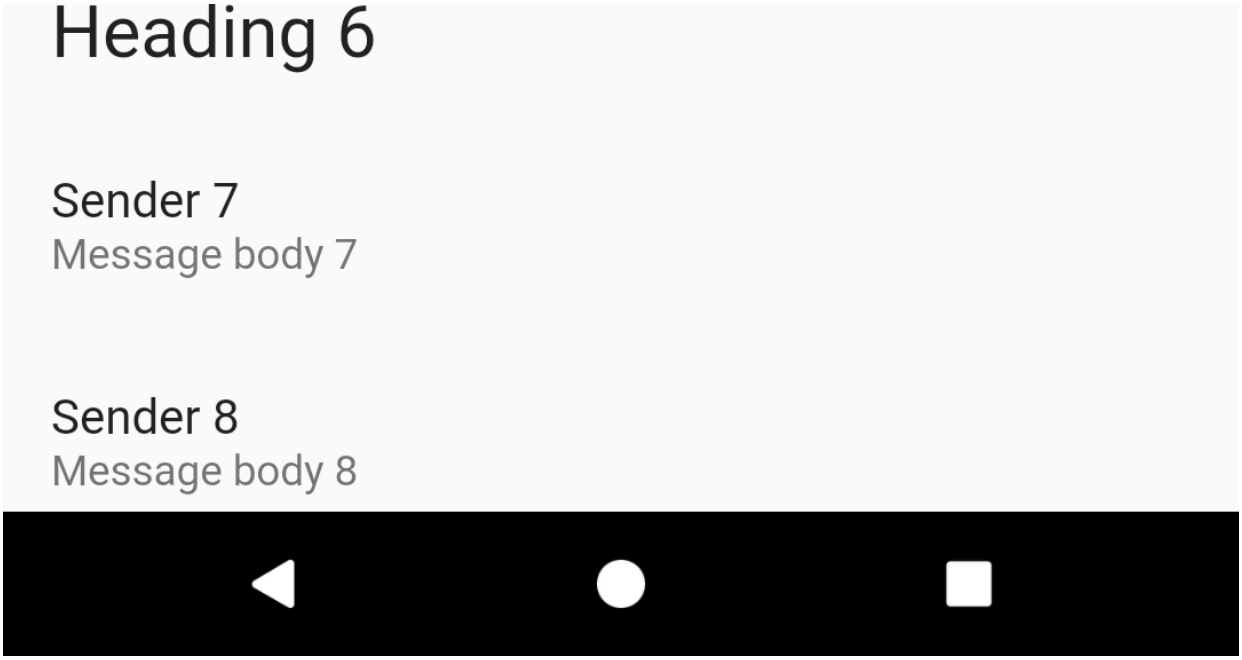
Complete Example

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

void main() {
  runApp(new MyApp(
    items: new List<ListItem>.generate(
      1000,
      (i) => i % 6 == 0
          ? new HeadingItem("Heading $i")
          : new MessageItem("Sender $i", "Message body $i"),
    ),
  ));
}

class MyApp extends StatelessWidget {
  final List<ListItem> items;
```

```
  MyApp({Key key, @required this.items}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final title = 'Mixed List';

    return new MaterialApp(
      title: title,
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text(title),
        ),
        body: new ListView.builder(
          // Let the ListView know how many items it needs to build
          itemCount: items.length,
          // Provide a builder function. This is where the magic happens!
We'll
          // convert each item into a Widget based on the type of item it
is.
          itemBuilder: (context, index) {
            final item = items[index];

            if (item is HeadingItem) {
              return new ListTile(
                title: new Text(
                  item.heading,
                  style: Theme.of(context).textTheme.headline,
                ),
              );
            } else if (item is MessageItem) {
              return new ListTile(
                title: new Text(item.sender),
                subtitle: new Text(item.body),
              );
            }
          },
        ),
      ),
    );
  }
}

// The base class for the different types of items the List can contain
abstract class ListItem {}

// A ListItem that contains data to display a heading
class HeadingItem implements ListItem {
  final String heading;
```

```
  HeadingItem(this.heading);
}

// A ListItem that contains data to display a message
class MessageItem implements ListItem {
  final String sender;
  final String body;

  MessageItem(this.sender, this.body);
}
```





Creating a Grid List

In some cases, you might want to display your items as a Grid rather than a normal list of items that come one after the next. For this task, we'll employ the `GridView` Widget.

The simplest way to get started using grids is by using the `GridView.count` constructor, because it allow us to specify how many rows or columns we'd like.

In this example, we'll generate a List of 100 Widgets that display their index in the list. This will help us us visualize how `GridView` works.

```
new GridView.count(
  // Create a grid with 2 columns. If you change the scrollDirection to
  // horizontal, this would produce 2 rows.
  crossAxisCount: 2,
  // Generate 100 Widgets that display their index in the List
  children: new List.generate(100, (index) {
    return new Center(
      child: new Text(
        'Item $index',
        style: Theme.of(context).textTheme.headline,
      ),
    );
  }),
);
```

Complete Example

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

```
void main() {
  runApp(new MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'Grid List';

    return new MaterialApp(
      title: title,
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text(title),
        ),
        body: new GridView.count(
          // Create a grid with 2 columns. If you change the
scrollDirection to
          // horizontal, this would produce 2 rows.
          crossAxisCount: 2,
          // Generate 100 Widgets that display their index in the List
          children: new List.generate(100, (index) {
            return new Center(
              child: new Text(
                'Item $index',
                style: Theme.of(context).textTheme.headline,
              ),
            );
          }),
        ),
      ),
    );
  }
}
```

Handling Taps

We not only want to display information to our users, we want our users to interact with our apps! So how do we respond to fundamental actions such as tapping and dragging? We'll use the `GestureDetector` Widget!

Say we want to make a custom button that shows a snackbar when tapped. How would we approach this?

Directions

1. Create the button

2. Wrap it in a `GestureDetector` with an `onTap` callback

```
// Our GestureDetector wraps our button
new GestureDetector(
  // When the child is tapped, show a snackbar
  onTap: () {
    final snackBar = new SnackBar(content: new Text("Tap"));

    Scaffold.of(context).showSnackBar(snackBar);
  },
  // Our Custom Button!
  child: new Container(
    padding: new EdgeInsets.all(12.0),
    decoration: new BoxDecoration(
      color: Theme.of(context).buttonColor,
      borderRadius: new BorderRadius.circular(8.0),
    ),
    child: new Text('My Button'),
  ),
);
```

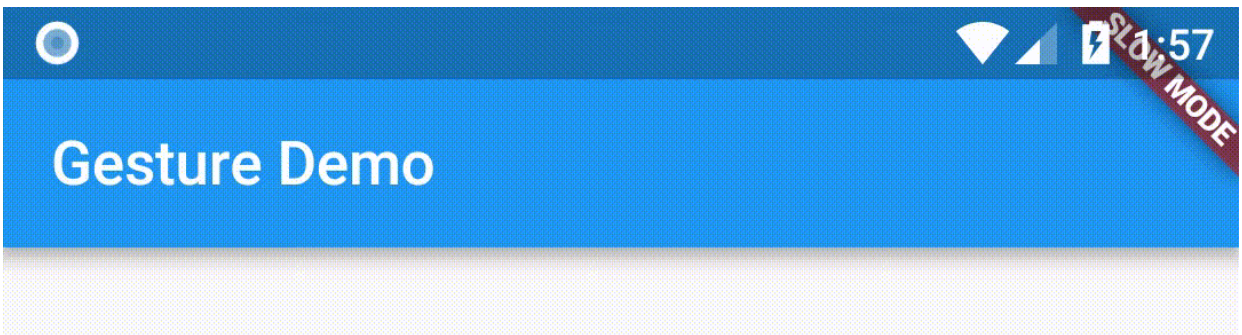
Notes

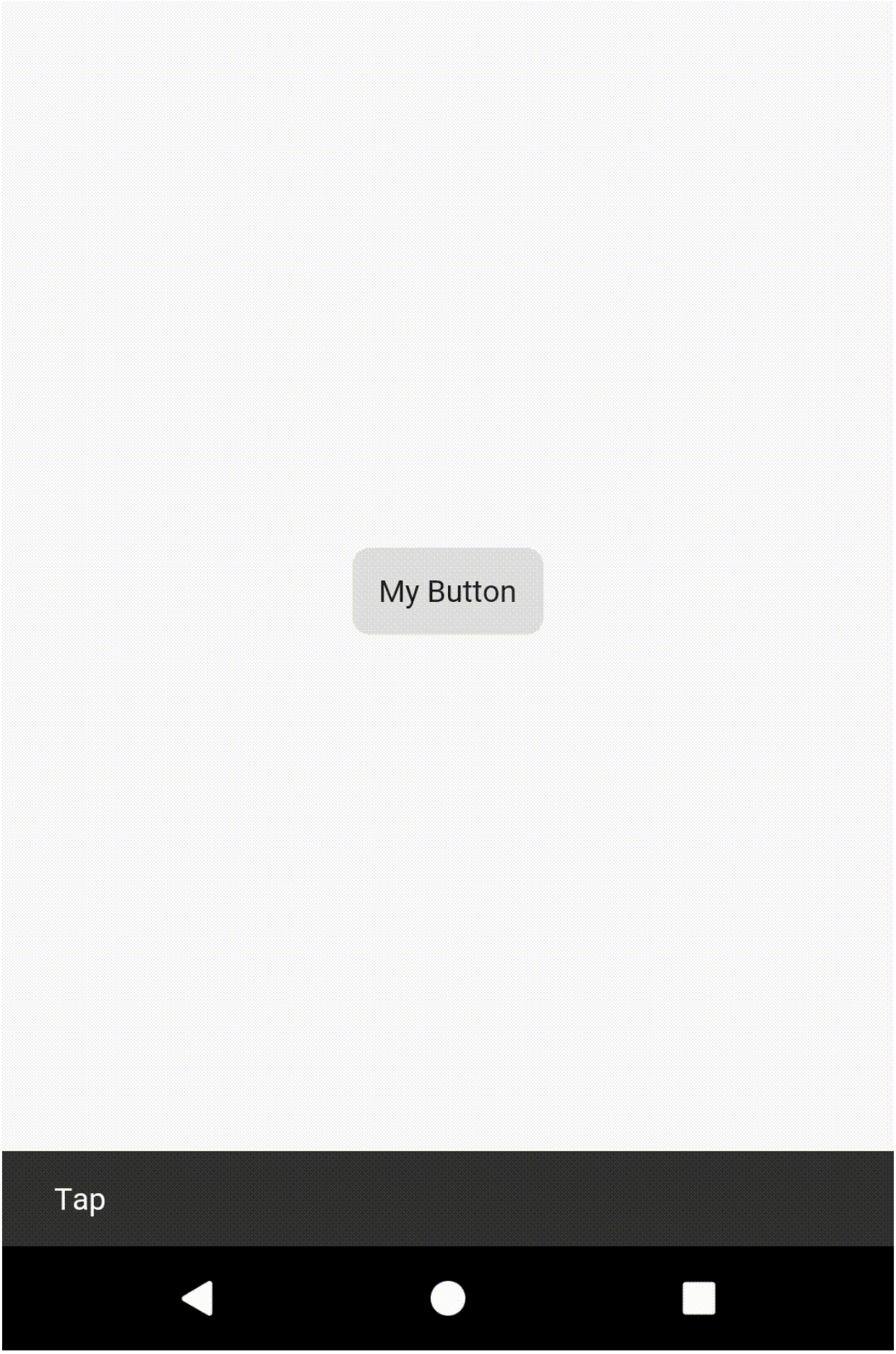
1. If you'd like to add the Material Ripple effect to your button, please see the "Adding Material Touch ripples" recipe.
2. While we've created a custom button to demonstrate these concepts, Flutter includes a handful of buttons out of the box: RaisedButton, FlatButton, and CupertinoButton

Complete Example

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

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'Gesture Demo';

    return new MaterialApp(
      title: title,
      home: new MyHomePage(title: title),
    );
  }
}
```

```
class MyHomePage extends StatelessWidget {
  final String title;

  MyHomePage({Key key, this.title}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(title),
      ),
      body: new Center(child: new MyButton()),
    );
  }
}

class MyButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // Our GestureDetector wraps our button
    return new GestureDetector(
      // When the child is tapped, show a snackbar
      onTap: () {
        final snackBar = new SnackBar(content: new Text("Tap"));

        Scaffold.of(context).showSnackBar(snackBar);
      },
      // Our Custom Button!
      child: new Container(
        padding: new EdgeInsets.all(12.0),
        decoration: new BoxDecoration(
          color: Theme.of(context).buttonColor,
          borderRadius: new BorderRadius.circular(8.0),
        ),
        child: new Text('My Button'),
      ),
    );
  }
}
```





Adding Material Touch Ripples

While designing an app that should follow the Material Design Guidelines, we'll want to add the ripple animation to Widgets when tapped.

Flutter provides the `InkWell` Widget to achieve this effect.

Directions

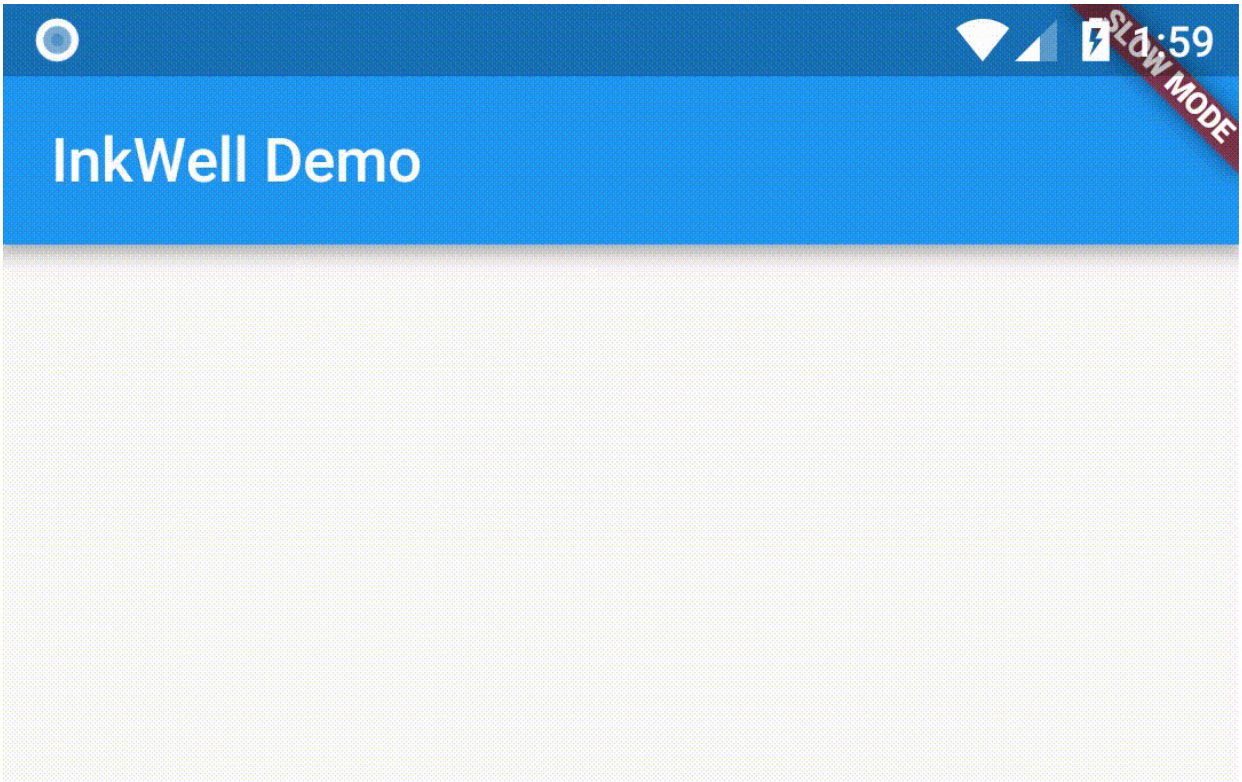
1. Create a Widget we want to tap
2. Wrap it in an `InkWell` Widget to manage tap callbacks and ripple animations

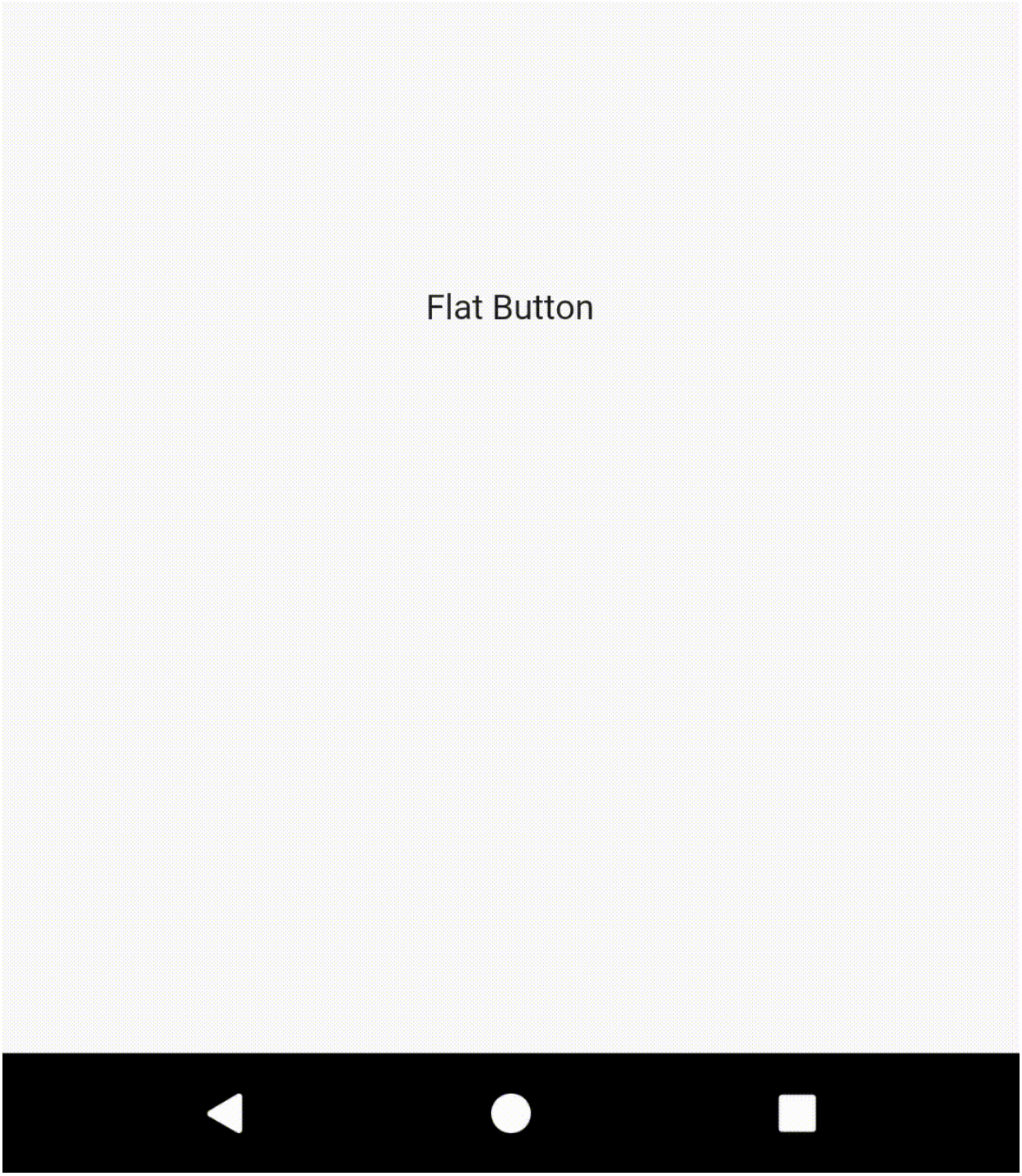
```
// The InkWell Wraps our custom flat button Widget
new InkWell(
  // When the user taps the button, show a snackbar
  onTap: () {
    Scaffold.of(context).showSnackBar(new SnackBar(
      content: new Text('Tap'),
    ));
  },
  child: new Container(
    padding: new EdgeInsets.all(12.0),
    child: new Text('Flat Button'),
  ),
);
```

Complete Example

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

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'InkWell Demo';

    return new MaterialApp(
      title: title,
      home: new MyHomePage(title: title),
    );
  }
}

class MyHomePage extends StatelessWidget {
  final String title;

  MyHomePage({Key key, this.title}) : super(key: key);
```

```
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(title),
      ),
      body: new Center(child: new MyButton()),
    );
  }
}

class MyButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // The InkWell Wraps our custom flat button Widget
    return new InkWell(
      // When the user taps the button, show a snackbar
      onTap: () {
        Scaffold.of(context).showSnackBar(new SnackBar(
          content: new Text('Tap'),
        ));
      },
      child: new Container(
        padding: new EdgeInsets.all(12.0),
        child: new Text('Flat Button'),
      ),
    );
  }
}
```





Implement Swipe to Dismiss

The "Swipe to dismiss" pattern is common in many mobile apps. For example, if we're writing an email app, we might want to allow our users to swipe away email messages in a list. When they do, we'll want to move the item from the Inbox to the Trash.

Flutter makes this task easy by providing the `Dismissible` Widget.

Directions

1. Create List of Items
2. Wrap each item in a `Dismissible` Widget

3. Provide "Leave Behind" indicators

1. Create List of Items

The first step in this recipe will be to create a list of items we can swipe away. For more detailed instructions on how to create a list, please follow the Working with long lists recipe.

Create a Data Source

In our example, we'll want 20 sample items to work with. To keep it simple, we'll generate a List of Strings.

```
final items = new List<String>.generate(20, (i) => "Item ${i + 1}");
```

Convert the data source into a List

At first, we'll simply display each item in the List on screen. Users will not be able to swipe away with these items just yet!

```
new ListView.builder(
  itemCount: items.length,
  itemBuilder: (context, index) {
    return new ListTile(title: new Text('${items[index]}'));
  },
);
```

2. Wrap each item in a Dismissible Widget

Now that we're displaying a list of items, we'll want to give our users the ability to swipe each item off the list!

After the user has swiped away the item, we'll need to run some code to remove the item from the list and display a Snackbar. In a real app, you might need to perform more complex logic, such as removing the item from a web service or database.

This is where the `Dismissible` Widget comes into play! In our example, we'll update our `itemBuilder` function to return a `Dismissible` Widget.

```
new Dismissible(
  // Each Dismissible must contain a Key. Keys allow Flutter to
  // uniquely identify Widgets.
  key: new Key(item),
  // We also need to provide a function that will tell our app
  // what to do after an item has been swiped away.
  onDismissed: (direction) {
    // Remove the item from our data source
    items.removeAt(index);
```

```
    // Show a snackbar! This snackbar could also contain "Undo" actions.
    Scaffold.of(context).showSnackBar(
        new SnackBar(content: new Text("$item dismissed")));
  },
  child: new ListTile(title: new Text('$item')),
);
```

3. Provide "Leave Behind" indicators

As it stands, our app will allow users to swipe items off the List, but it might not give them a visual indication of what happens when they do. To provide a cue that we're removing items, we'll display a "Leave Behind" indicator as they swipe the item off the screen. In this case, a red background!

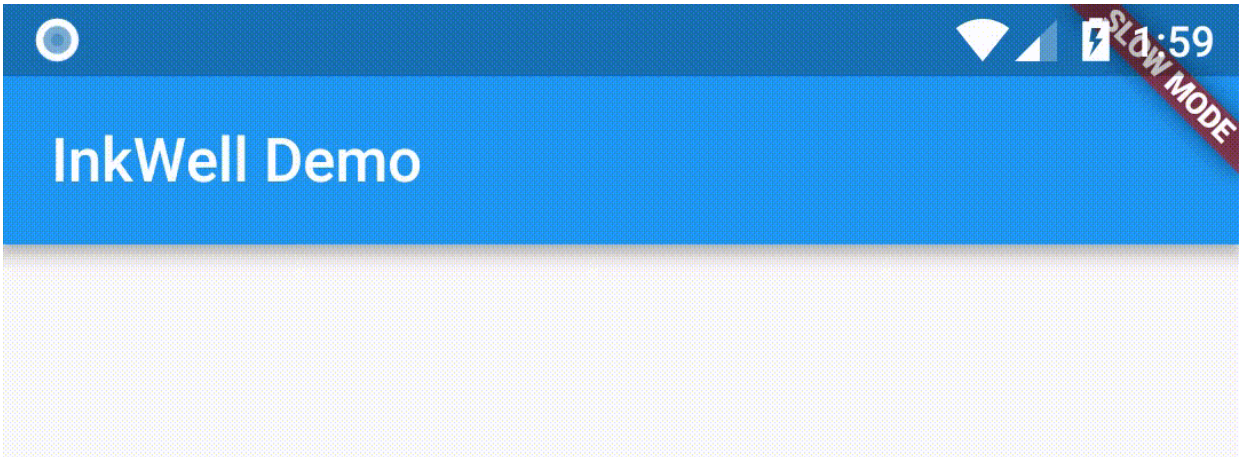
For this purpose, we'll provide a `background` parameter to the `Dismissible`.

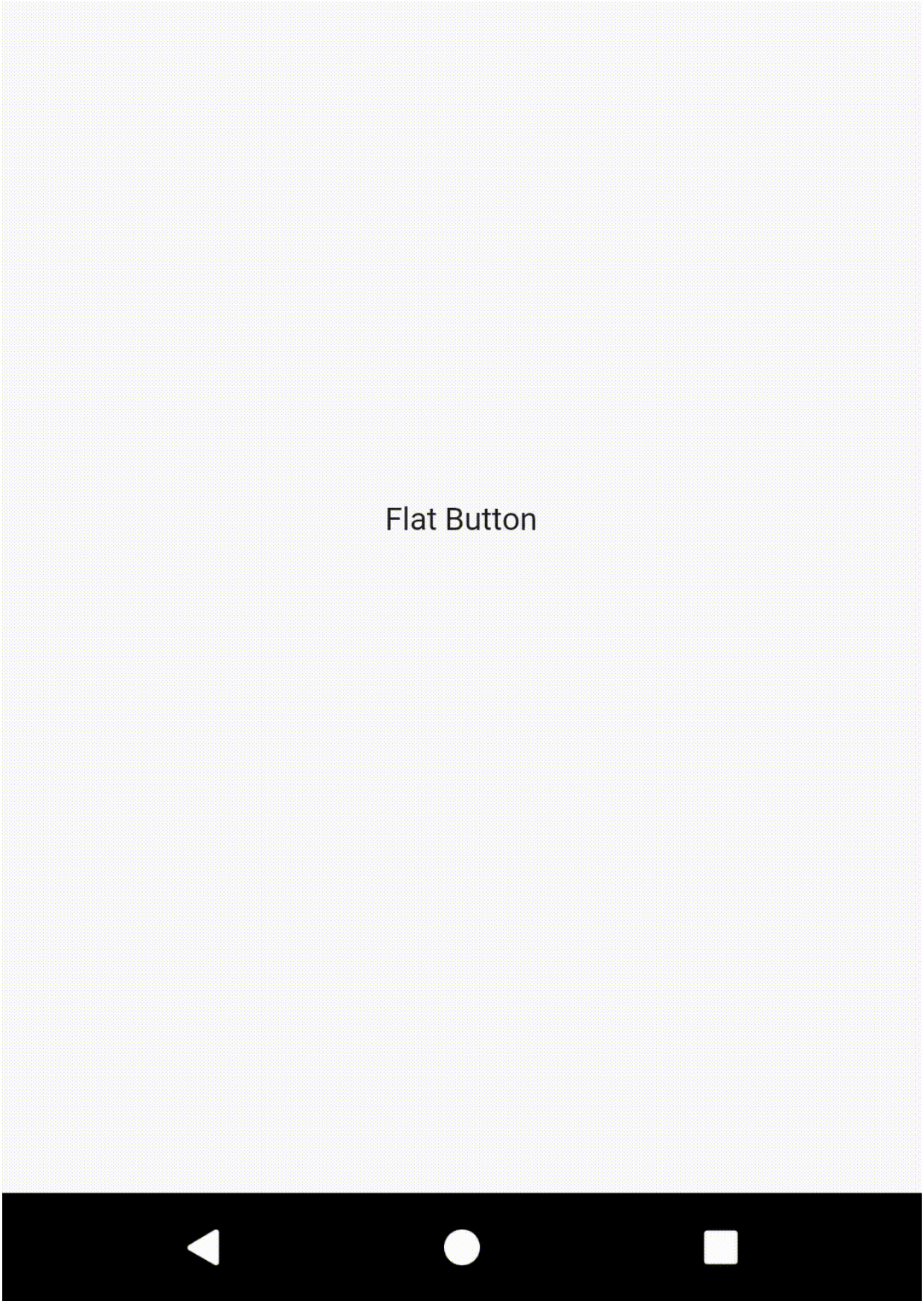
```
new Dismissible(
  // Show a red background as the item is swiped away
  background: new Container(color: Colors.red),
  key: new Key(item),
  onDismissed: (direction) {
    items.removeAt(index);

    Scaffold.of(context).showSnackBar(
        new SnackBar(content: new Text("$item dismissed")));
  },
  child: new ListTile(title: new Text('$item')),
);
```

Complete Example

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

void main() {
  runApp(new MyApp(
    items: new List<String>.generate(20, (i) => "Item ${i + 1}"),
  ));
}

class MyApp extends StatelessWidget {
  final List<String> items;

  MyApp({Key key, @required this.items}) : super(key: key);
```

```
  @override
  Widget build(BuildContext context) {
    final title = 'Dismissing Items';

    return new MaterialApp(
      title: title,
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text(title),
        ),
        body: new ListView.builder(
          itemCount: items.length,
          itemBuilder: (context, index) {
            final item = items[index];

            return new Dismissible(
              // Each Dismissible must contain a Key. Keys allow Flutter to
              // uniquely identify Widgets.
              key: new Key(item),
              // We also need to provide a function that will tell our app
              // what to do after an item has been swiped away.
              onDismissed: (direction) {
                items.removeAt(index);

                Scaffold.of(context).showSnackBar(
                    new SnackBar(content: new Text("$item dismissed")));
              },
              // Show a red background as the item is swiped away
              background: new Container(color: Colors.red),
              child: new ListTile(title: new Text('$item')),
            );
          },
        ),
      ),
    );
  }
}
```





Navigate to a new screen and back

Most apps contain several screens for displaying different types of information. For example, we might have a screen that shows products. Our users could then tap on a product to get more information about it on a new screen.

In Android terms, our screens would be new Activities. In iOS terms, new ViewControllers. In Flutter, screens are just Widgets!

So how do we navigate to new screens? Using the `Navigator`!

Directions

1. Create two screens
2. Navigate to the second screen using `Navigator.push`
3. Return to the first screen using `Navigator.pop`

1. Create two screens

First, we'll need two screens to work with. Since this is a basic example, we'll create two screens, each containing a single button. Tapping the button on the first screen will navigate to the second screen. Tapping the button on the second screen will return our user back to the first!

First, we'll set up the visual structure.

```
class FirstScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('First Screen'),
      ),
      body: new Center(
        child: new RaisedButton(
          child: new Text('Launch new screen'),
          onPressed: () {
            // Navigate to second screen when tapped!
          },
        ),
      ),
    );
  }
}

class SecondScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text("Second Screen"),
      ),
```

```
      body: new Center(
        child: new RaisedButton(
          onPressed: () {
            // Navigate back to first screen when tapped!
          },
          child: new Text('Go back!'),
        ),
      ),
    );
  }
}
```

2. Navigate to the second screen using `Navigator.push`

In order to Navigate to a new screen, we'll need to use the `Navigator.push` method. The `push` method will add a `Route` to the stack of routes managed by the Navigator!

The `push` method requires a `Route`, but where does the `Route` come from? We can create our own, or use the `MaterialPageRoute` out of the box. The `MaterialPageRoute` is handy because it transitions to the new screen using a platform-specific animation.

In the `build` method of our `FirstScreen` Widget, we'll update the `onPressed` callback:

```
// Within the `FirstScreen` Widget
onPressed: () {
  Navigator.push(
    context,
    new MaterialPageRoute(builder: (context) => new SecondScreen()),
  );
}
```

3. Return to the first screen using `Navigator.pop`

Now that we're on our second screen, how do we close it out and return to the first? Using the `Navigator.pop` method! The `pop` method will remove the current `Route` from the stack of routes managed by the navigator.

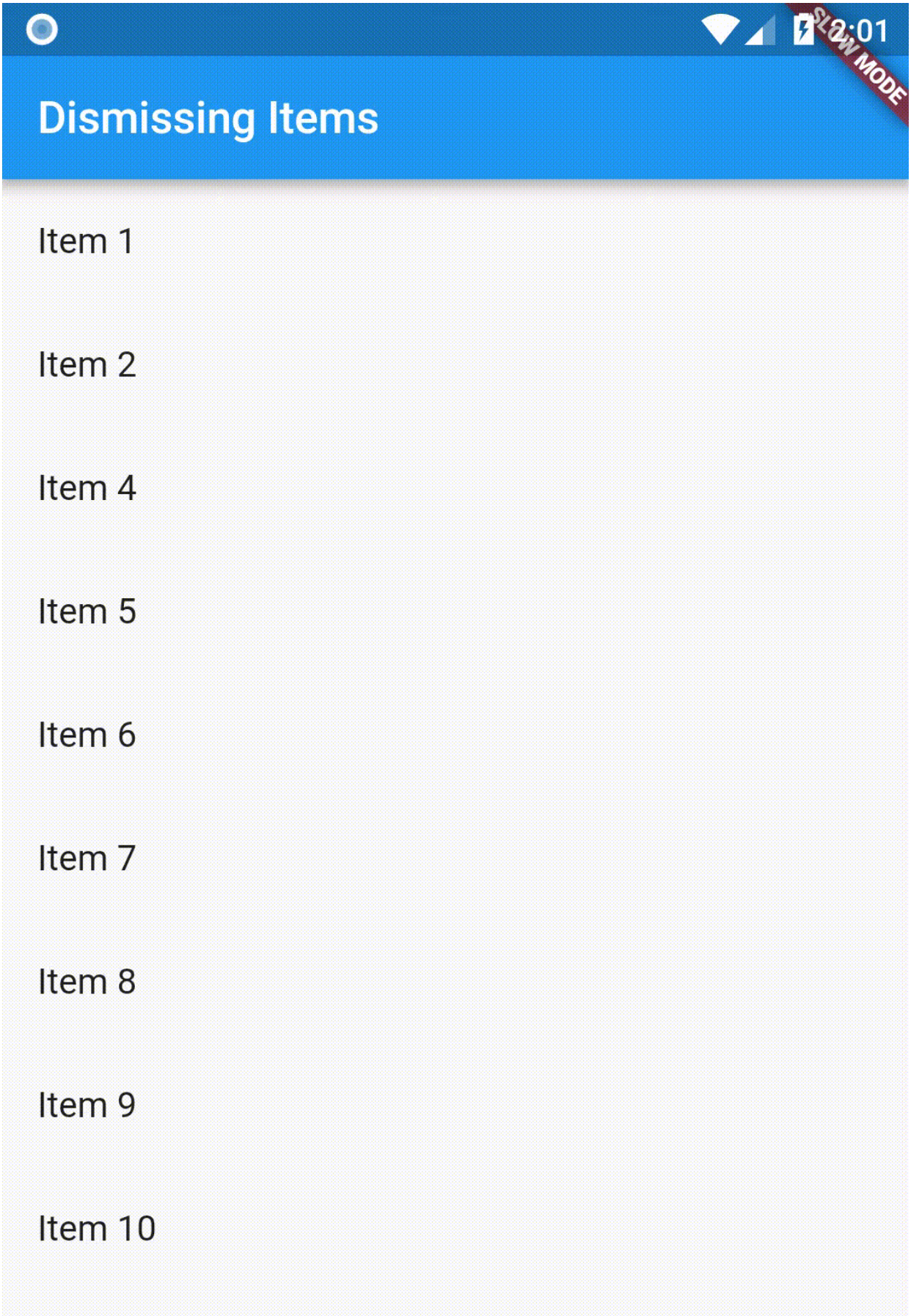
For this part, we'll need to update the `onPressed` callback found in our `SecondScreen` Widget

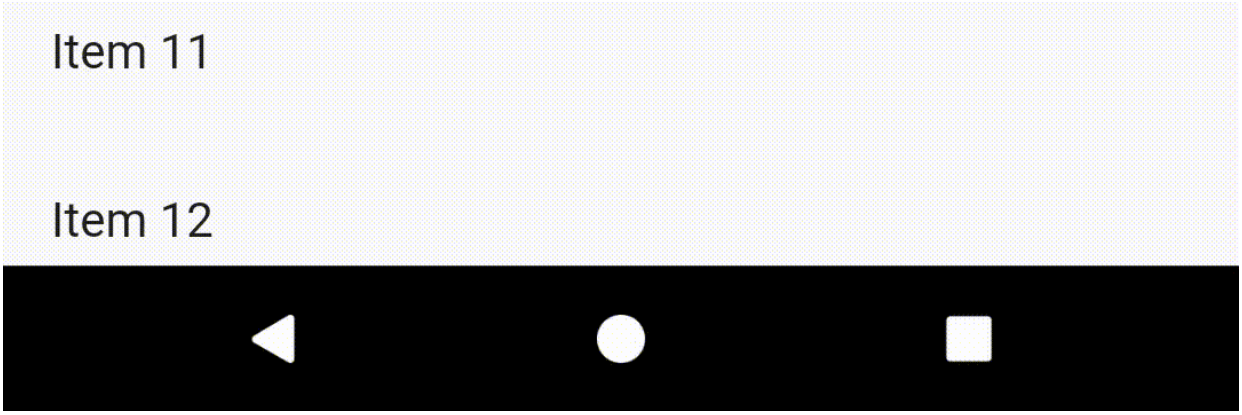
```
// Within the SecondScreen Widget
onPressed: () {
  Navigator.pop(context);
}
```

Complete Example

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

void main() {
  runApp(new MaterialApp(
    title: 'Navigation Basics',
    home: new FirstScreen(),
  ));
}

class FirstScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('First Screen'),
      ),
      body: new Center(
        child: new RaisedButton(
          child: new Text('Launch new screen'),
          onPressed: () {
            Navigator.push(
              context,
              new MaterialPageRoute(builder: (context) => new
SecondScreen()),
            );
          },
        ),
      ),
    );
  }
}

class SecondScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text("Second Screen"),
      ),
      body: new Center(
        child: new RaisedButton(
          onPressed: () {
            Navigator.pop(context);
          },
          child: new Text('Go back!'),
        ),
```

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


8:01
SLOW MODE
Dismissing Items
Item 1
Item 2
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
Item 10



Send data to a new screen

Oftentimes, we not only want to navigate to a new screen, but also pass some data to the screen as well. For example, we often want to pass information about the item we tapped on.

Remember: Screens are Just Widgets™. In this example, we'll create a List of Todos. When a todo is tapped on, we'll navigate to a new screen (Widget) that displays information about the todo.

Directions

1. Define a Todo class
2. Display a List of Todos
3. Create a Detail Screen that can display information about a todo
4. Navigate and pass data to the Detail Screen

1. Define a Todo class

First, we'll need a simple way to represent Todos. For this example, we'll create a class that contains two pieces of data: the title and description.

```
class Todo {
  final String title;
  final String description;

  Todo(this.title, this.description);
}
```

2. Create a List of Todos

Second, we'll want to display a list of Todos. In this example, we'll generate 20 todos and show them using a ListView. For more information on working with Lists, please see the `Basic List` recipe.

Generate the List of Todos

```
final todos = new List<Todo>.generate(
  20,
  (i) => new Todo(
        'Todo $i',
        'A description of what needs to be done for Todo $i',
      ),
);
```

Display the List of Todos using a ListView

```
new ListView.builder(
  itemCount: todos.length,
  itemBuilder: (context, index) {
    return new ListTile(
      title: new Text(todos[index].title),
    );
  },
);
```

So far, so good. We'll generate 20 Todos and display them in a ListView!

3. Create a Detail Screen that can display information about a todo

Now, we'll create our second screen. The title of the screen will contain the title of the todo, and the body of the screen will show the description.

Since it's a normal `StatelessWidget`, we'll simply require that users creating the Screen pass through a `Todo`! Then, we'll build a UI using the given Todo.

```
class DetailScreen extends StatelessWidget {
  // Declare a field that holds the Todo
  final Todo todo;

  // In the constructor, require a Todo
  DetailScreen({Key key, @required this.todo}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    // Use the Todo to create our UI
    return new Scaffold(
      appBar: new AppBar(
        title: new Text("${todo.title}"),
      ),
      body: new Padding(
        padding: new EdgeInsets.all(16.0),
```

```
        child: new Text('${todo.description}'),
      ),
    );
  }
}
```

4. Navigate and pass data to the Detail Screen

With our `DetailScreen` in place, we're ready to perform the Navigation! In our case, we'll want to Navigate to the `DetailScreen` when a user taps on a Todo in our List. When we do so, we'll also want to pass the Todo to the `DetailScreen`.

To achieve this, we'll write an `onTap` callback for our `ListTile` Widget. Within our `onTap` callback, we'll once again employ the `Navigator.push` method.

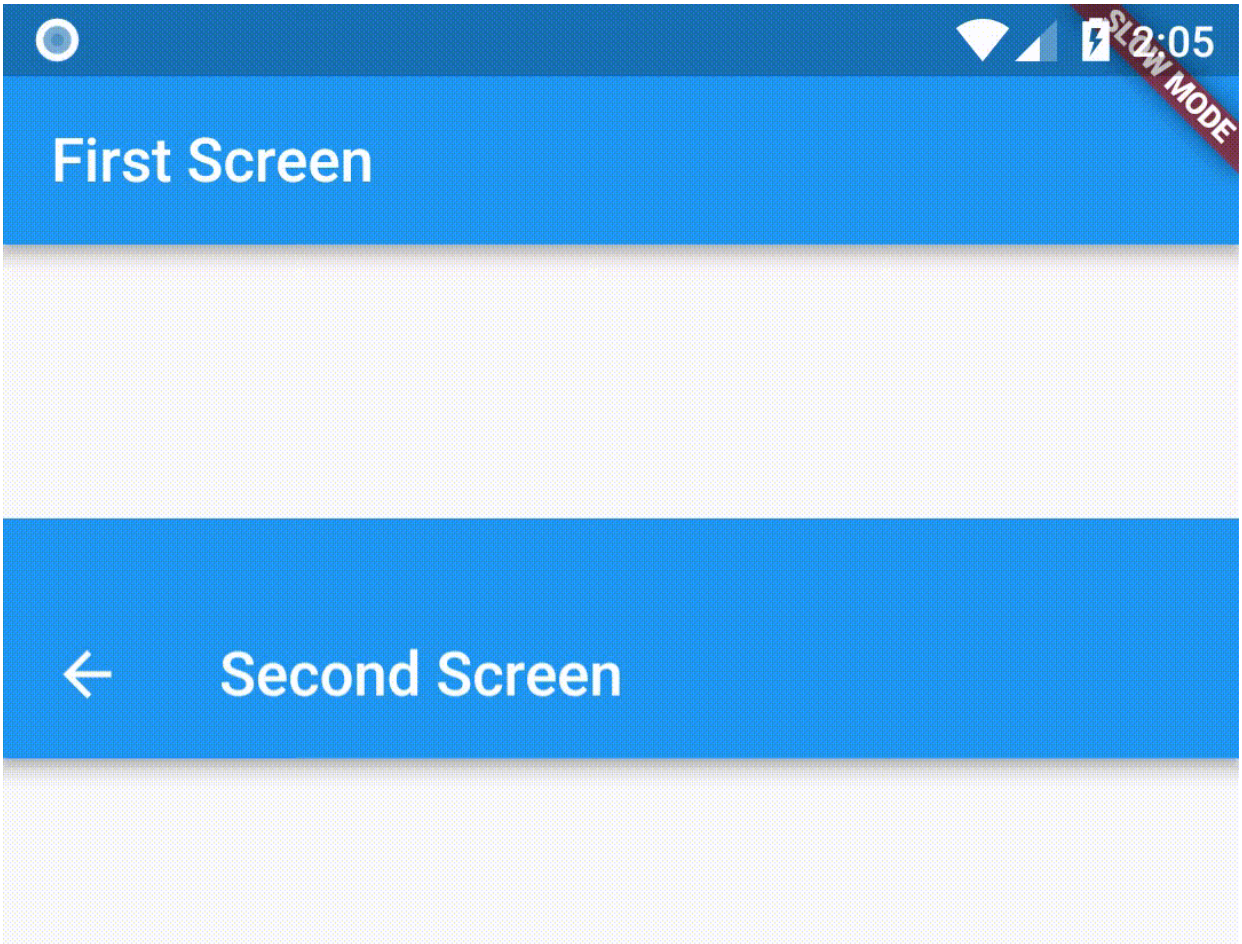
```
new ListView.builder(
  itemCount: todos.length,
  itemBuilder: (context, index) {
    return new ListTile(
      title: new Text(todos[index].title),
      // When a user taps on the ListTile, navigate to the DetailScreen.
      // Notice that we're not only creating a new DetailScreen, we're
      // also passing the current todo to it!
      onTap: () {
        Navigator.push(
          context,
          new MaterialPageRoute(
            builder: (context) => new DetailScreen(todo: todos[index]),
          ),
        );
      },
    );
  },
);
```

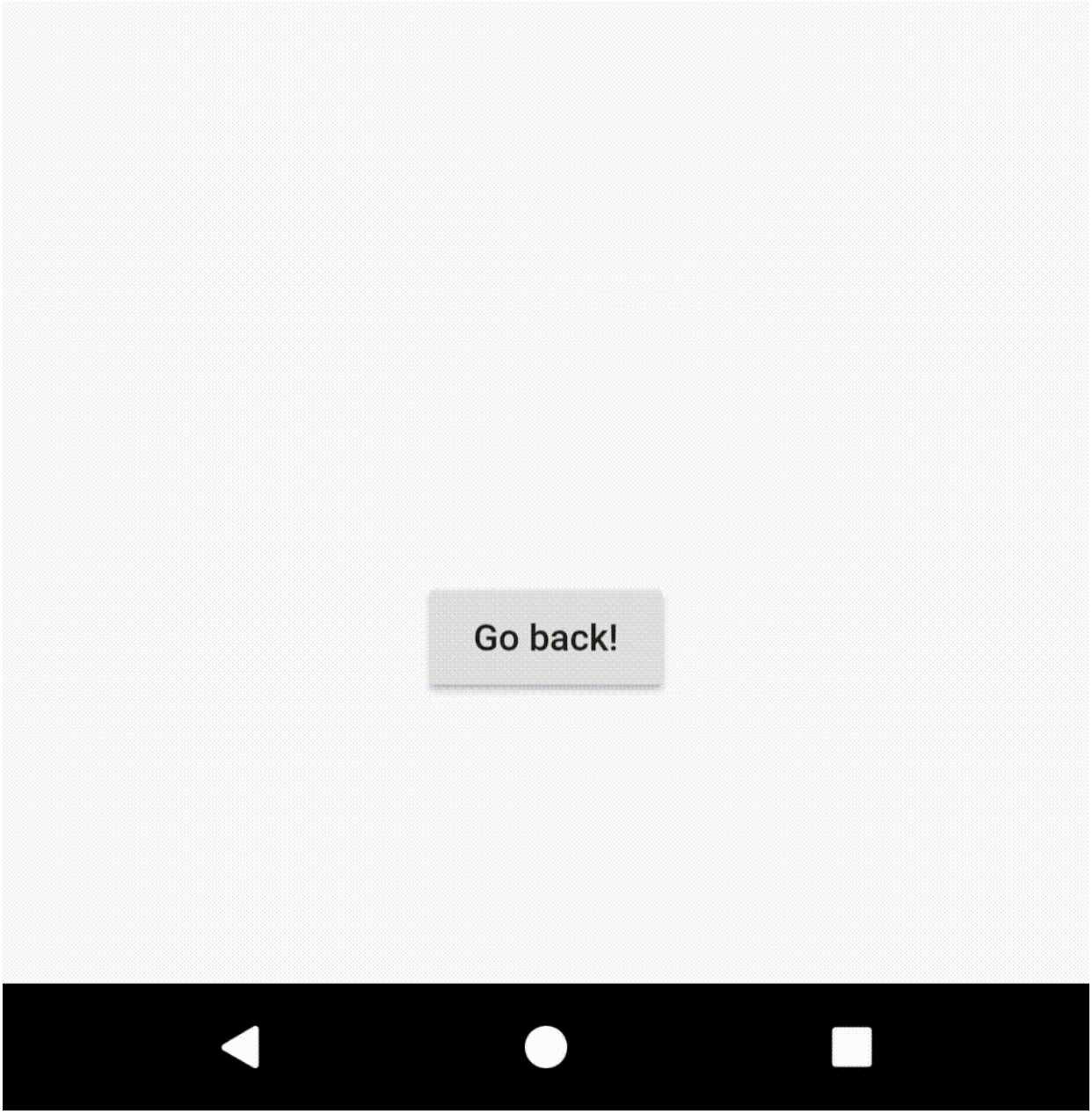
Complete Example

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

class Todo {
  final String title;
  final String description;

  Todo(this.title, this.description);
}
```

```
void main() {
  runApp(new MaterialApp(
    title: 'Passing Data',
    home: new TodosScreen(
      todos: new List.generate(
        20,
        (i) => new Todo(
              'Todo $i',
              'A description of what needs to be done for Todo $i',
            ),
      ),
    ),
  ));
}

class TodosScreen extends StatelessWidget {
  final List<Todo> todos;

  TodosScreen({Key key, @required this.todos}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('Todos'),
      ),
      body: new ListView.builder(
        itemCount: todos.length,
        itemBuilder: (context, index) {
          return new ListTile(
            title: new Text(todos[index].title),
            // When a user taps on the ListTile, navigate to the
DetailScreen.
            // Notice that we're not only creating a new DetailScreen,
we're
            // also passing the current todo through to it!
            onTap: () {
              Navigator.push(
                context,
                new MaterialPageRoute(
                  builder: (context) => new DetailScreen(todo:
todos[index]),
                ),
              );
            },
          );
        },
      ),
```

```
    );
  }
}

class DetailScreen extends StatelessWidget {
  // Declare a field that holds the Todo
  final Todo todo;

  // In the constructor, require a Todo
  DetailScreen({Key key, @required this.todo}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    // Use the Todo to create our UI
    return new Scaffold(
      appBar: new AppBar(
        title: new Text("${todo.title}"),
      ),
      body: new Padding(
        padding: new EdgeInsets.all(16.0),
        child: new Text('${todo.description}'),
      ),
    );
  }
}
```





Return data from a screen

In some cases, we might want to return data from a new screen. For example, say we push a new screen that presents two options to a user. When the user taps on an option, we'll want to inform our first screen of the user's selection so it can act on that information!

How can we achieve this? Using `Navigator.pop`!

Directions

1. Define the home screen
2. Add a button that launches the selection screen
3. Show the selection screen with two buttons
4. When a button is tapped, close the selection screen
5. Show a snackbar on the home screen with the selection

1. Define the home screen

The home screen will display a button. When tapped, it will launch the selection screen!

```
class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('Returning Data Demo'),
      ),
      // We'll create the SelectionButton Widget in the next step
      body: new Center(child: new SelectionButton()),
    );
  }
}
```

2. Add a button that launches the selection screen

Now, we'll create our SelectionButton. Our selection button will:

1. Launch the SelectionScreen when it's tapped
2. Wait for the SelectionScreen to return a result

```
class SelectionButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new RaisedButton(
      onPressed: () {
        _navigateAndDisplaySelection(context);
      },
      child: new Text('Pick an option, any option!'),
    );
  }

  // A method that launches the SelectionScreen and awaits the result from
  // Navigator.pop
  _navigateAndDisplaySelection(BuildContext context) async {
    // Navigator.push returns a Future that will complete after we call
    // Navigator.pop on the Selection Screen!
    final result = await Navigator.push(
      context,
      // We'll create the SelectionScreen in the next step!
      new MaterialPageRoute(builder: (context) => new SelectionScreen()),
    );
  }
}
```

3. Show the selection screen with two buttons

Now, we'll need to build a selection screen! It will contain two buttons. When a user taps on a button, it should close the selection screen and let the home screen know which button was tapped!

For now, we'll define the UI, and figure out how to return data in the next step.

```
class SelectionScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('Pick an option'),
      ),
      body: new Center(
        child: new Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            new Padding(
              padding: const EdgeInsets.all(8.0),
              child: new RaisedButton(
                onPressed: () {
                  // Pop here with "Yep"...
                },
                child: new Text('Yep!'),
              ),
            ),
            new Padding(
              padding: const EdgeInsets.all(8.0),
              child: new RaisedButton(
                onPressed: () {
                  // Pop here with "Nope"
                },
                child: new Text('Nope.'),
              ),
            )
          ],
        ),
      ),
    );
  }
}
```

4. When a button is tapped, close the selection screen

Now, we'll want to update the `onPressed` callback for both of our buttons! In order to return data to the first screen, we'll need to use the `Navitator.pop` method.

`Navigator.pop` accepts an optional second argument called `result`. If we provide a result, it will be returned to the `Future` in our SelectionButton!

Yep button

```
new RaisedButton(
  onPressed: () {
    // Our Yep button will return "Yep!" as the result
    Navigator.pop(context, 'Yep!');
  },
  child: new Text('Yep!'),
);
```

Nope button

```
new RaisedButton(
  onPressed: () {
    // Our Nope button will return "Nope!" as the result
    Navigator.pop(context, 'Nope!');
  },
  child: new Text('Nope!'),
);
```

5. Show a snackbar on the home screen with the selection

Now that we're launching a selection screen and awaiting the result, we'll want to do something with the information that's returned!

In this case, we'll show a Snackbar displaying the result. To do so, we'll update the `_navigateAndDisplaySelection` method in our `SelectionButton`.

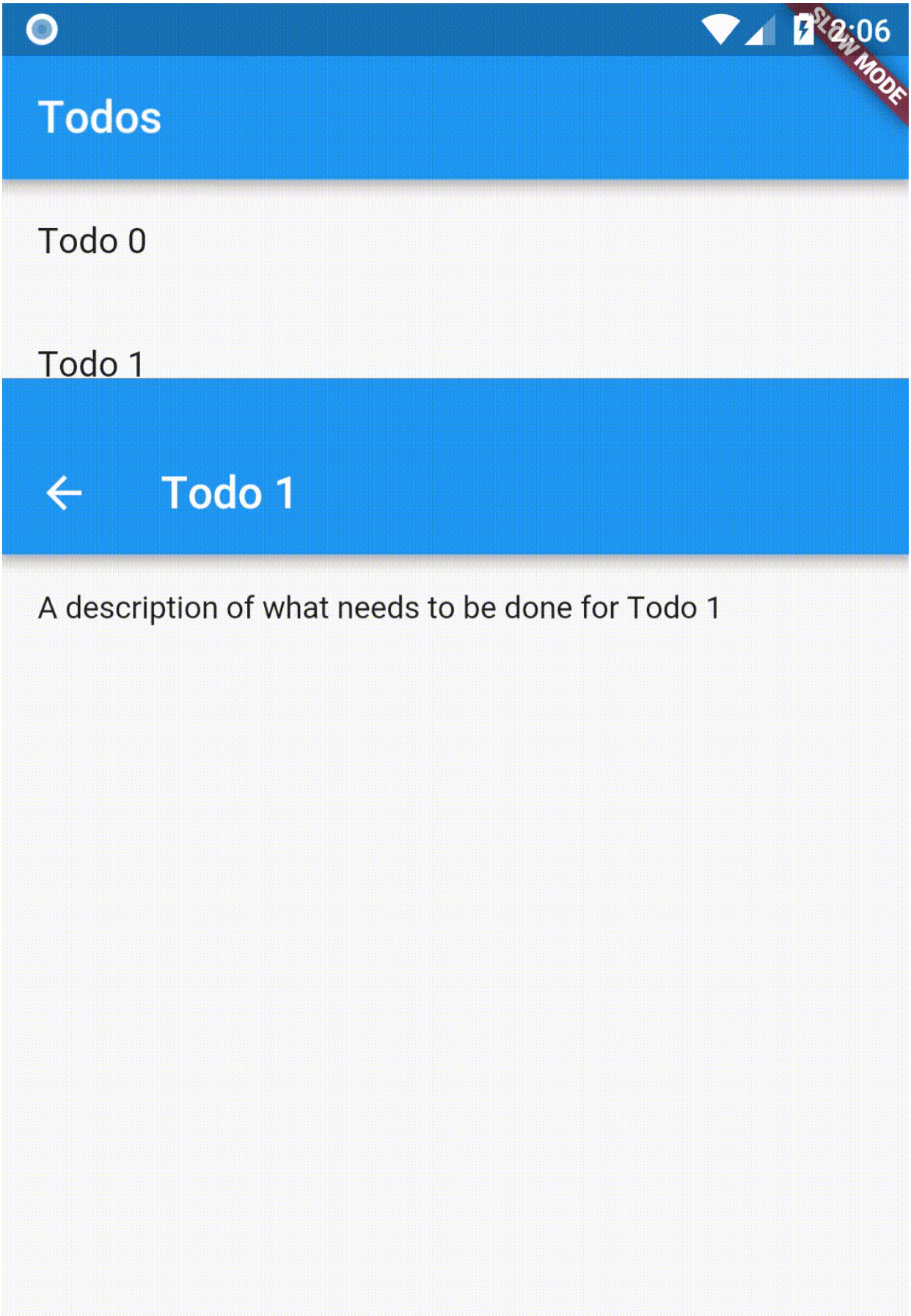
```
_navigateAndDisplaySelection(BuildContext context) async {
  final result = await Navigator.push(
    context,
    new MaterialPageRoute(builder: (context) => new SelectionScreen()),
  );

  // After the Selection Screen returns a result, show it in a Snackbar!
  Scaffold
      .of(context)
      .showSnackBar(new SnackBar(content: new Text("$result")));
}
```

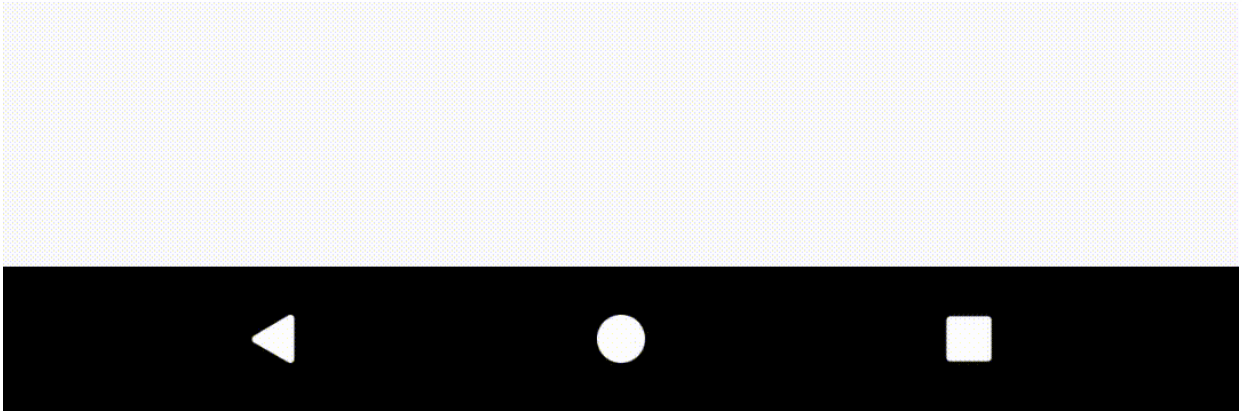
Complete Example

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

void main() {
  runApp(new MaterialApp(
    title: 'Returning Data',
    home: new HomeScreen(),
  ));
}

class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('Returning Data Demo'),
      ),
      body: new Center(child: new SelectionButton()),
    );
  }
}

class SelectionButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new RaisedButton(
      onPressed: () {
        _navigateAndDisplaySelection(context);
      },
      child: new Text('Pick an option, any option!'),
    );
  }

  // A method that launches the SelectionScreen and awaits the result from
```

```
  // Navigator.pop!
  _navigateAndDisplaySelection(BuildContext context) async {
    // Navigator.push returns a Future that will complete after we call
    // Navigator.pop on the Selection Screen!
    final result = await Navigator.push(
      context,
      new MaterialPageRoute(builder: (context) => new SelectionScreen()),
    );

    // After the Selection Screen returns a result, show it in a Snackbar!
    Scaffold
        .of(context)
        .showSnackBar(new SnackBar(content: new Text("$result")));
  }
}

class SelectionScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('Pick an option'),
      ),
      body: new Center(
        child: new Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            new Padding(
              padding: const EdgeInsets.all(8.0),
              child: new RaisedButton(
                onPressed: () {
                  // Close the screen and return "Yep!" as the result
                  Navigator.pop(context, 'Yep!');
                },
                child: new Text('Yep!'),
              ),
            ),
            new Padding(
              padding: const EdgeInsets.all(8.0),
              child: new RaisedButton(
                onPressed: () {
                  // Close the screen and return "Nope!" as the result
                  Navigator.pop(context, 'Nope.');
                },
                child: new Text('Nope.'),
              ),
            )
          ],
        ),
```

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


9:06
SLOW MODE
Todos
Todo 0
Todo 1
Todo 1
A description of what needs to be done for Todo 1



Fetch data from the internet

Fetching data from the internet is necessary for most apps. Luckily, Dart and Flutter provide tools for this type of work!

Directions

1. Make a network request using the `http` package
2. Convert the response into a custom Dart object

1. Make a network request

The `http` package provides the simplest way to fetch data from the internet.

In this example, we'll fetch a sample post from the JSONPlaceholder REST API using the `http.get` method.

```
Future<http.Response> fetchPost() {
  return http.get('https://jsonplaceholder.typicode.com/posts/1');
}
```

The `http.get` method returns a `Future` that contains a `Response`.

- `Future` is
  a core Dart class for working with async operations. It is used to represent a potential value or error that will be available at some time in the future.
- The `http.Response` class contains the data received from a successful http call.

2. Convert the response into a custom Dart object

While it's easy to make a network request, working with a raw `Future<http.Response>` isn't very convenient. To make our lives easier, we can convert the `http.Response` into our own Dart object.

Create a `Post` class

First, we'll need to create a `Post` class that contains the data from our network request. It will also include a factory constructor that allows us to create a `Post` from json.

Converting JSON by hand is only one option. For more information, please see the full article on JSON and serialization.

```
class Post {
  final int userId;
  final int id;
  final String title;
  final String body;

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

  factory Post.fromJson(Map<String, dynamic> json) {
    return new Post(
      userId: json['userId'],
      id: json['id'],
      title: json['title'],
      body: json['body'],
    );
  }
}
```

Convert the `http.Response` to a `Post`

Now, we'll update the `fetchPost` function to return a `Future<Post>`. To do so, we'll need to:

1. Convert the response body into a json `Map` with the `dart:convert` package
2. Convert the json `Map` into a `Post` using the `fromJson` factory.

```
Future<Post> fetchPost() async {
  final response = await
http.get('https://jsonplaceholder.typicode.com/posts/1');
  final json = JSON.decode(response.body);

  return new Post.fromJson(json);
}
```

Hooray! Now we've got a function that we can call to fetch a Post from the internet!

Complete Example

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

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

Future<Post> fetchPost() async {
  final response = await
http.get('https://jsonplaceholder.typicode.com/posts/1');
  final json = JSON.decode(response.body);

  return new Post.fromJson(json);
}

class Post {
  final int userId;
  final int id;
  final String title;
  final String body;

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

  factory Post.fromJson(Map<String, dynamic> json) {
    return new Post(
      userId: json['userId'],
      id: json['id'],
      title: json['title'],
      body: json['body'],
    );
  }
}
```

Making authenticated requests

In order to fetch data from many web services, you need to provide authorization. There are many ways to do this, but perhaps the most common requires using the `Authorization` HTTP header.

Add Authorization Headers

The `http` package provides a convenient way to add headers to your requests. You can also take advantage of the `dart:io` package for common `HttpHeaders`.

```
Future<http.Response> fetchPost() {
  return http.get(
    'https://jsonplaceholder.typicode.com/posts/1',
    // Send authorization headers to your backend
    headers: {HttpHeaders.AUTHORIZATION: "Basic your_api_token_here"},
  );
}
```

Complete Example

This example builds upon the Fetching Data from the Internet recipe.

```
import 'dart:async';
import 'dart:convert';
import 'dart:io';
import 'package:http/http.dart' as http;

Future<Post> fetchPost() async {
  final response = await http.get(
    'https://jsonplaceholder.typicode.com/posts/1',
    headers: {HttpHeaders.AUTHORIZATION: "Basic your_api_token_here"},
  );
  final json = JSON.decode(response.body);

  return new Post.fromJson(json);
}

class Post {
  final int userId;
  final int id;
  final String title;
  final String body;

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

  factory Post.fromJson(Map<String, dynamic> json) {
    return new Post(
      userId: json['userId'],
      id: json['id'],
      title: json['title'],
      body: json['body'],
    );
  }
}
```

Working with WebSockets

In addition to normal HTTP requests, we can connect to servers using WebSockets. WebSockets allow for two-way communication with a server without polling.

In this example, we'll connect to a [test server provided by websocket.org] (http://www.websocket.org/echo.html). The server will simply send back the same message we send to it!

Directions

1. Connect to a WebSocket server
2. Listen for messages from the server
3. Send Data to the Server
4. Close the WebSocket connection

1. Connect to a WebSocket server

The web_socket_channel package provides the tools we'll need to connect to a WebSocket server.

The package provides a `WebSocketChannel` that allows us to both listen for messages from the server as well as push messages to the server.

In Flutter, we can create a `WebSocketChannel` that connects to a server in one line:

```
final channel = new IOWebSocketChannel.connect('ws://echo.websocket.org');
```

2. Listen for messages from the server

Now that we've established a connection, we can listen to messages from our server.

After we send a message to the test server, it will send the same message back.

How how do we listen for messages and display them? In this example, we'll use a `StreamBuilder` Widget to listen for new messages and a `Text` Widget to display them.

```
new StreamBuilder(
  stream: widget.channel.stream,
  builder: (context, snapshot) {
    return new Text(snapshot.hasData ? '${snapshot.data}' : '');
  },
);
```

How does this work?

The `WebSocketChannel` provides a `Stream` of messages from the server.

The `Stream` class is a fundamental part of the `dart:async` package. It provides a way to listen to async events from a data source. Unlike `Future`, which returns a single async response, the `Stream` class can deliver many events over time.

The `StreamBuilder` Widget will connect to a `Stream` and ask Flutter to rebuild every time it receives an event using the given `builder` function!

3. Send Data to the Server

In order to send data to the server, we'll `add` messages to the `sink` provided by the `WebSocketChannel`.

```
channel.sink.add('Hello!');
```

How does this work

The `WebSocketChannel` provides a `StreamSink` to push messages to the server.

The `StreamSink` class provides a general way to add sync or async events to a data source.

4. Close the WebSocket connection

After we're done using the WebSocket, we'll want to close the connection! To do so, we can close the `sink`.

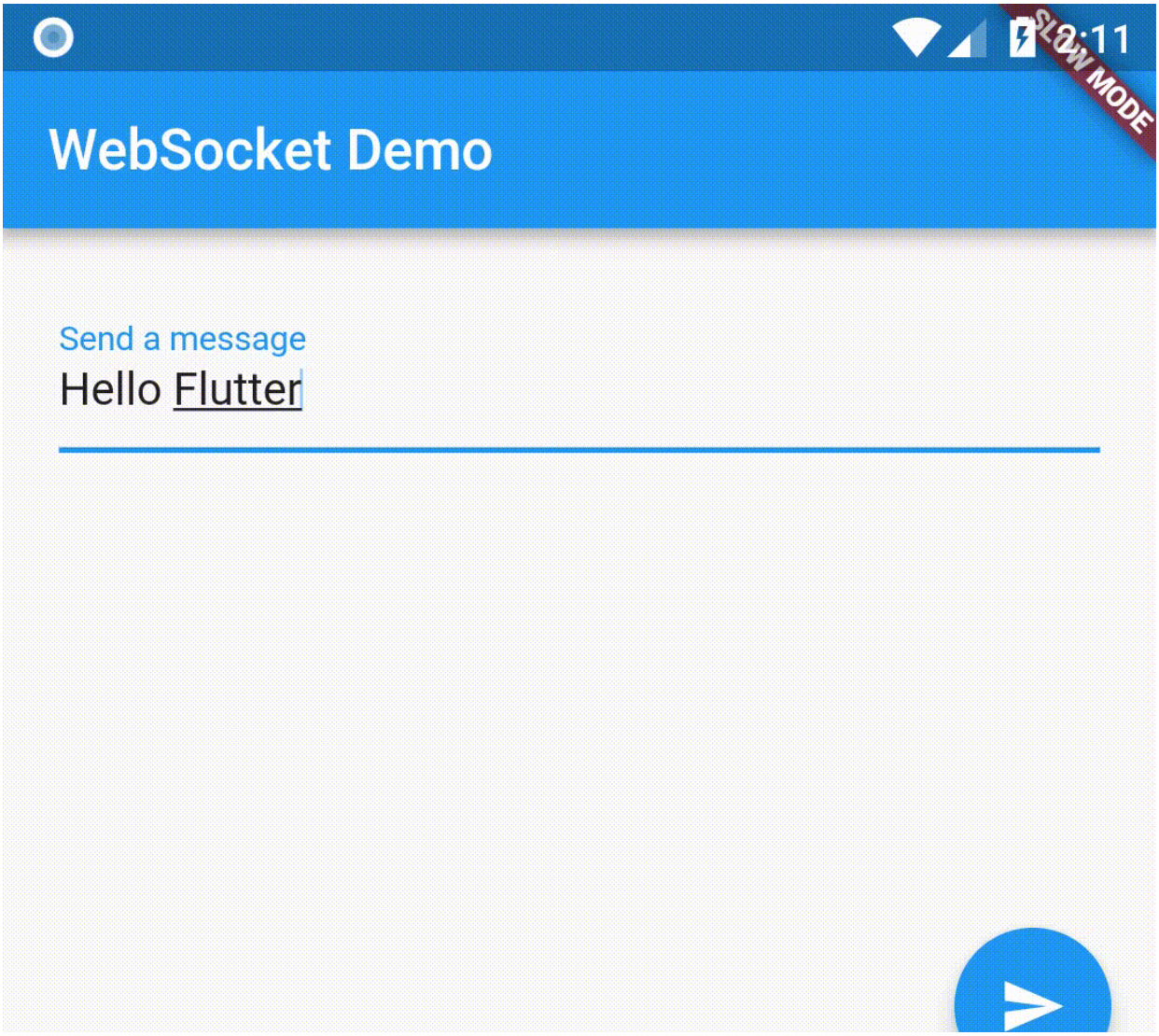
```
channel.sink.close();
```

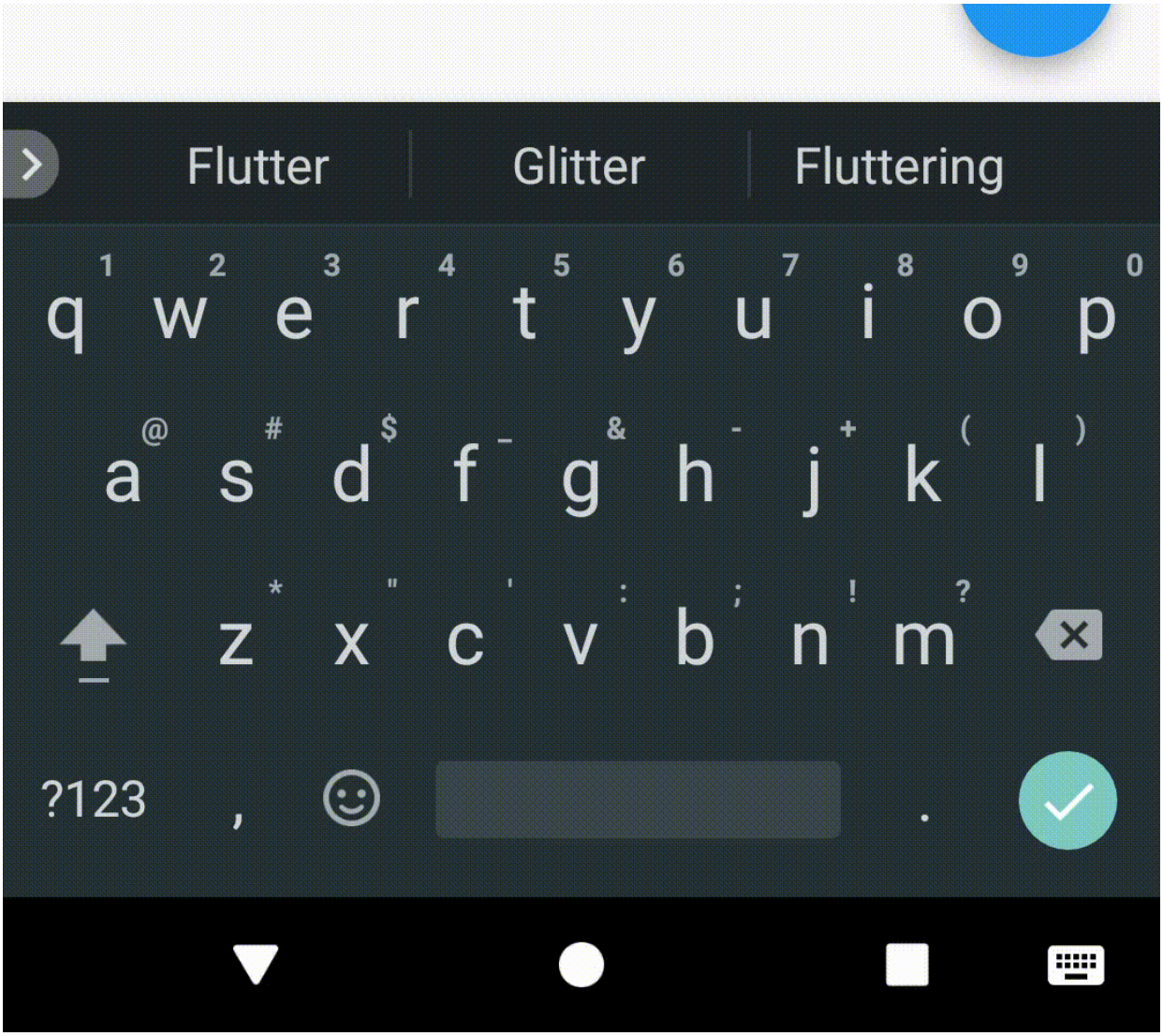
Complete Example

```
import 'package:flutter/foundation.dart';
import 'package:web_socket_channel/io.dart';
import 'package:flutter/material.dart';
import 'package:web_socket_channel/web_socket_channel.dart';

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'WebSocket Demo';
    return new MaterialApp(
      title: title,
      home: new MyHomePage(
        title: title,
        channel: new IOWebSocketChannel.connect('ws://echo.websocket.org'),
      ),
```

```
    );
  }
}

class MyHomePage extends StatefulWidget {
  final String title;
  final WebSocketChannel channel;

  MyHomePage({Key key, @required this.title, @required this.channel})
      : super(key: key);

  @override
  _MyHomePageState createState() => new _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  TextEditingController _controller = new TextEditingController();

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(widget.title),
      ),
      body: new Padding(
        padding: const EdgeInsets.all(20.0),
        child: new Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: <Widget>[
            new Form(
              child: new TextFormField(
                controller: _controller,
                decoration: new InputDecoration(labelText: 'Send a
message'),
              ),
            ),
            new StreamBuilder(
              stream: widget.channel.stream,
              builder: (context, snapshot) {
                return new Padding(
                  padding: const EdgeInsets.symmetric(vertical: 24.0),
                  child: new Text(snapshot.hasData ? '${snapshot.data}' :
''),
                );
              },
            )
          ],
        ),
      ),
```

```
      floatingActionButton: new FloatingActionButton(
        onPressed: _sendMessage,
        tooltip: 'Send message',
        child: new Icon(Icons.send),
      ), // This trailing comma makes auto-formatting nicer for build
methods.
    );
  }

  void _sendMessage() {
    if (_controller.text.isNotEmpty) {
      widget.channel.sink.add(_controller.text);
    }
  }

  @override
  void dispose() {
    widget.channel.sink.close();
    super.dispose();
  }
}
```




Flutter
Glitter
Fluttering
?123