

Biomedical Wearable Technologies for Healthcare and Wellbeing

Hello, Flutter!

A.Y. 2021-2022

Giacomo Cappon



Outline

- Recap
- Creating a new project
- App dissection
- Expanding our first app

- Homework & Resources

- Final project overview

Outline

- **Recap**
- Creating a new project
- App dissection
- Expanding our first app
- Homework & Resources
- Final project overview

Flutter

➤ What is Flutter?

- Simply a declarative framework for Dart

➤ Why this choice?

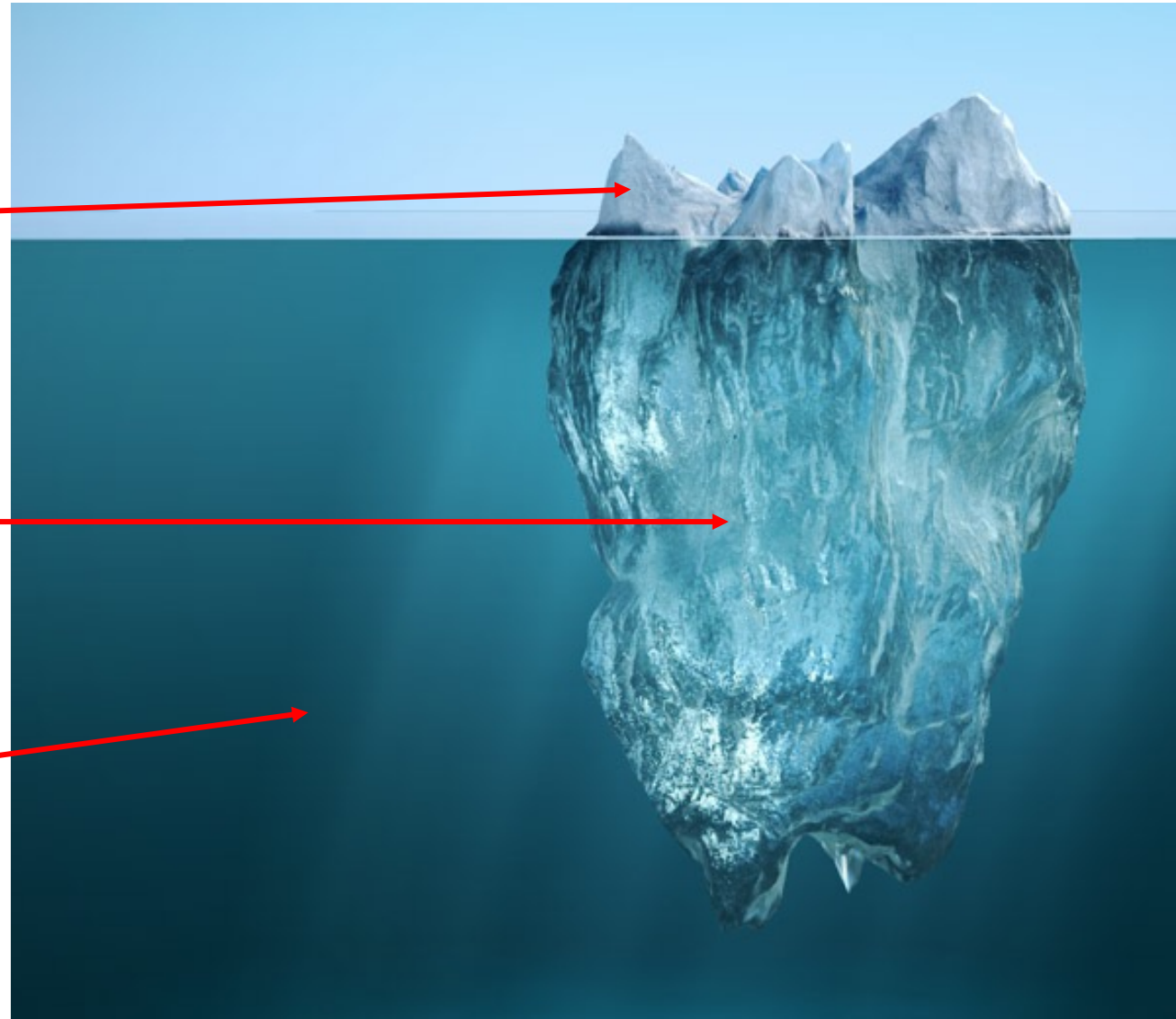
- State-of-the-art and Google-maintained
- Single codebase for iOS and Android (and Mac, Windows, Web)
- Relatively easy to learn
- Lots of examples
- Fastly growing job market

➤ Today we will create and study our first Flutter app



Before starting...

- What you'll see in these labs about Flutter examples
- Actual things that you will probably use
- Flutter possibilities



Outline

- Recap
- **Creating a new project**
- App dissection
- Expanding our first app

- Homework
- Resources

- Final project overview

Hello, Flutter!

- In this lesson, we will run and analyse our first Flutter app
- First, setup VS Code to work with Flutter
 1. Start VS Code.
 2. Invoke **View > Command Palette....**
 3. Type “install”, and select **Extensions: Install Extensions**.
 4. Type “flutter” in the extensions search field, select **Flutter** in the list, and click **Install**. This also installs the required Dart plugin.
- Then, create the app
 1. Invoke **View > Command Palette**.
 2. Type “flutter”, and select the **Flutter: New Project**.
 3. Select **Application**
 4. Select the parent directory that will contain the app
 5. Enter a project name, such as “my_first_app”, and press **Enter**.
 6. Wait for project creation to complete and the main.dart file to appear.

Hello, Flutter!

- Replace all the code of main.dart with

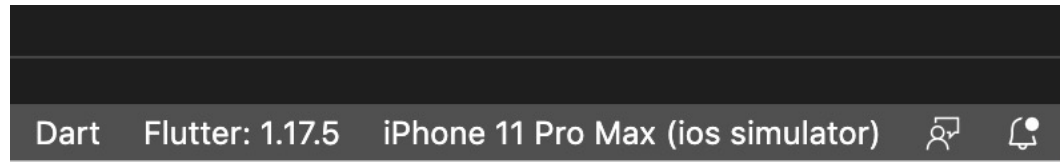
```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
} //main
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(title: const Text('Welcome to Flutter')),
        body: const Center(child: Text('Hello World'))),
    );
  } //build
} //MyApp
```

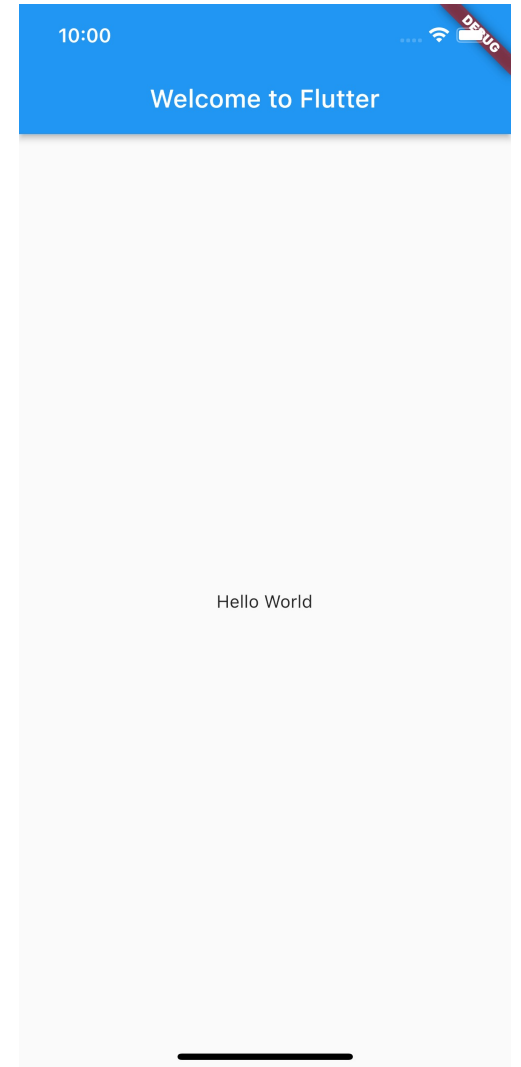

Hello, Flutter!

➤ Finally, run the app!

1. Locate the VS Code status bar (the blue bar at the bottom of the window):



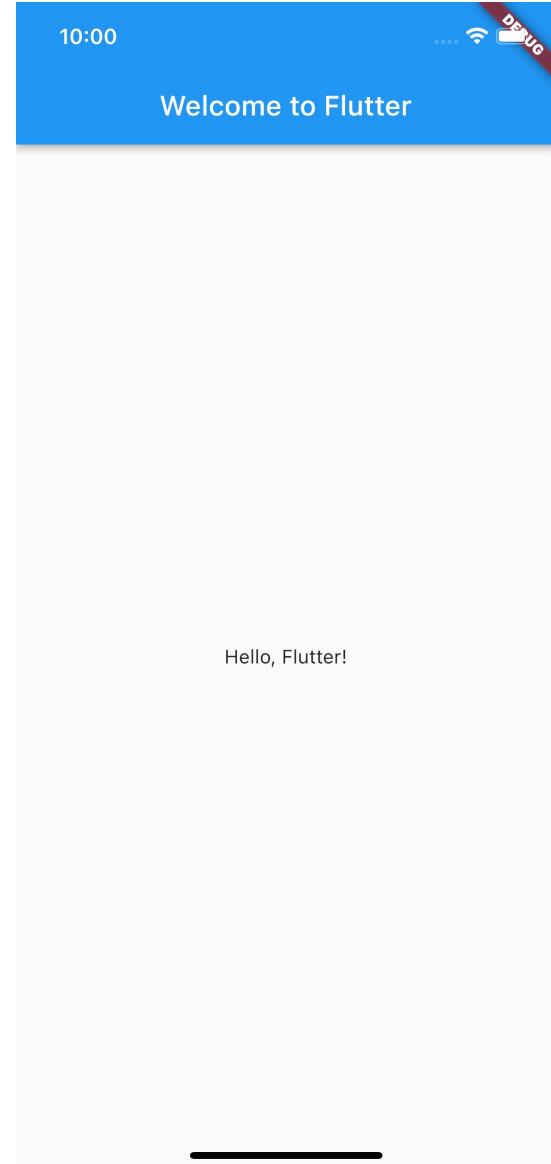
2. Select a mobile device from the **Device Selector** area
3. Invoke **Run > Start Debugging** or press **F5**
4. Wait for the app to launch — progress is printed in the **Debug Console** view.
5. After the app build completes, you'll see the starter app on your device.



A great feature: Hot reload

- Dart offers a fast development cycle with *Stateful Hot Reload*, the ability to reload the code of a live running app without restarting or losing app state. Make a change to app source, tell your IDE or command-line tool that you want to hot reload, and see the change in your simulator, emulator, or device.

- Try that!
 1. Open lib/main.dart.
 2. Change the string
 'Hello World'
 with
 'Hello, Flutter!'
 3. Save your changes: invoke **Save All**, or click **Hot Reload**
 4. You'll see the updated string in the running app almost immediately.



Outline

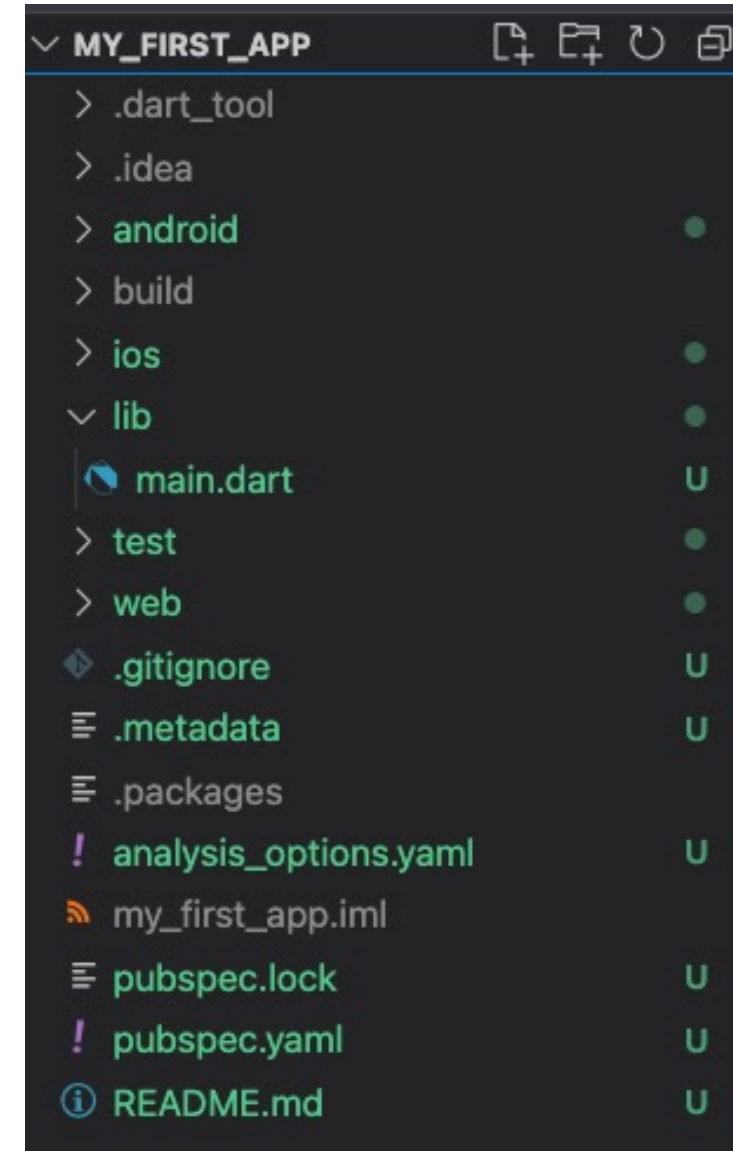
- Recap
- Creating a new project
- **App dissection**
- Expanding our first app
- Homework & Resources
- Final project overview

Let's dissect the app

- Let's understand what we have done.

Let's dissect the app – Project folder

- First, what's inside the project folder?
- Important things
 - **lib folder:** it contains the app source code
 - **main.dart file:** the entry point for the compiler
 - **pubspec.yaml file:** it specifies high level app features as well as listing which third party libraries our app needs and uses
 - **README.md file:** a markdown file describing the app
- (Less) Important things
 - **android/ios/web folders:** where native specific code can be defined if needed
 - **test folder:** where to put code for running automatic testers
- (Even less) Important things
 - All other folders and files are very use case specific and probably you will never use those in this course. If you are curious...



Let's dissect the app – main.dart

➤ Let's understand the main.dart file.

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
} //main
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

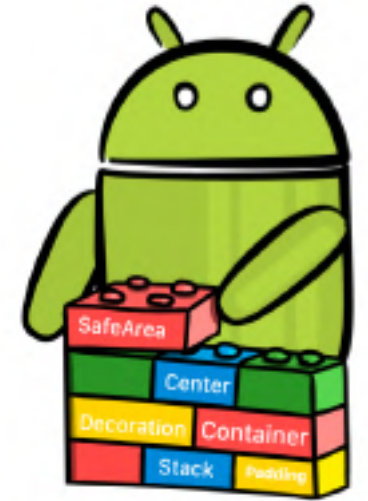
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(title: const Text('Welcome to Flutter')),
        body: const Center(child: Text('Hello World'))),
    );
  } //build
} //MyApp
```

To run an app using the Flutter framework we can use the **runApp** method which takes a **Widget** object as an input.

What's a **Widget**?

Everything is a Widget

- In Flutter, almost everything is (inherits from) a Widget!
- A Widget is a building block for your user interface (UI). Using widgets is like combining Legos.
- More technically, a Widget is a sort of blueprint for displaying your app state.
- Widgets can be thought as a function of UI. Given a state, the build() method (that every custom Widget must override and implement) constructs the widget UI:



$$\text{UI} = f(\text{state})$$

Screen build

Let's dissect the app – main.dart

➤ In **bold** the Widgets of our app

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
} //main
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(title: const Text('Welcome to Flutter'),),
        body: const Center(child: Text('Hello World'),),),
    );
  } //build
} //MyApp
```

Key method for building the Widget that must be implemented

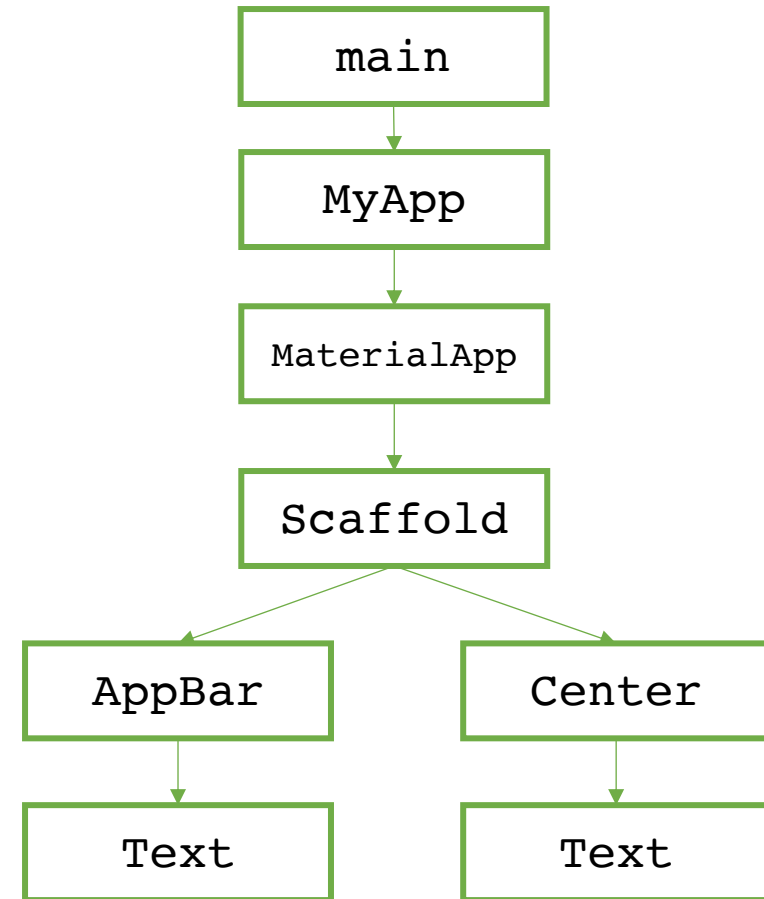
But how Widgets are combined together?

The Widget Tree

- Widgets are combined together using a **tree structure**

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
} //main
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(title: const Text('Welcome to Flutter'),),
        body: const Center(child: Text('Hello World'),),),
    );
  } //build
} //MyApp
```



State and widgets

➤ In bold the Widgets of our app

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
} //main
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(title: const Text('Welcome to Flutter'),),
        body: const Center(child: Text('Hello World'),),),
    );
  } //build
} //MyApp
```

MyApp is not just a Widget,
it is a StatelessWidget

Stateless vs. Stateful widgets

- **StatelessWidgets** are Widgets that always build the same way given a particular configuration and ambient state. So, they never re-build while they are displayed to the user (their lifetime).
- **StatefulWidget** for widgets that can build differently several times over their lifetime.
- You can think about StatelessWidget as a sort of constant and StatefulWidget as a variable.

Let's dissect the app – pubspec.yaml

- pubspec.yaml contains high-level instructions for the development environment and information on the app

```
name: my_first_app
description: A new Flutter project.
publish_to: 'none'
version: 1.0.0+1
```

my_first_app information (name, description, version, ...)

```
environment:
  sdk: ">=2.15.1 <3.0.0"
```

Flutter sdk version to be used

```
dependencies:
  flutter:
    sdk: flutter

  cupertino_icons: ^1.0.2
```

App dependencies: what the app needs in order to work: other packages? Other libraries? Put them here.

```
dev_dependencies:
  flutter_test:
    sdk: flutter
  flutter_lints: ^1.0.0
```

App dependencies while developing the app

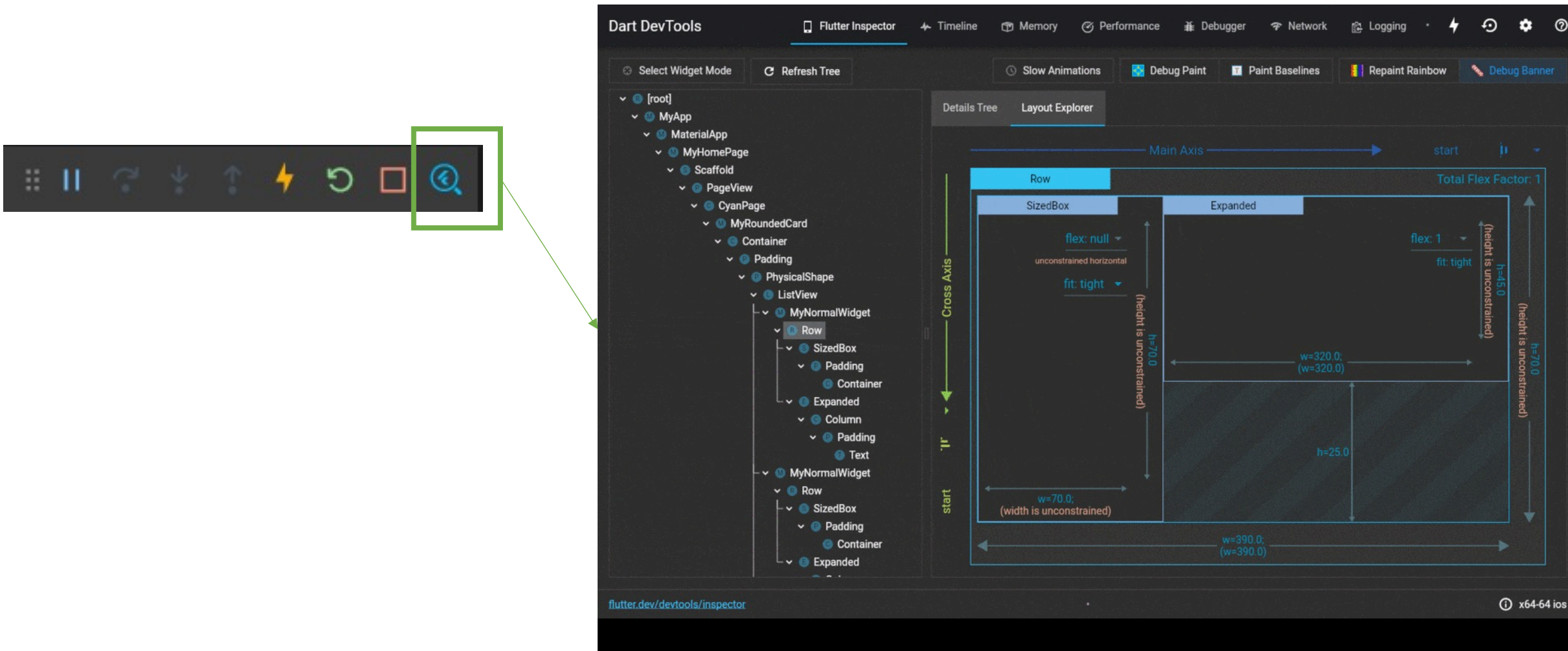
```
flutter:
  uses-material-design: true
```

Information for the Flutter environment such as where to find assets.

DevTools

BONUS

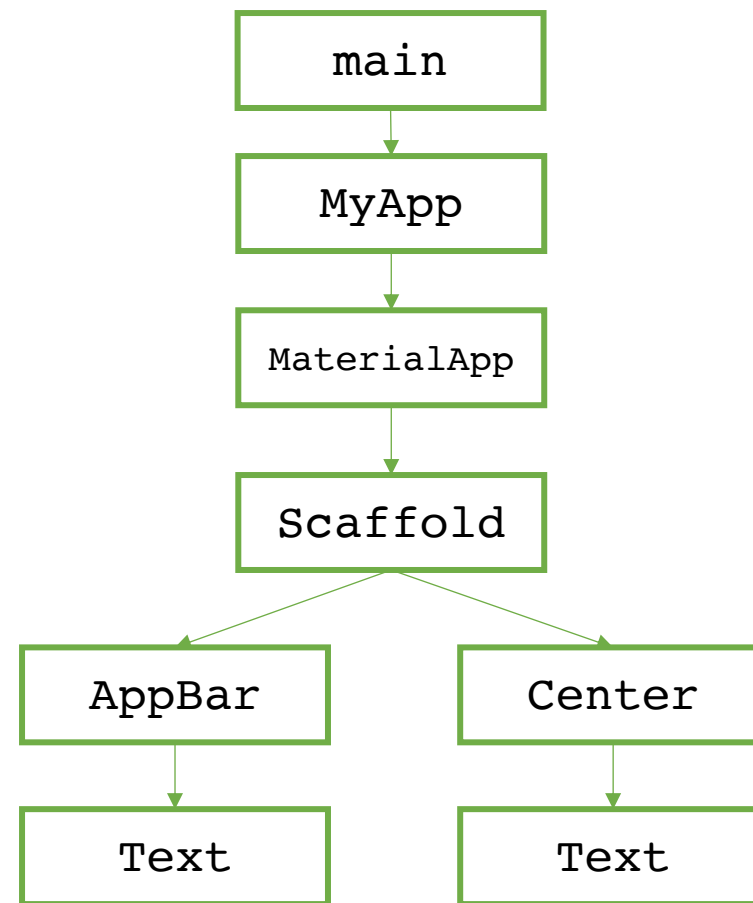
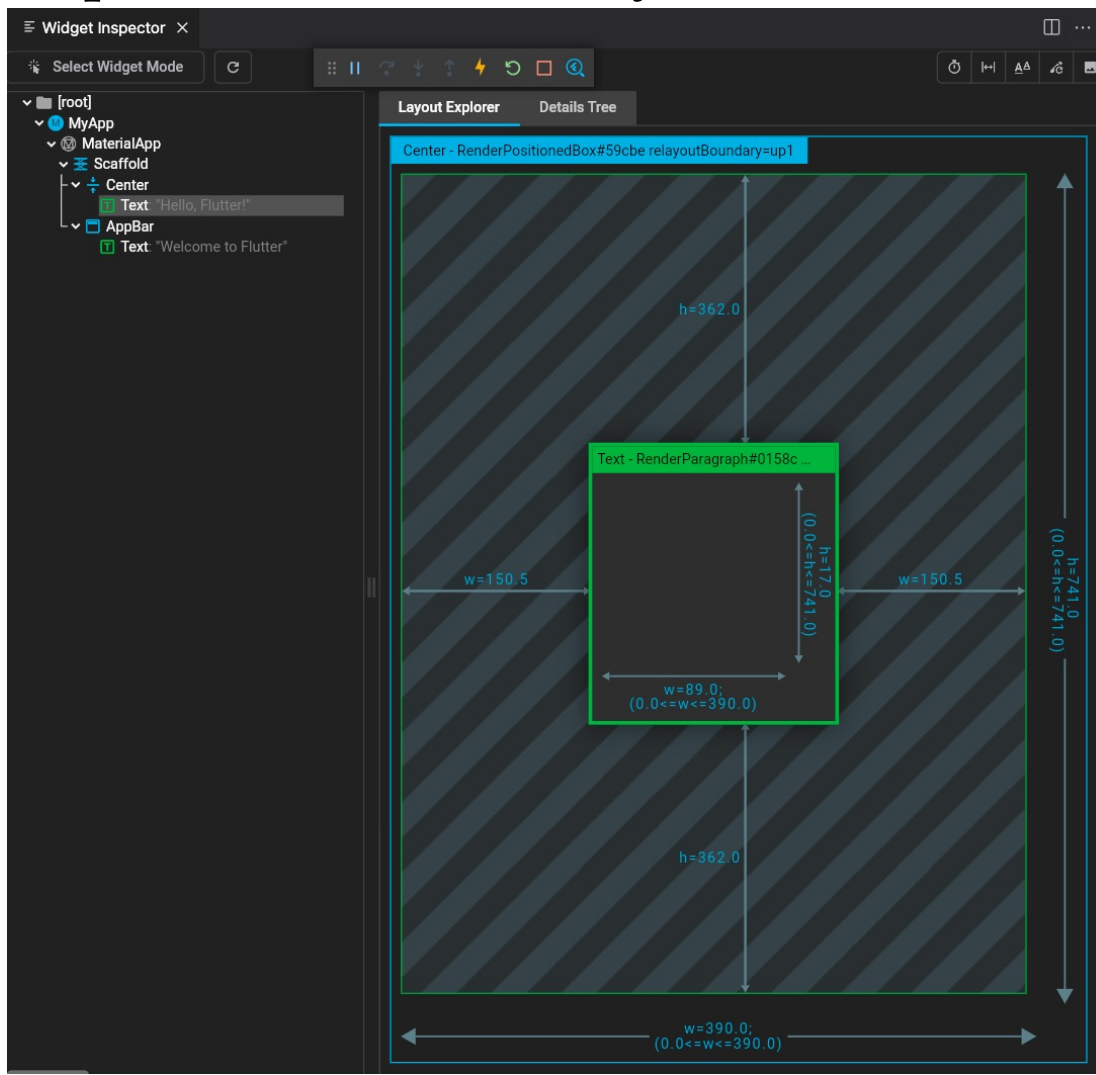
- DevTools is a suite of performance and debugging tools for Dart and Flutter.



DevTools

BONUS

- Simple example: with DevTools you can see the Widget Tree and it's layout!



Outline

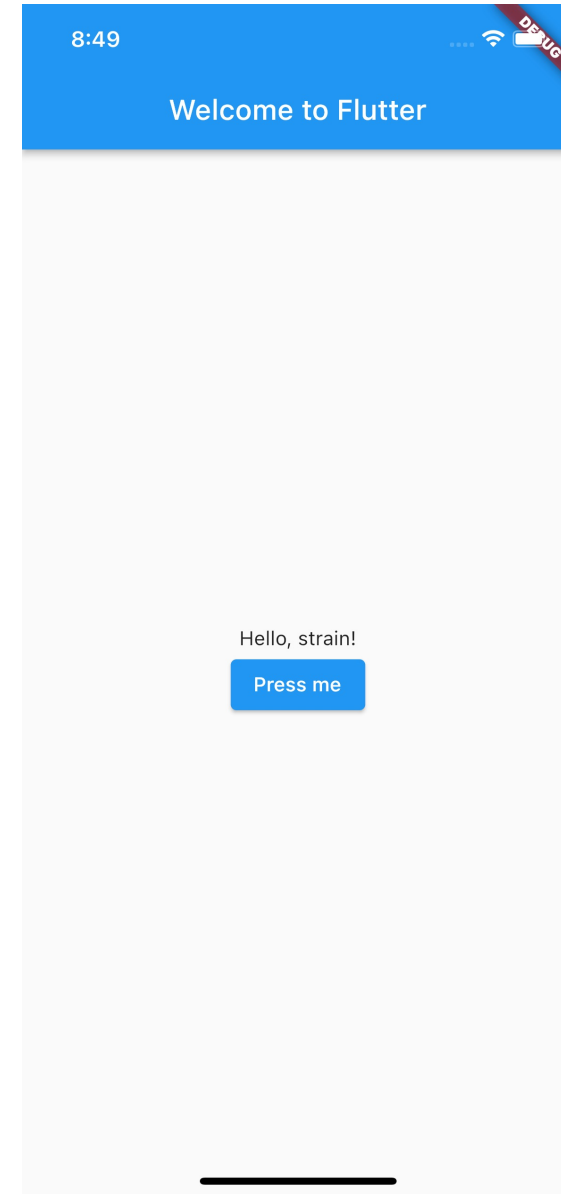
- Recap
- Creating a new project
- App dissection
- **Expanding our first app**
- Homework & Resources
- Final project overview

My first app with steroids

- Let's play with `my_first_app` and let's expand it
- Aim: change `my_first_app` to this
- We will learn how to:
 - Install an external package and add it as a dependency
 - Use the external package inside our app
 - StatefulWidget 101
 - How to modify the UI

My first app with steroids

- **Aim:** The result will be a very simple app that, each time a button is tapped, a new random "Hello" message is shown to the user.



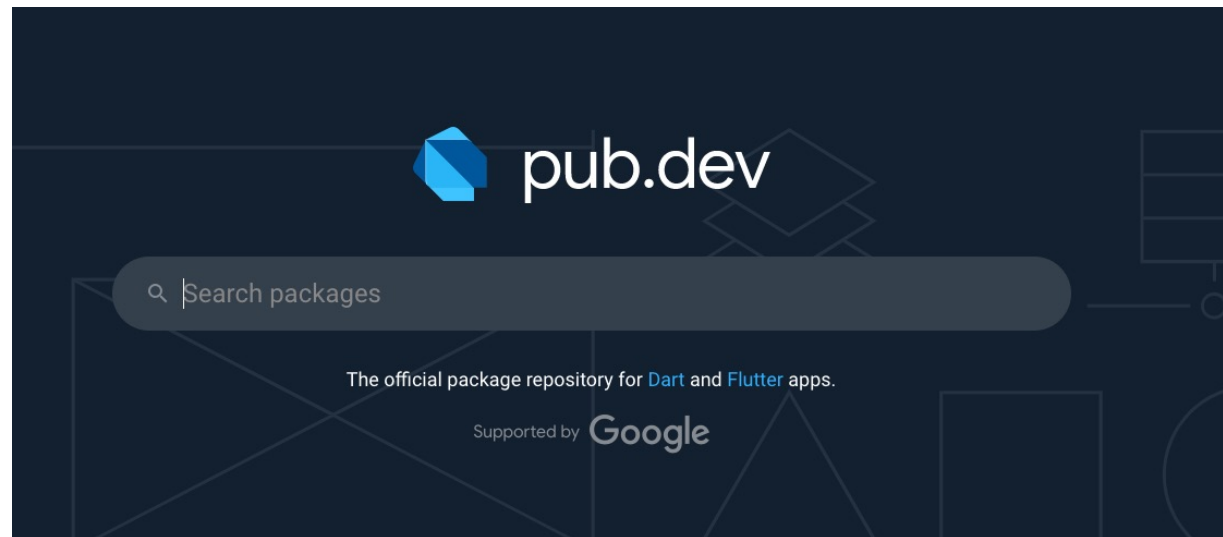
My first app with steroids

➤ Roadmap

1. Understand what to use to generate a random word
2. Generate a random word and check that everything is working
3. Display the word in the “Hello” message
4. Modify the UI to generate a new message each time a button is tapped

Solving point 1

- We do not want to code a random English word generator!
- On the Internet we can find a lot of already made code and ready-to-use packages that can fit your needs
- A place that we will visit often during this course is pub.dev:



This is the package I was looking for

- After some research, it seems like the **english_words** package can solve our needs
- It can generate words and words pairs!

How to use it? Docs!

Code is available too!

english_words 4.0.0

Published 9 months ago • [filiph.net](#) Null safety

[DART](#) [NATIVE](#) [JS](#) [FLUTTER](#) [ANDROID](#) [IOS](#) [LINUX](#) [MACOS](#) [WEB](#) [WINDOWS](#)

311

[Readme](#) [Changelog](#) [Example](#) [Installing](#) [Versions](#) [Scores](#)

english_words

build passing

A package containing the most ~5000 used English words and some utility functions.

Usage

Printing the top 50 most used nouns in the English language:

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

main() {
  nouns.take(50).forEach(print);
}
```

Computing number of syllables in a word:

```
syllables('beautiful'); // 3
syllables('abatement'); // 3
syllables('zoology'); // 4
```

Generating 5 interesting 2-syllable word combinations:

```
generateWordPairs().take(5).forEach(print);
```

311 130 98%
LIKES PUB POINTS POPULARITY

Publisher

[filiph.net](#)

Metadata

Utilities for working with English words. Counts syllables, generates well-sounding word combinations, and provides access to the top 5000 English words by usage.

[Repository \(GitHub\)](#)
[View/report issues](#)

Documentation
[API reference](#)

License
MIT ([LICENSE](#))

Dependencies
[string_scanner](#)

More

[Packages that depend on english_words](#)

Including english_words in the app

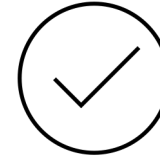
- Installing the english_words package in our app is very easy.
- By definition, it is a dependency right?
- So, let's add it under the dependency list of our app into pubspec.yaml
- After adding it, save pubspec.yaml and you will see VSCode running **flutter pub get** for you.
- Done!

```
...  
dependencies:  
  flutter:  
    sdk: flutter  
  
  cupertino_icons: ^1.0.2  
  
  english_words: ^4.0.0  
...
```

My first app with steroids

➤ Roadmap

1. Understand what to use to generate a random word
2. Generate a random word and check that everything is working
3. Display the word in the “Hello” message
4. Modify the UI to generate a new message each time a button is tapped



Generating a random word

- Let's add some line of code to main.dart to generate a word using the english_words package

- Modify the build method by adding

```
final word = WordPair.random().first;
```

before the return statement and run the app.

- Nothing it's happening. How to see if we are generating a random word?
- We can use the logger and the debug console!

Logging things

- Simply try to print the word value as a normal Dart program:

```
final word = WordPair.random().first;  
print(word);
```

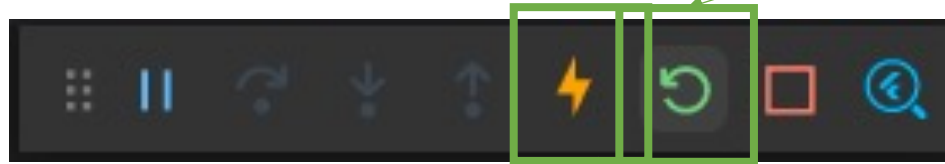
- If you run the application now you will see something like this in the **Debug Console** of VS Code:



```
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL  
Launching lib/main.dart on iPhone 13 in debug mode...  
Xcode build done. 24.9s  
Connecting to VM Service at ws://127.0.0.1:49666/brC-rDHNu3s=/ws  
flutter: duck
```


Logging things

- Every time you reload/restart the app



Restart button



Reload button

- ...you will see a different word

```
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL
Restarted application in 416ms.
flutter: left
```

My first app with steroids

➤ Roadmap

1. Understand what to use to generate a random word 
2. Generate a random word and check that everything is working 
3. Display the word in the “Hello” message
4. Modify the UI to generate a new message each time a button is tapped

Change the Hello message

- You should be able to solve this point by yourself now
- Simply, using string interpolation, change




`'Hello, Flutter!'` to `'Hello, $word!'`

- and save to reload the app and see the changes.

Hello, soft!

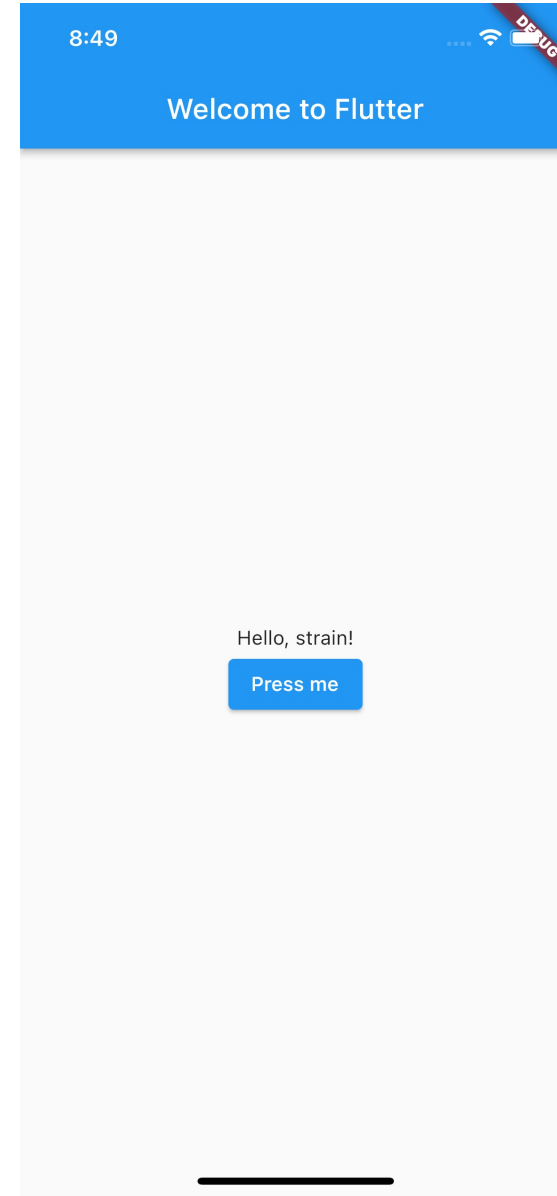
My first app with steroids

➤ Roadmap

1. Understand what to use to generate a random word 
2. Generate a random word and check that everything is working 
3. Display the word in the “Hello” message 
4. Modify the UI to generate a new message each time a button is tapped

Changing the UI

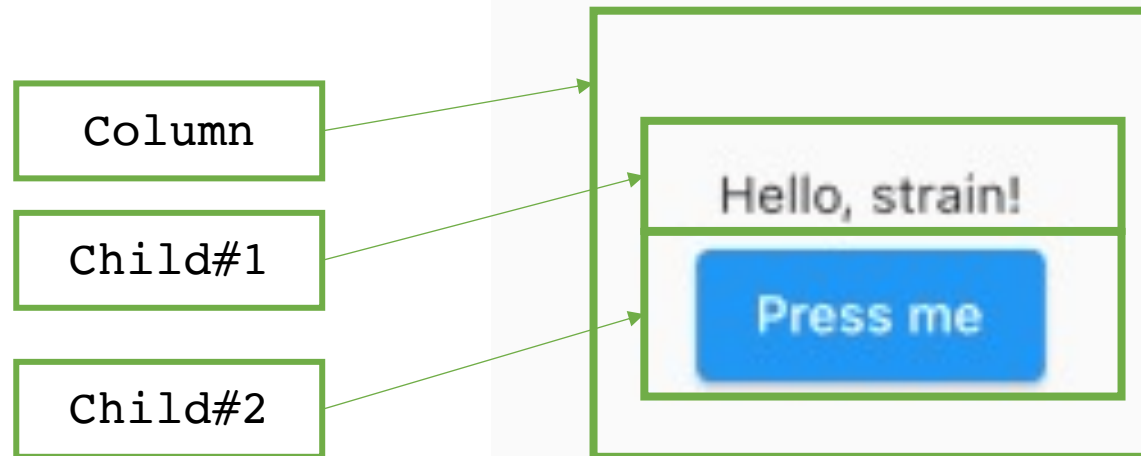
- Let's start by simply changing the UI
- We need to obtain something like
- Problems:
 1. How to add a button
 2. How to put it there



The Column Widget

- We can use the Column widget.
- It has a list of children (not like Text or Center or Scaffold)
- Children are lined up to a column from top to the bottom

```
Column(  
  children: [  
    Child#1,  
    Child#2,  
  ],  
);
```



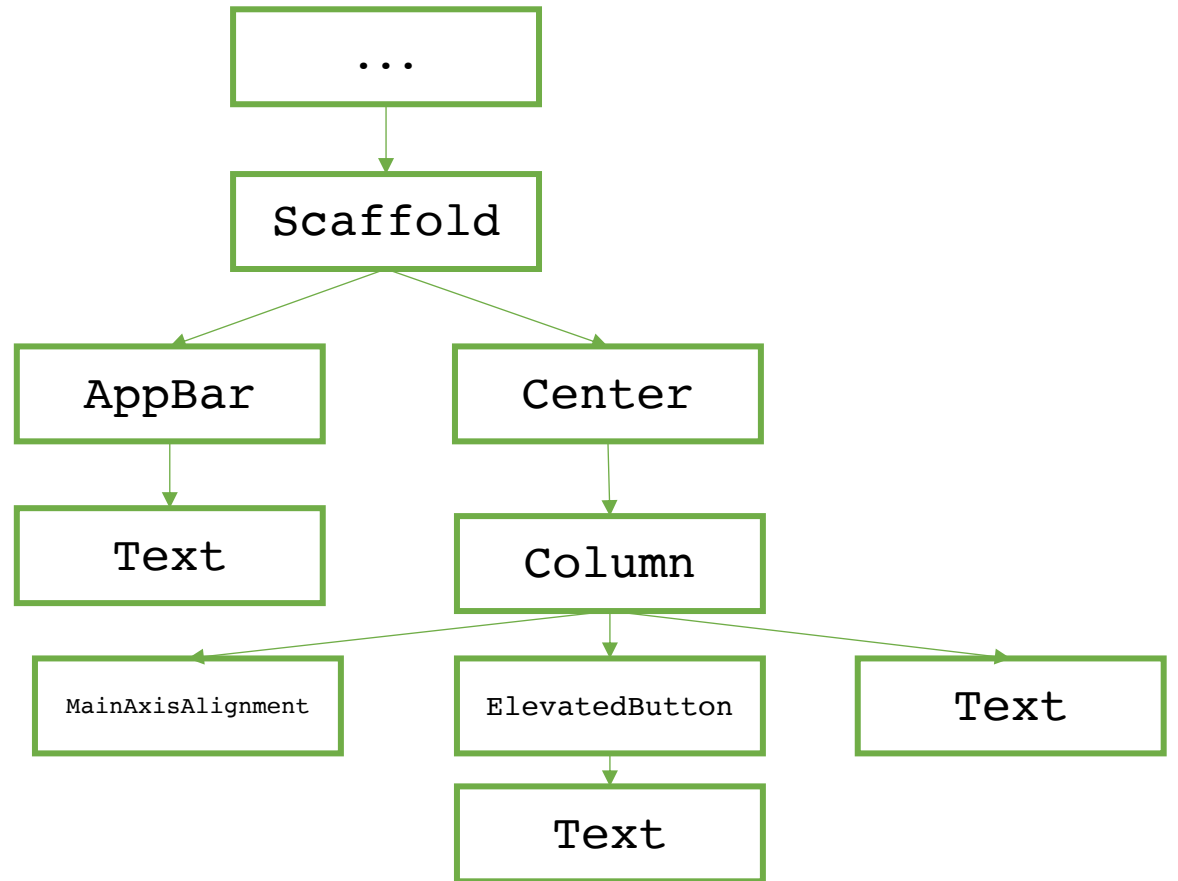
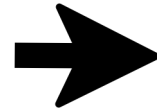
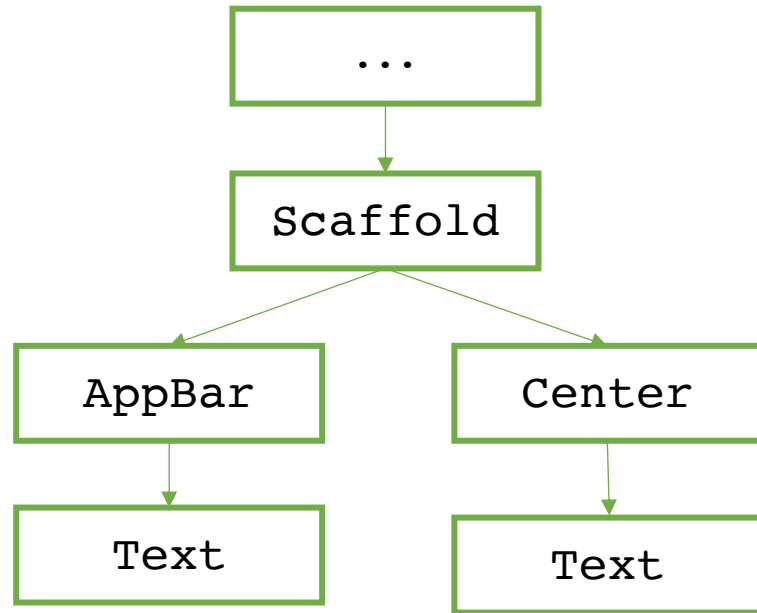
Implement the new UI

- Change the build method of MyApp to

```
Widget build(BuildContext context) {  
  final word = WordPair.random().first;  
  print(word);  
  return MaterialApp(  
    title: 'Welcome to Flutter',  
    home: Scaffold(  
      appBar: AppBar(title: const Text('Welcome to Flutter')),  
      body: Center(  
        child: Column(  
          mainAxisAlignment: MainAxisAlignment.center,  
          children: [  
            Text('Hello, $word!'),  
            ElevatedButton(onPressed: (){}, child: const Text('Press me')),  
          ],  
        ),  
      ),  
    ),  
  );  
} //build
```

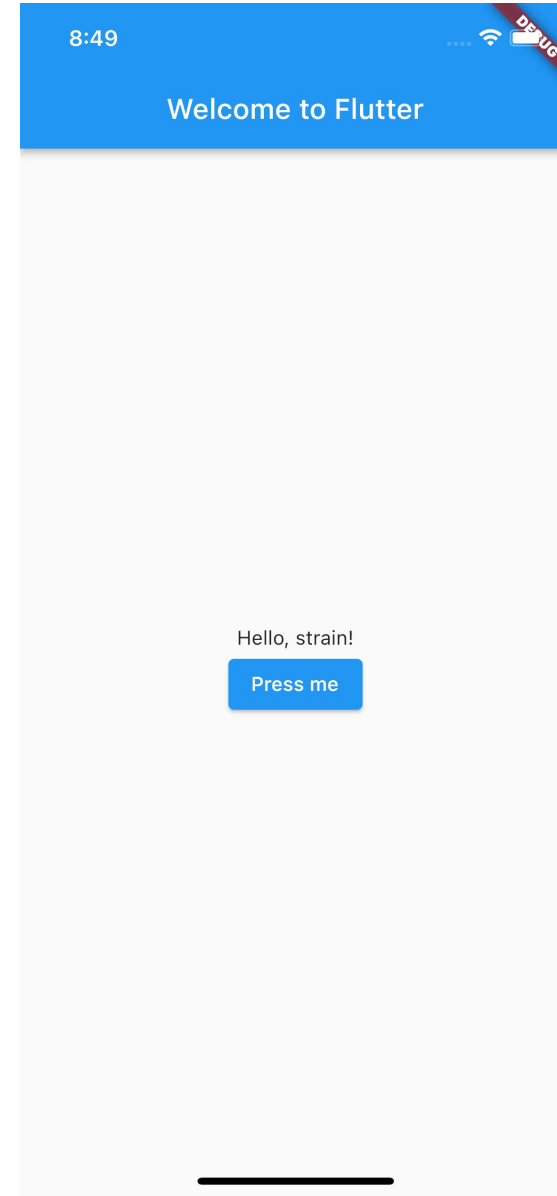
Different UI, different tree

- How the widget tree changed?



Changing the UI

- (New) Problem: How to change the message when we press the button?
- In other words: how to change the **app state** without reloading or restarting everything
- We need a **StatefulWidget**



StatefulWidget

- As we mentioned before, stateful widgets maintain state that might change during the lifetime of the widget.
- Implementing a stateful widget requires at least two classes:
 1. A **StatefulWidget** class that creates an instance of...
 2. ...a **State** class.
- Note: The **StatefulWidget** class is, itself, immutable and can be thrown away and regenerated, but the **State** class persists over the lifetime of the widget.

The boilerplate code of a StatefulWidget

```
class RandomHello extends StatefulWidget{
  const RandomHello({Key? key}) : super(key: key);

  @override
  _RandomHelloState createState() => _RandomHelloState();
} //RandomHello

class _RandomHelloState extends State<RandomHello>{

  @override
  Widget build(BuildContext buildContext){
    //return some widget
  } //build

} //_RandomHelloState
```

Note: the state is private to the Widget. Not necessary, but it is a good practice to understand what is private and what is not.

Refactoring the UI - RandomHello

- Let's copy some code into the build method new Widget

```
...
class _RandomHelloState extends State<RandomHello>{

  @override
  Widget build(BuildContext buildContext){
    final word = WordPair.random().first;
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        Text('Hello, $word!'),
        ElevatedButton(onPressed: (){}, child: const
Text('Press me')),
      ],
    );
  } //build

} // _RandomHelloState
```

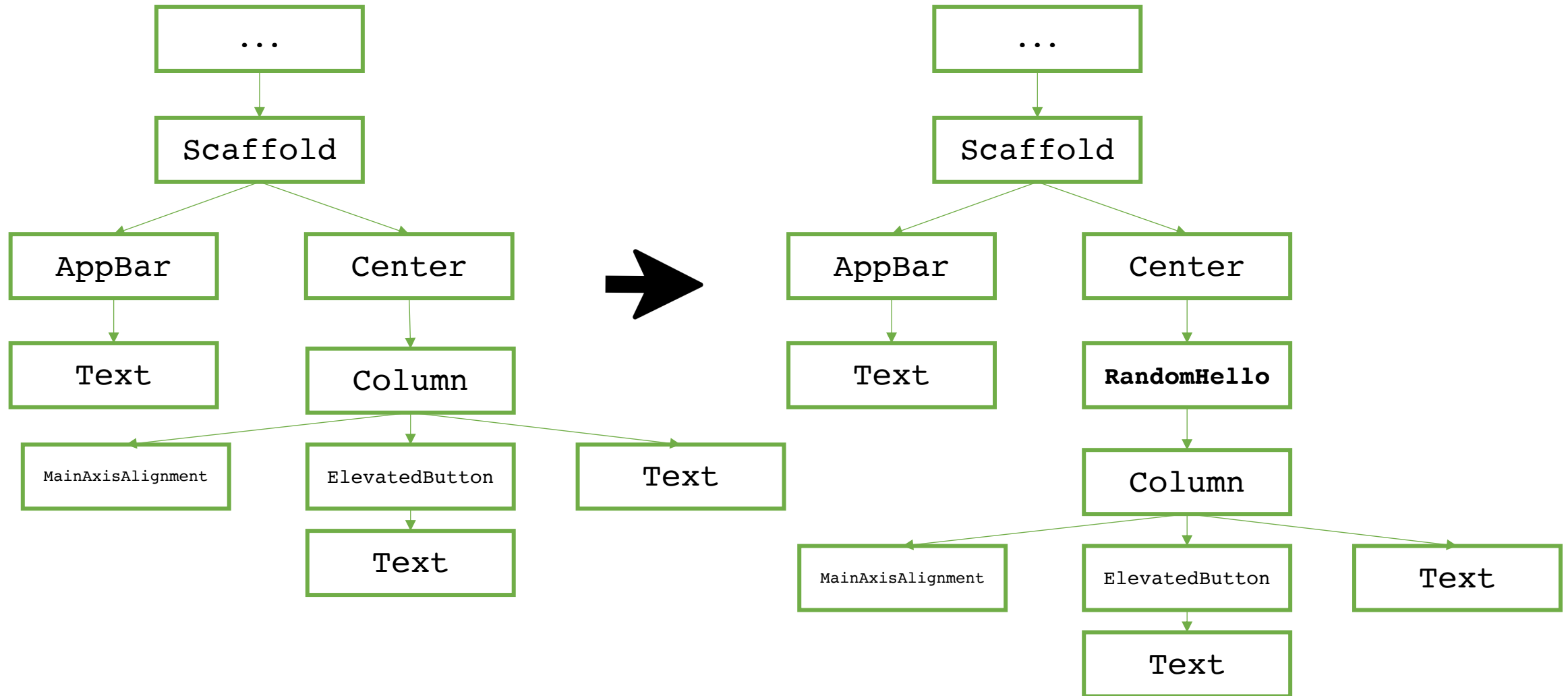
Refactoring the UI - MyApp

➤ Now let's refactor the MyApp code

```
class MyApp extends StatelessWidget {  
  const MyApp({Key? key}) : super(key: key);  
  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Welcome to Flutter',  
      home: Scaffold(  
        appBar: AppBar(title: const Text('Welcome to Flutter')),  
        body: const Center(child: RandomHello()),  
      ),  
    );  
  } //build  
  
} //MyApp
```

Same UI, different tree

- The UI should look like the same as before, but we have a new widget tree



void initState(){} ---

- Let's do some changes to RandomHello to make it more **stateful**

```
class _RandomHelloState extends State<RandomHello>{
```

```
    String? _word; ◀
```

_word will represent the state of the Widget

```
    @override
```

```
    void initState() { ◀
```

initState is a special method that is called the first time the Widget is created. It is used (as its name suggest) to initialize the state of the Widget itself.

```
        _word = WordPair.random().first;
```

```
        super.initState();
```

```
    } //initState
```

```
    ...
```

setState((){})

- We are ready to implement the function to provide to onPressed

```
...
@override
Widget build(BuildContext buildContext){
  return Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: [
      Text('Hello, $_word!'),
      ElevatedButton(onPressed: _changeWord, child: const
Text('Press me')),
    ],);
} //build

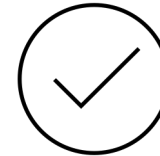
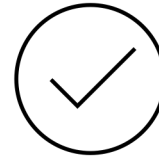
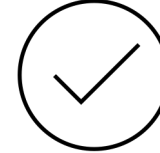
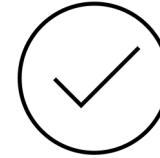
void _changeRandomWord(){
  setState(() {
    _word = WordPair.random().first;
  });
} // _changeRandomWord
...
```

setState is a special method that requires a callback function as input. setState notifies the Flutter framework that the state might be changed causing to delete and rebuild the widget itself.

My first app with steroids

➤ Roadmap

1. Understand what to use to generate a random word
2. Generate a random word and check that everything is working
3. Display the word in the “Hello” message
4. Modify the UI to generate a new message each time a button is tapped



Outline

- Recap
- Creating a new project
- App dissection
- Expanding our first app
- **Homework & Resources**
- Final project overview

Homework

- Get familiar with the structure of a Flutter project and how to install new packages using pubspec.yaml
- Get familiar with the concept of Widget
- To know what to do to create a StatelessWidget and a StatefulWidget
- Understanding the Flutter flow

Resources

➤ Introduction to Widgets

- <https://docs.flutter.dev/development/ui/widgets-intro>

➤ Write your first Flutter app, part 1 codelab

- <https://docs.flutter.dev/get-started/codelab>

➤ DevTools

- <https://docs.flutter.dev/development/tools/devtools/overview> ⓓ

Outline

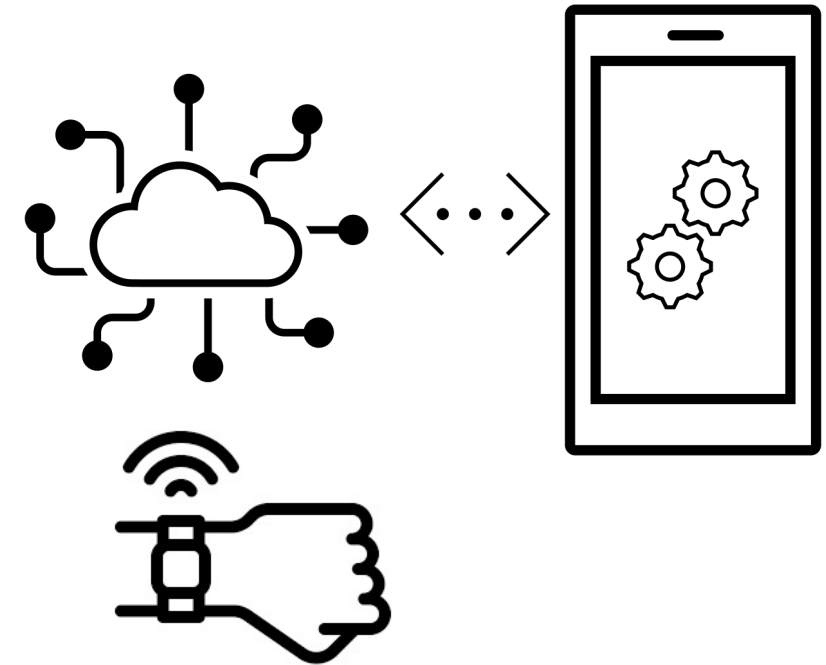
- Recap
- Creating a new project
- App dissection
- Expanding our first app
- Homework & Resources
- **Final project overview**

What's the spirit here

- The goal is learning “**How do SOMETHING using wearable devices**”
- What **I DO** expect from you
 - To use your imagination to create that SOMETHING
 - To learn to build something from scratch
 - To use code built by others
 - To work together
 - To use Google and StackOverflow
- What **I DO NOT** expect from you
 - To create the new Google
 - To reinvent the wheel (copying others code is fine)
 - To create extra complicated UI with animations and stuff
 - To implement sophisticated design pattern (even if it would be a nice thing to do)

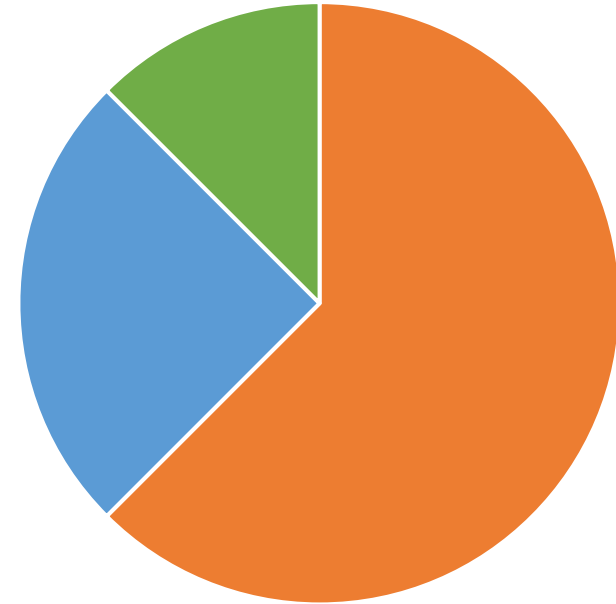
Project structure

- The project consists of building an app for iOS or Android that collects user data from a wearable device through Web APIs, stores them, visualizes them, and does some tricks with them.
- Core functionalities:
 - User authentication and management
 - Data collection
 - Data persistence
 - Data visualization and presentation
- Additional functionalities → It's up to you!
Some examples:
 - Run some analysis on data and provide suggestions to the user
 - Implement some literature algorithm
 - ...



Grading

- What are the grading criteria
 - Originality of the additional feature
 - Quality of teamwork
 - Compliance to core functionalities
- The vote is 32/30 (31/32 = 30L)
- How the grade is split?
 - 20 points: core functionalities
 - 8 points: additional app functionalities
 - 4 points: GIT working tree



- Core functionalities
- Additional functionalities
- GIT working tree