

Flutter & Dart Complete Guide

Prepared by Eng. Yousef Salah Kassem

DAFL 2
Responsive

Lecture 6



Table Of Content

- Localization
- Expands & Flexible
- Media Query
- Responsive Layout Builder



Localizations

Chapter 06

Localizations

Why easy_localization?

- 🚀 Easy translations for many languages
- 📁 Load translations as JSON, CSV, Yaml, Xml using [Easy Localization Loader](#)
- 💾 React and persist to locale changes
- ⚡ Supports plural, gender, nesting, RTL locales and more
- 🔄 Fallback locale keys redirection
- ❓ Error widget for missing translations
- ❤️ Extension methods on `Text` and `BuildContext`
- 💻 Code generation for localization files and keys.
- 🛡️ Null safety
- 📄 Customizable logger.



Chapter
06

Localizations

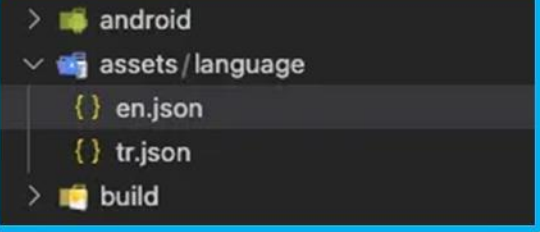
Then create a folder and add translation files.

en.json {

```
"title" : "Title",  
}
```

tr.json {

```
"title" : "Başlık",  
}
```



```
> android  
✓ assets / language  
  {} en.json  
  {} tr.json  
> build
```



Chapter
06

Localizations

Add assets directory in pubspec.yaml:

```
flutter:
```

```
  assets:
```

```
    - assets/language/
```

For iOS you need to add supported locales to ios/Runner/Info.plist as described

```
<key>CFBundleLocalizations</key>
```

```
<array>
```

```
  <string>en</string>
```

```
  <string>tr</string>
```

```
</array>
```



Chapter
06

Localizations

run this scripts:

```
flutter pub run easy_localization:generate -S assets/translations -f keys -o locale_keys.g.dart
flutter pub run easy_localization:generate -S assets/translations
echo -e "// ignore_for_file: constant_identifier_names\n$(cat ./lib/generated/locale_keys.g.dart)" >
./lib/generated/locale_keys.g.dart
```



Chapter 06

Localizations

Change the void main() part like this.

```
Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await EasyLocalization.ensureInitialized();
  runApp(
    EasyLocalization(
      supportedLocales: const [Locale('en'), Locale('tr')],
      path: 'assets/language',
      fallbackLocale: const Locale('en'),
      child: const MyApp(),
    ), // EasyLocalization
  );
}
```

```
return MaterialApp(
  debugShowCheckedModeBanner: false,
  title: 'Easy Localization Demo',
  localizationsDelegates: context.localizationDelegates,
  supportedLocales: context.supportedLocales,
  locale: context.locale,
  theme: ThemeData(primarySwatch: Colors.blue,),
  home: HomePage(),
); // MaterialApp
```



Chapter 06

Localizations

```
class _HomePageState extends State<HomePage> {
  bool isEN = true;

  void changeLanguage() {
    setState(() {
      isEN = !isEN;
      isEN ? context.setLocale(const Locale('en')) : context.setLocale(const Locale('tr'));
      debugPrint(isEN.toString());
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Easy Localization'),
        centerTitle: true,
      ), // AppBar
      body: Center(
        child: Column(
          children: [
            Text(
              'title'.tr(),
              style: Theme.of(context).textTheme.headline5,
            ), // Text
            Text('demoText'.tr()),
            ElevatedButton(
              onPressed: () {
                changeLanguage();
              },
              child: Text('buttonText'.tr()), // ElevatedButton
            ),
          ],
        ), // Column
      ), // Center
    ); // Scaffold
  }
}
```



Expands & Flexible

Chapter
06

Expanded & Flexible

Using an [Expanded](#) widget makes a child of a [Row](#) or [Column](#) (also for [Flex](#)) expand to fill the available space in the main axis (horizontally for a [Row](#) or vertically for a [Column](#)). If multiple children are expanded, the available space is divided among them according to the [flex](#) factor.

For example we have a column with 3 container including a letter.



```
class Example extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Row(  
      children: <Widget>[  
        Container(  
          padding: EdgeInsets.all(10),  
          color: Colors.blueGrey,  
          child: Text('A'),  
        ), // Container  
        Container(  
          padding: EdgeInsets.all(10),  
          color: Colors.yellow,  
          child: Text('B'),  
        ), // Container  
        Container(  
          padding: EdgeInsets.all(10),  
          color: Colors.red,  
          child: Text('C'),  
        ) // Container  
      ], // <Widget>[]  
    ); // Row  
  }  
}
```



Chapter
06

Expanded & Flexible

Normally Row and Column divide the free space according the size of the children widgets. Children are inflexible. Also this space usage depend on the padding area or mainAxisAlignment property ([Row and Column Usage](#))

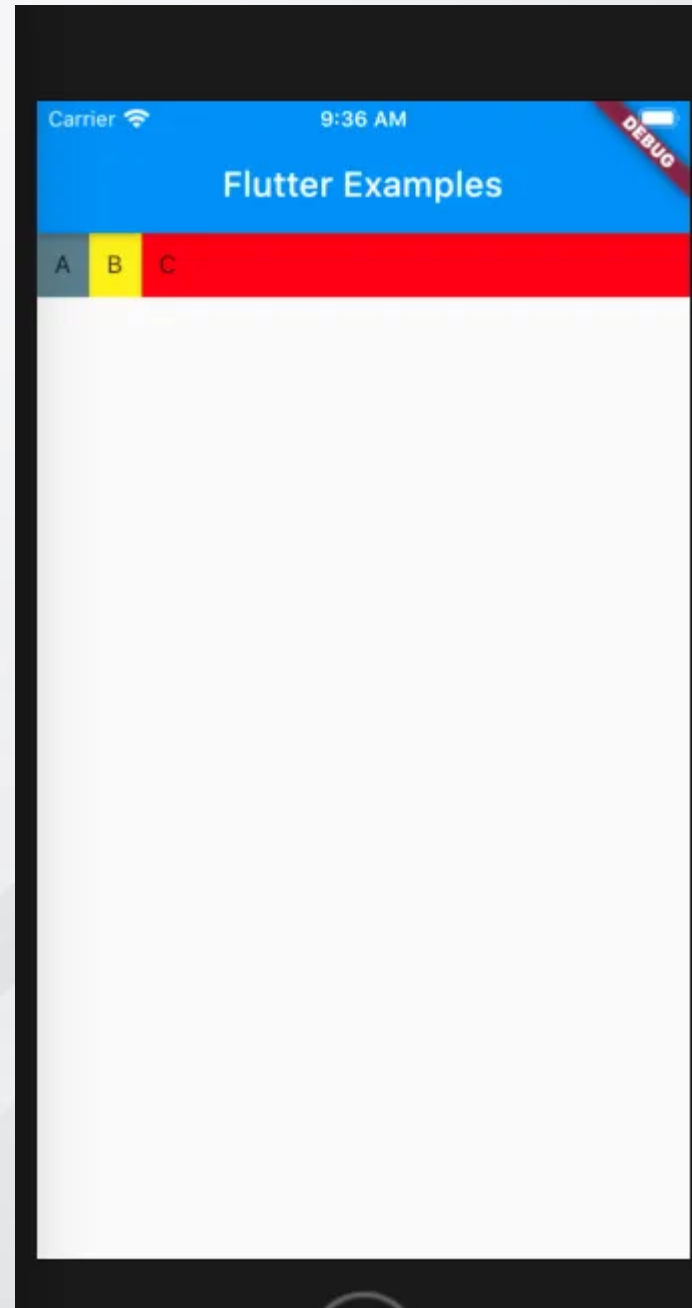
But if you want some of the children stretch and fill the extra space then you should use Expanded widget (Wrap the children with Expanded widget) In this situation child become flexible.



```
class Example extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Row(  
      children: <Widget>[  
        Container(  
          padding: EdgeInsets.all(10),  
          color: Colors.blueGrey,  
          child: Text('A'),  
        ), // Container  
        Container(  
          padding: EdgeInsets.all(10),  
          color: Colors.yellow,  
          child: Text('B'),  
        ), // Container  
        Expanded(  
          child: Container(  
            padding: EdgeInsets.all(10),  
            color: Colors.red,  
            child: Text('C'),  
          ), // Container  
        ), // Expanded  
      ], // <Widget>[]  
    ); // Row  
  }  
}
```



```
class Example extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Row(  
      children: <Widget>[  
        Container(  
          padding: EdgeInsets.all(10),  
          color: Colors.blueGrey,  
          child: Text('A'),  
        ), // Container  
        Container(  
          padding: EdgeInsets.all(10),  
          color: Colors.yellow,  
          child: Text('B'),  
        ), // Container  
        Expanded(  
          child: Container(  
            padding: EdgeInsets.all(10),  
            color: Colors.red,  
            child: Text('C'),  
          ), // Container  
        ), // Expanded  
      ], // <Widget>[]  
    ); // Row  
  }  
}
```



Chapter
06

Expanded & Flexible

Normally Row and Column divide the free space according the size of the children widgets. Children are inflexible. Also this space usage depend on the padding area or mainAxisAlignment property ([Row and Column Usage](#))

But if you want some of the children stretch and fill the extra space then you should use Expanded widget (Wrap the children with Expanded widget) In this situation child become flexible.



Chapter
06

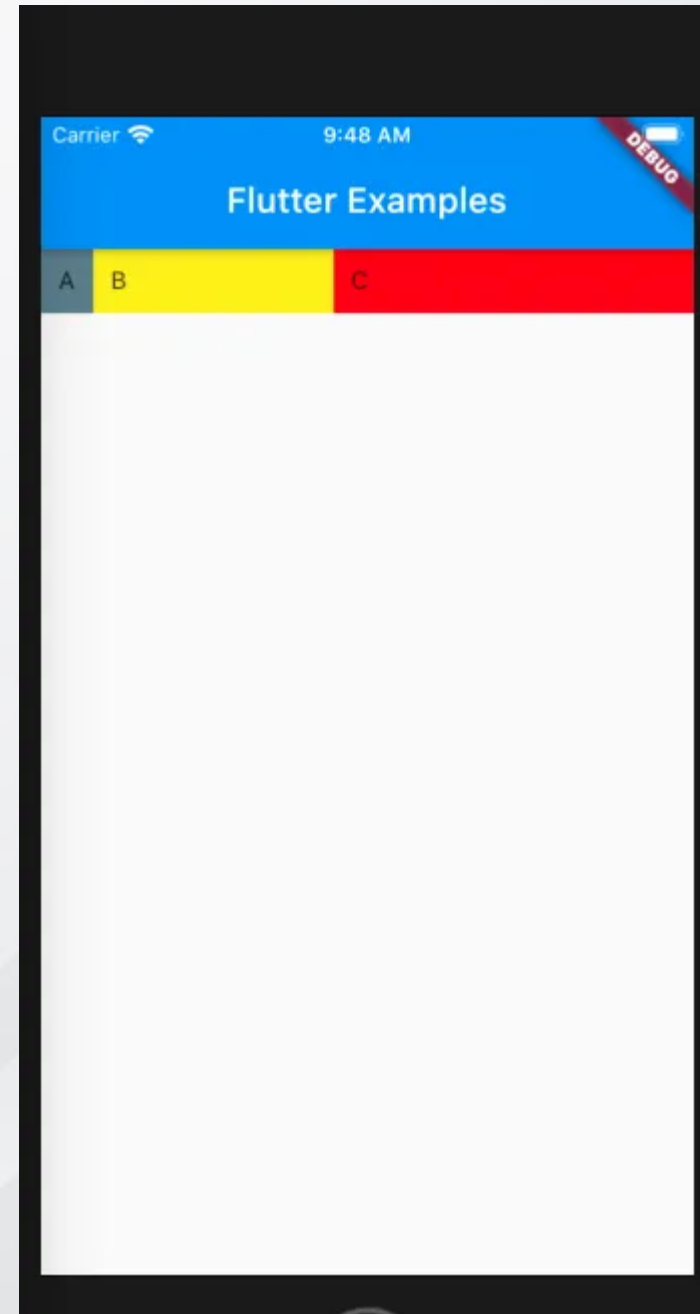
Expanded & Flexible

Without giving any extra property like flex, Expanded widgets use all remaining space equally. If one child is expanded use all free space, if there are two or more children are expanded, they share all free space equally.(Default flex:1)

If you want to give some scale to share the free space, you should use flex factor. Flex factor is a property of Expanded widget and determines the usage and sharing of the remaining space between children widgets.



```
class Example extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Row(  
      children: <Widget>[  
        Container(  
          padding: EdgeInsets.all(10),  
          color: Colors.blueGrey,  
          child: Text('A'),  
        ), // Container  
        Expanded(  
          flex: 2,  
          child: Container(  
            padding: EdgeInsets.all(10),  
            color: Colors.yellow,  
            child: Text('B'),  
          ), // Container  
        ), // Expanded  
        Expanded(  
          flex: 3,  
          child: Container(  
            padding: EdgeInsets.all(10),  
            color: Colors.red,  
            child: Text('C'),  
          ), // Container  
        ), // Expanded  
      ], // <Widget>[]  
    ); // Row  
  }  
}
```



Chapter
06

Expanded & Flexible

In fact, Expanded widget is extended from Flexible widget. Expanded is a shorthand for Flexible with tight fit. To clarify the issue, you can find the code of the Expanded class in flutter below. What you can see it extends the Flexible class with fit property FlexFit.tight.

Expanded is a special usage of the Flexible Widget

```
class Expanded extends Flexible {  
  
  // Creates a widget that expands a child of a [Row], [Column], or [Flex]  
  
  /// so that the child fills the available space along the flex widget's main axis.  
  
  const Expanded({  
  
    Key key,  
  
    int flex = 1,  
  
    @required Widget child,  
  
  }) : super(key: key, flex: flex, fit: FlexFit.tight, child: child);  
  
}
```



Flexible vs Expanded	
Expanded	Flexible
Expanded	Expanded
Flexible(fit: FlexFit.tight	Flexible (fit: Fle xFit.loo
Flexible(fit: FlexFit.tight	Expanded

Media Query

Chapter
06

Media Query

In Flutter, **MediaQuery** is a widget that provides information about the device's screen and user preferences. It allows developers to build responsive and adaptive user interfaces by giving access to various details about the screen's size, orientation, pixel density, and more. How, you ask?

Just use,

```
//Use this anywhere in the Widget tree where BuildContext context is available  
MediaQuery.of(context);
```



Chapter
06

Media Query

Here is the thing, every widget in flutter application(Stateless, Stateful or Inherited Widget) has a [build\(\)](#) method which is **used to build a widget** and is a **necessary override** while creating any widget. This build method takes a **required argument** which is (*drumroll...*), **BuildContext**.

Now, since **BuildContext** is available to all the widgets inside the **build()** method and we can use the `MediaQuery.of()` method and pass the `BuildContext` to it.

We will typically write something like this in order to create a Widget and use `MediaQuery.of(context)` inside the **`build(BuildContext context)`** method where the **BuildContext** is available.



Chapter
06

Media Query

```
//Ex
//MediaQuery being used inside a widget

class MyCustomWidget extends StatelessWidget {
  const MyCustomWidget({super.key});
  @override
  Widget build(BuildContext context) {
    //this MediaQueryData instance can be only used
    //inside the build method where the context is available
    final MediaQueryData mediaQuery = MediaQuery.of(context);

    return Scaffold(
      body: Center(
        child: Container(
          width: mediaQuery.size.width * 0.5, //50% of screen width
          height: mediaQuery.size.height * 0.5, //50% of screen height
          color: Colors.blue,
        ),
      ),
    );
  }
}
```



Chapter
06

Media Query

Media Query has multiple data members(properties) that contain information about device specific information like devicePixelRatio, orientation, padding, size, etc. Let's go through all such important data members.

Note: *I will be covering the important and most used data members and not all of them. Consider [reading the docs](#) for a detailed info*



Chapter
06

Media Query

Data members containing information about “*Device Screen*”

1. ***devicePixelRatio***: Device Pixel Ratio or DPR refers to the ratio between the physical pixels on screen and logical pixels. Physical pixels are the actual pixels present on the display of the device, whereas one logical pixel corresponds to ‘n’ number of physical pixels. We can say something like **1 logical pixel** correspond to **3 physical pixels**. If DPR for a device is 3 then, 3 physical pixels will be used to draw a line which is 1px wide. Logical Pixels are same across various devices and are used to maintain “visual consistency”. DPR is mostly used in dealing with images with different resolutions to maintain consistency. This is how to use ***devicePixelRatio***

```
final double dpr = MediaQuery.of(context).devicePixelRatio;
```



Chapter
06

Media Query

2. **padding**: Padding in context of Media Query refers to the amount of space reserved by operating system like the system status bar(the one where we see the time and battery percentage, etc.) and the bottom region which is used for navigation and other purposes which may vary from one platform to other. Padding is of type [EdgeInsets](#) and has four offsets, namely bottom, top, right and left. Bottom padding will have some value if the keyboard is not visible, if keyboard is Up then bottom padding will be reduced to "0". This is how to use **padding**

```
final EdgeInsets padding = MediaQuery.of(context).padding;
```

```
debugPrint("Padding => Bottom: ${padding.bottom}, Top: ${padding.top}, Left: ${padding.left}, Right: ${padding.right}");
```



Chapter
06

Media Query

orientation: Orientation is as simple as it gets. It refers to the **alignment and positioning** of the device. There are two types of orientations, **Landscape** and **Potrait**. In Potrait orienatation the width of device is smaller than the height and in Landscape orientation it just gets reversed, now the width is larger and height is smaller. Orientation plays a crucial role in determining the layout of the application, as changing the orientation affects the amount of information or data that can be displayed on the screen. A good application is one which accounts for different orientations. This is how to use **orientation**

```
final Orientation orientation= MediaQuery.of(context).orientation;
if(orientation == Orientation.landscape){
  debugPrint("Device is in Landscape mode");
}else{
  debugPrint("Device is in Potrait mode");
}
```



Chapter
06

Media Query

4. **size**: Size is the most used data member offered by the MediaQueryData class and provides a lot of information about the device screen. But we will be talking about three, namely **width**, **height** and **aspectRatio**. As the name suggests width gives the **device width**, height gives the **device height** and aspectRatio gives the **ratio of device width to device height**.

This is how to use **size**

```
final Size size = MediaQuery.of(context).size;
```

```
debugPrint("Size => Width: ${size.width}, Height: ${size.height}, AspectRatio: ${size.aspectRatio}");
```



Chapter
06

Media Query

5. ***viewInsets*** : View Insets is of type [EdgeInsets](#). ***viewInsets*** is used to determine the amount of space consumed by the keyboard when it is visible. If the keyboard is not visible the all the four offsets of viewInsets i.e bottom, top, right and left will be "0". View Insets bottom offset will be non zero when the keyboard is up and every other offset will be zero (always). This is how to use ***viewInsets***

```
final EdgeInsets viewInsets = MediaQuery.of(context).viewInsets;
```

```
if(viewInstes.bottom > 0){
```

```
    debugPrint("Keyboard is UP");
```

```
}
```



Chapter
06

Media Query

5. *platform Brightness*: Platform Brightness is the property which developers will find using often. It tells whether user has enabled light theme or dark theme. platform Brightness returns an enum called **Brightness** which has two values **light** and **dark**. It is common for users to change the system theme mode based on time of the day or just personal preference, so accounting for such changes is must. This is how to use *platform Brightness*

```
final Brightness platformBrightness = MediaQuery.of(context).platformBrightness;
if(platformBrightness == Brightness.dark){
  debugPrint("System is using dark mode");
} else{
  debugPrint("System is using light mode");}
return MaterialApp(
  theme: ThemeData.light(), //light theme
  darkTheme: ThemeData.dark(), //dark theme
  themeMode: ThemeMode.system,
  home: MyHomePage(),
);
```



Responsive Layout Builder

Chapter
06

Responsive

Creating Responsive layouts using Media Query.

Now, we have come a long way exploring Media Query and are familiar with its in's and out's. This is the right time to discuss about creating responsive designs and the role of Media Query in creating responsive apps.

Media Query tells us about the device screen which is basically everything that is required to build a responsive layout. What are you waiting for, just start using Media Query and create responsive layouts.



Chapter
06

Responsive

knowing what MediaQuery is and utilizing MediaQuery for creating Responsive Layouts are flip sides of the same coin. It is essential understanding both of them to create what is called as a “***Responsive App***”, one that is **visually consistent**, offers **optimal user experience** a cross **wide range of devices**, **adapts** and **uses the available space in best possible way**, so on and so forth. Covering all of that in a single blog post is not possible, but don't worry as this is just the first part and **next parts** will make it way more clear.

Now, since we have included the section about responsive layouts in the blog post, lets see a tiny example that will demonstrate the power of MediaQuery in creating responsive app.



Chapter
06

Responsive

Let's say you are coding a application and it will be used on smartphones, tablets, desktop and web as well. That's a lot of screens to account for. How do we know which device screen the app is been used on, and how can MediaQuery help in this situation?

You can use something like this which determines the type of layout based on screen width



Chapter
06

Responsive

```
double screenWidth = MediaQuery.of(context).size.width;  
double screenHeight = MediaQuery.of(context).size.height;  
  
if (screenWidth >= 1024) {  
  // Desktop layout  
  return buildDesktopLayout();  
} else if (screenWidth >= 600) {  
  // Tablet layout  
  return buildTabletLayout();  
} else {  
  // Mobile layout  
  return buildMobileLayout();  
}
```

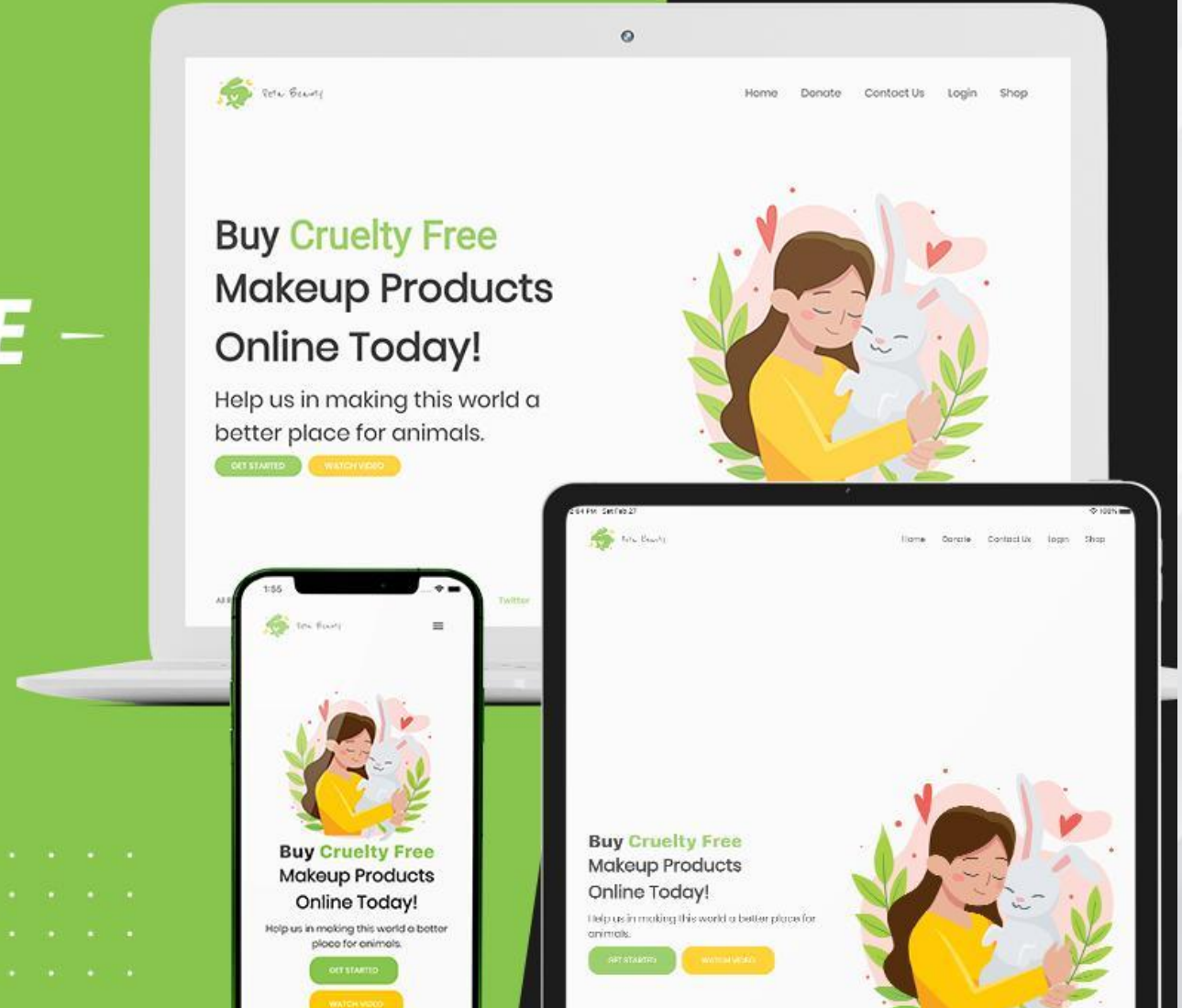


Live Coding

FLUTTER
WEB

RESPONSIVE –
DESIGN

IN 10MIN



What is Next ?



Thank You