Link Download Doc Codelab

 enter image description here

Flutter

Flutter è un framework open-source di Google per creare User Interface native su un'unica unica code base, unico codice e possiamo compilare applicazioni su più piattaforme(Android, iOS, Windows, macOS, Linux, Web).

Vantaggi:

Prestazioni e compatibilità

- Le applicazioni sviluppate con Flutter girano nativamente sui dispositivi, questo vuol dire che le prestazioni sono elevate(frame rate elevato), il rendering è a basso livello e utilizzando la libreria grafica di Google, Skia.
- Flutter si interfaccia con gli SDK del sistema operativo su cui gira(Android o iOS) quindi è anche possibile creare funzionalità specifiche rispetto al sistema su cui l'applicazione gira.

Debug

- In Flutter, il Debug è molto semplice, durante lo sviluppo è possibile visualizzare ed eseguire l'applicazione in modalità **debug mode**
- **Flutter inspector**, è uno strumento potente per visualizzare il layout della vostra applicazione in sviluppo, oltre che per visualizzare il layout è molto utile per diagnosticare e intercettare problemi di layout dell'app.
- Una funzionalità apprezzata dagli sviluppatori è "l'**hot-reload**" che consente di iniettare e visualizzare istantaneamente le modifiche fatte dall'editor sull'esecuzione dell'app, senza dover ricompilare l'intero codice. Questo ne ottimizza il workflow dello sviluppatore e le varie fasi di debug.

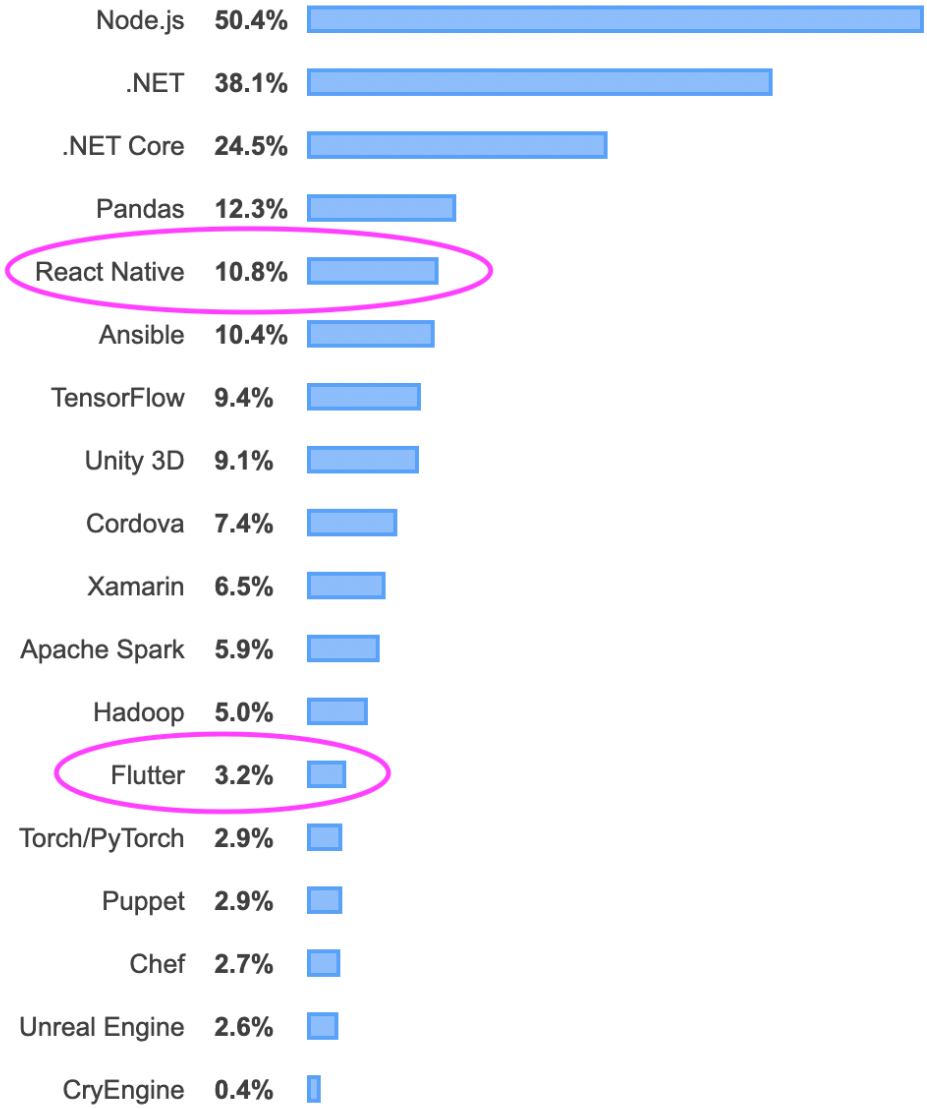
Curva di apprendimento

- Flutter è molto più **facile da imparare**(Dart) rispetto ad altri frameworks in cui si usa javascript come linguaggio per React Native
- Il linguaggio Dart è un linguaggio molto più **intuitivo** e anche **più vicino ai paradigmi di programmazione** ed è molto simile ai linguaggi di programmazione usati nello sviluppo di app mobili native(Java o Kotlin)

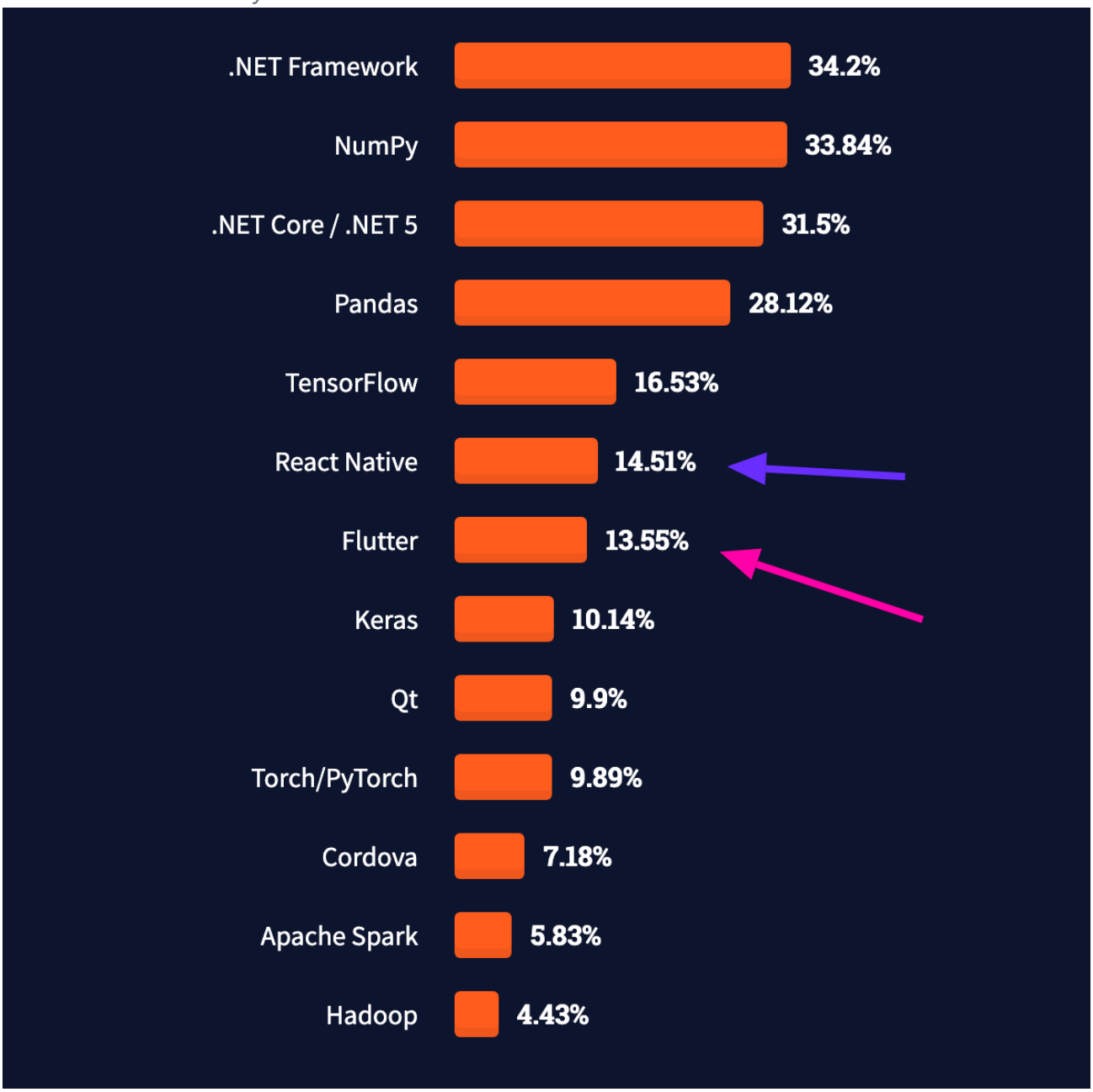
Popolarità e community

- La popolarità di Flutter è in continua **crescita** da quando è disponibile per tutti gli sviluppatori infatti è possibile constatare da sondaggi fatti su Stack Overflow che l'interesse da parte degli sviluppatori è in continua crescita:

> Source: Stack Overflow Survey 2019

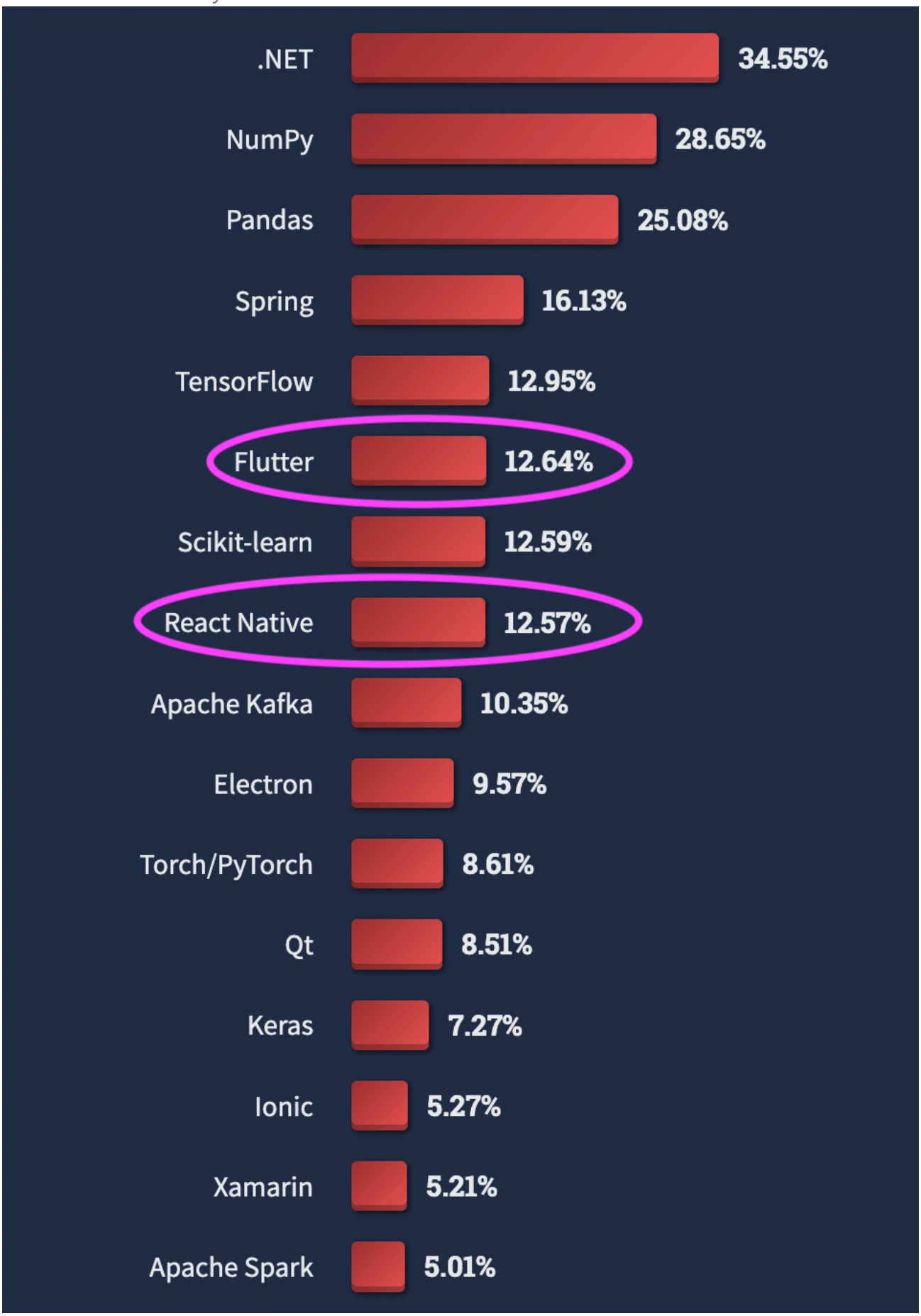


> Stack Overflow Survey 2021



Nell'ultimo sondaggio su Stack Overflow notiamo che Flutter e React Native sono in stretta competizione

Stack Overflow Survey 2023



- Essendo anche la Community di Flutter in costante crescita dalla sua uscita, è possibile ritrovare sul web una vasta community di sviluppatori e librerie da integrare per i progetti Flutter

Installazione e configurazione

Probabilmente questa sarà la parte più difficile di questo codelab, ovvero installare tutto e assicurarsi che funzioni. È l'unica cosa meno divertente ma è importante per iniziare a sviluppare applicazioni in Flutter.

2) Installazione di Flutter

Per prima cosa scarichiamo l'sdk di flutter da questo indirizzo. Una volta scaricato, estrarre il contenuto dell'archivo in:

> C:\SDKs

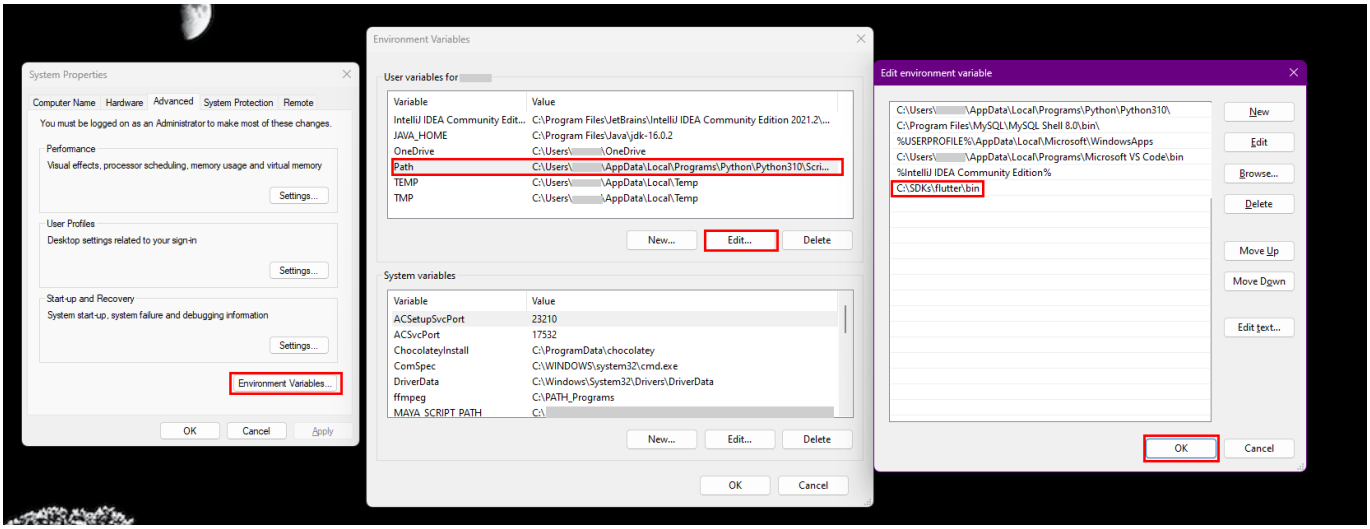
Creare una cartella SDK per posizionare l'SDK di Flutter. La posizione finale sarà:

> C:\SDKs\flutter

3) Configurare Path di sistema

È estremamente importante aggiornare il PATH nelle variabili di sistema. Per eseguire i comandi Flutter attraverso la console di Windows sarà necessario aggiungere al PATH la variabile di ambiente. Nella barra di ricerca di Windows inserisci "**env**" e seleziona "**Modifica variabili di ambiente**" e aggiungi questo percorso:

> C:\SDKs\flutter\bin



Per verificare che tutto funzioni, chiudere tutte i terminali Windows eventualmente aperti, aprire un Terminale ed eseguire il comando:

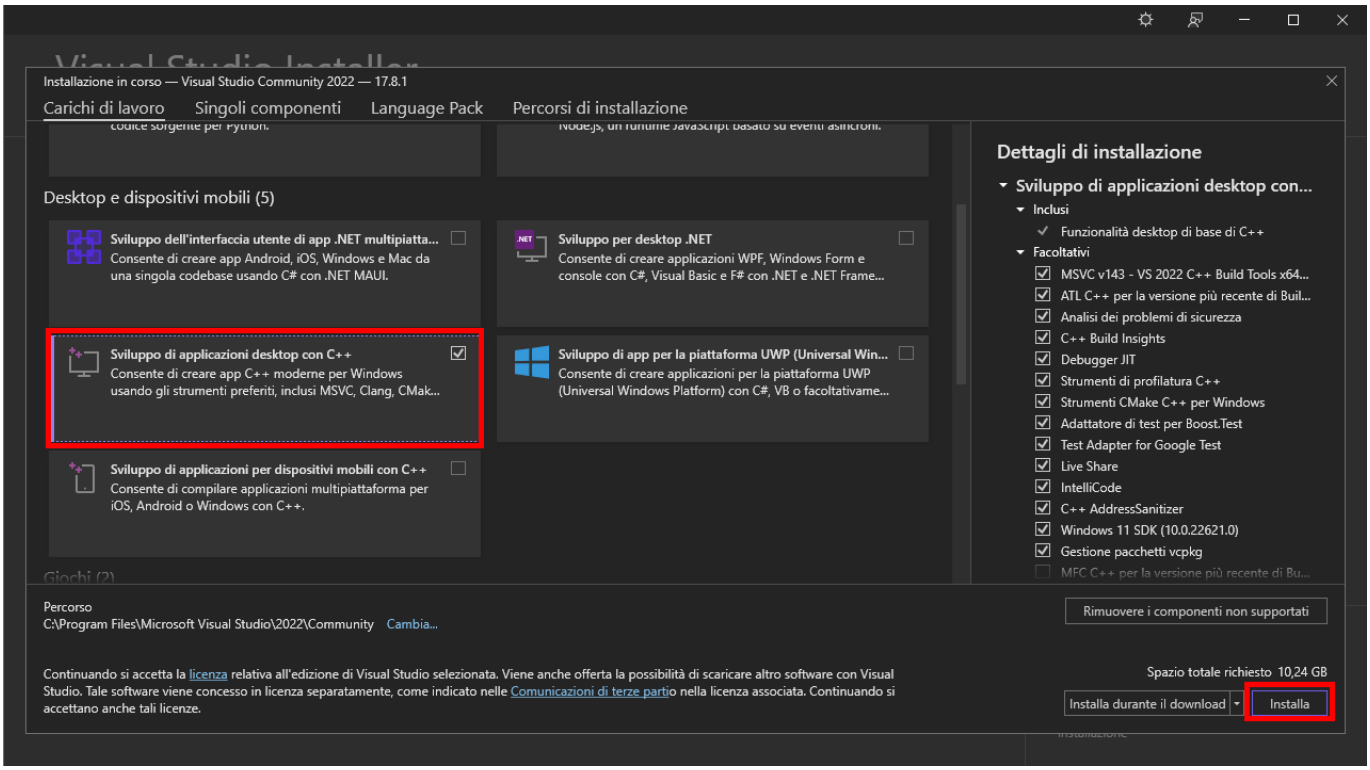
> flutter doctor

Se si dovesse verificare un errore con codice **"Error: Unable to find git in your PATH."** segui la guida chiamata "gitError/GitError.pdf" all'interno della cartella "CodelabFlutterUni-main".

4) Installazione Tool Sviluppo Windows

In questo Codelab scriveremo una piccola app per Windows e Flutter per compilare la base di codice avrà bisogno del kit di sviluppo Windows. Scaricare il setup del kit di sviluppo windows Visual Studio 2022, Scaricabile da qui. **Attenzione non stiamo parlando dell'editor Visual Studio Code!**. Una volta scaricato il **setup di Visual Studio 2022** bisogna verificare nelle impostazioni di installazione che “Desktop development with C++” sia selezionato, incluse tutte le componenti

di default.

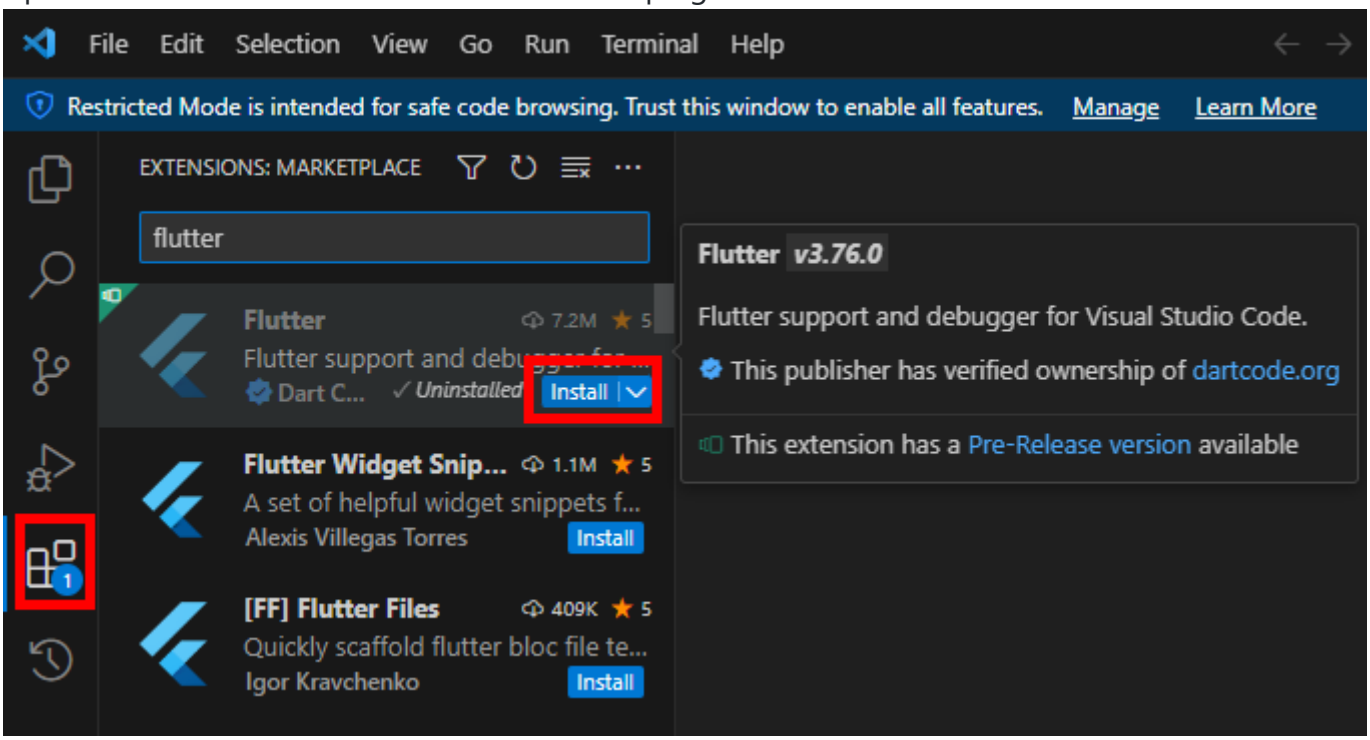


5) Installazione di Visual Studio Code

Useremo Visual Studio Code perchè è semplice, affidabile e lo usano molte persone. Chiaramente è gratis. Link installazione

6) Installazione Plugin Flutter

Apri l'editor di Visual Studio Code e Installa il plugin flutter



Codelab di oggi

Questo corso accelerato è una rielaborazione di un Codelab(Tutorial) di Google (Fonte codelab) di cui oggi vedremo:

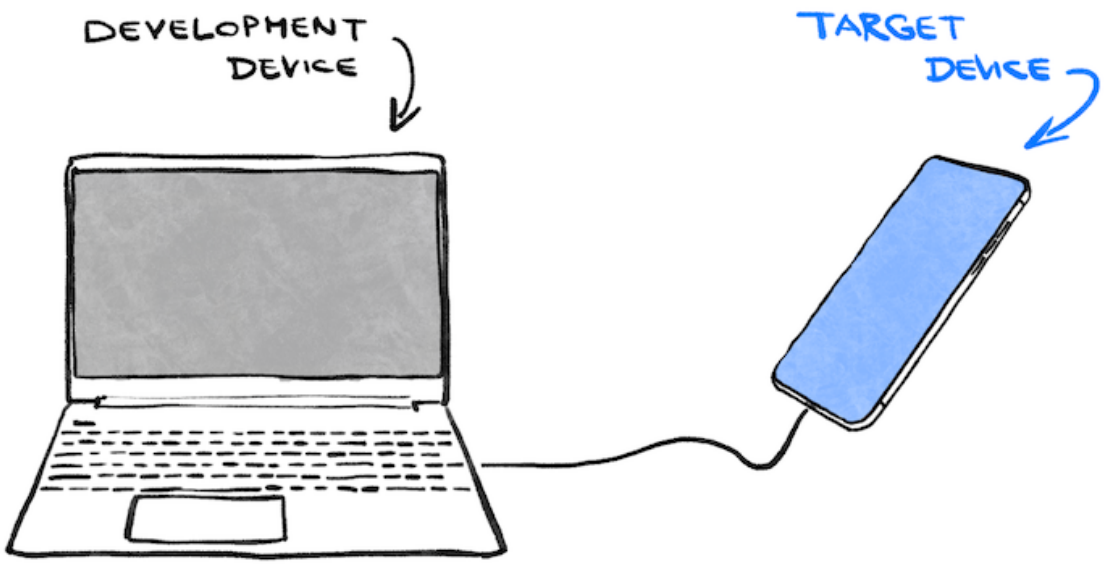
- Creare la prima app
- Capire le basi del funzionamento di Flutter
- Creare un layout per la nostra prima app
- Collegare le interazioni dell'utente al comportamento dell'app(usando ad esempio un pulsante e visualizzando il risultato a schermo)
- Organizzare il codice Flutter
- Gestire gli stati dell'applicazione(dati)

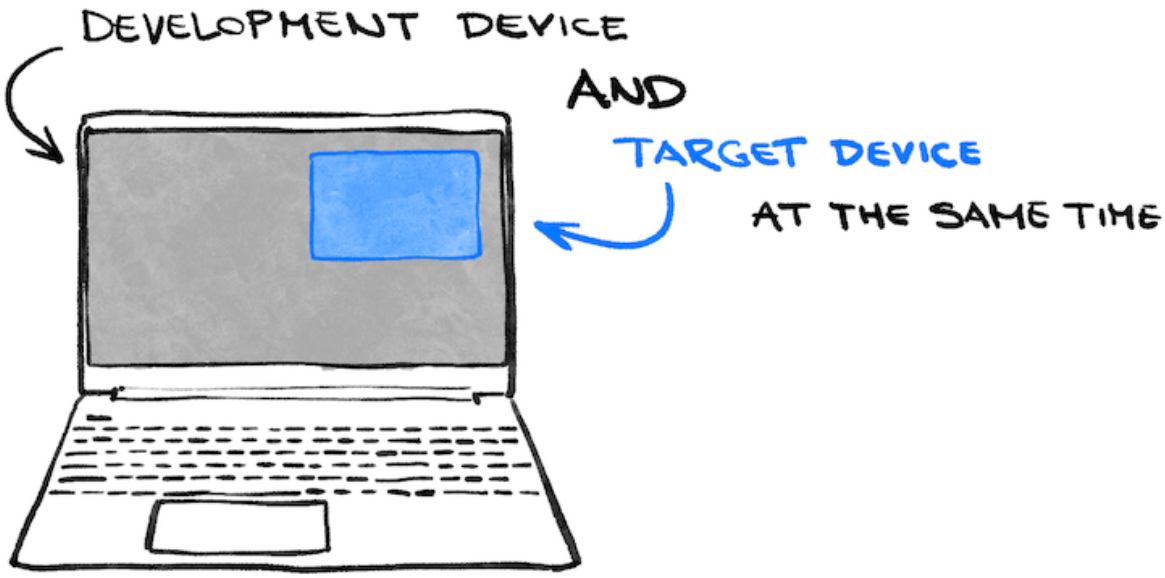
Considerazioni sviluppo con Flutter

Quando svilupperete un'app spesso non la svilupperete per tutte le piattaforme, ma è importante scegliere una piattaforma target iniziale su cui sviluppare e magari in seguito adattare l'app per tutte le altre piattaforme.

In questo caso sceglieremo Windows come **Target Device** e **Development Device**.

Un altro suggerimento è che se state sviluppando un'app per computer Windows allora usate un sistema operativo Windows per farlo. Questo rende alcune cose più facili e veloci.

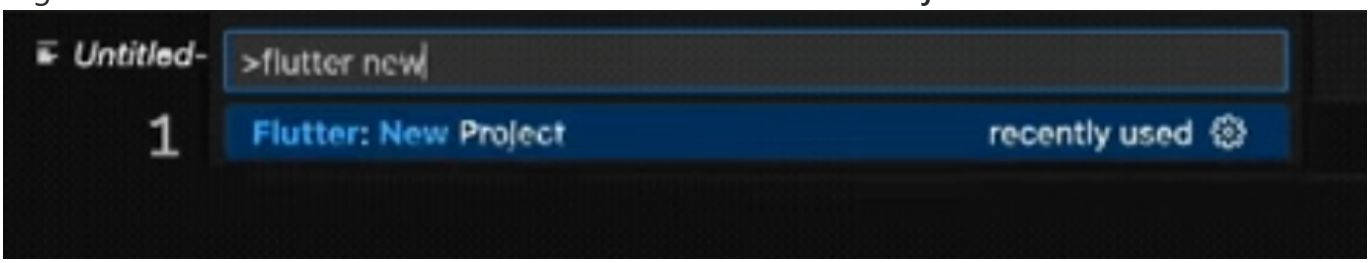
> Esempio **Development Device** e **Target Device**
>
> 
>

> Esempio **Development Device** usato sia per lo sviluppo che come **Target Device**
>
> 
>

Nuovo progetto

Creiamo un nuovo progetto

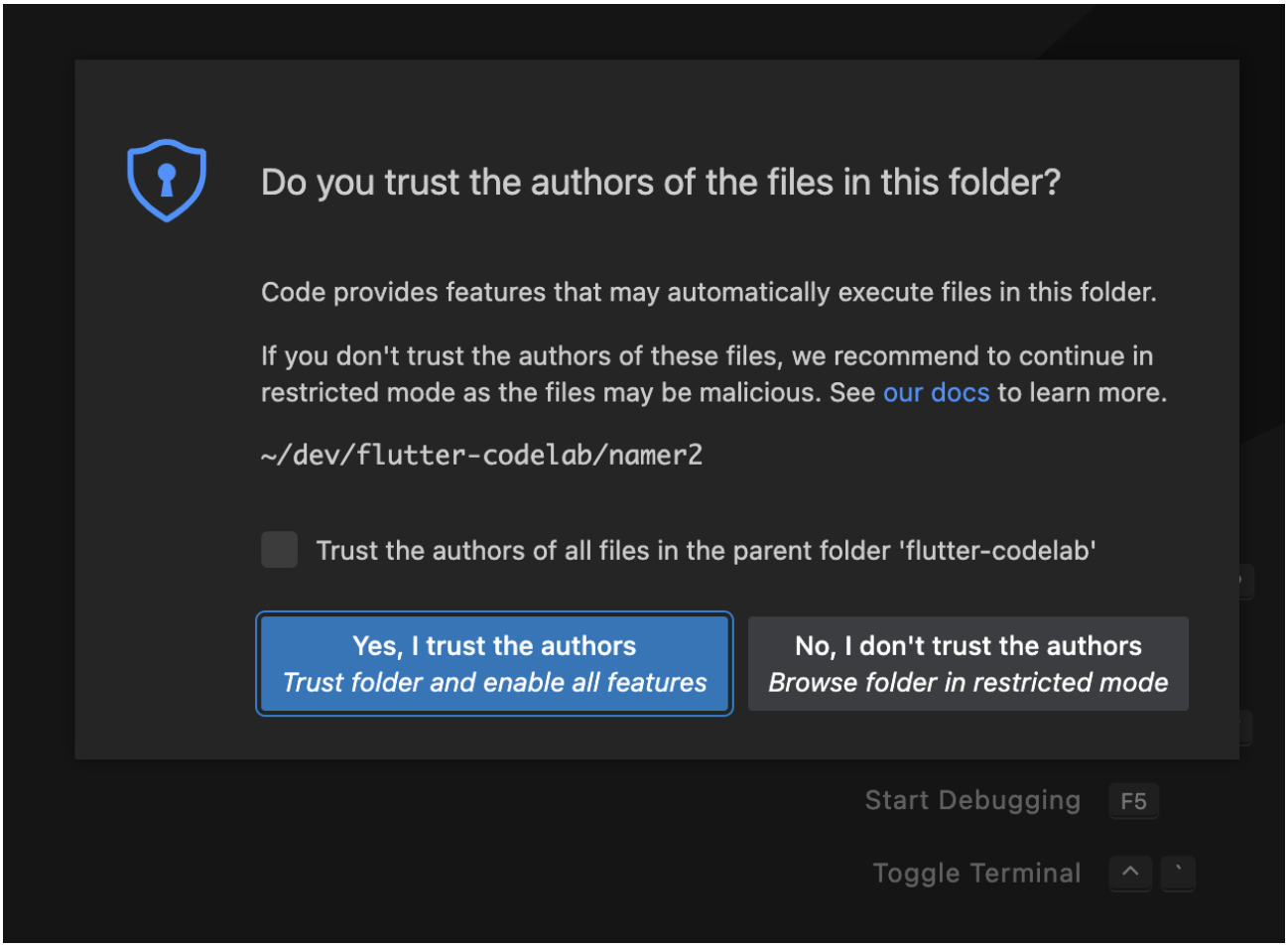
Avvia Visual Studio Code, aprire la command palette (con F1 o Ctrl+Shift+P o Shift+Cmd+P). Digitare "**flutter new**" e selezionare il comando "**Flutter: New Project**"



Dopo selezionare "**Application**", seleziona ora una cartella in cui creare un nuovo progetto. Per comodità nella cartella Documenti Creiamo una cartella chiamata Flutter e la selezioniamo. Infine dai un nome alla tua app come `flutter_app` o `flutter_application_1`.

A questo punto flutter procederà con la creazione del progetto. Ci potrebbero volere alcuni minuti. Appena il progetto sarà creato VS lo aprirà.

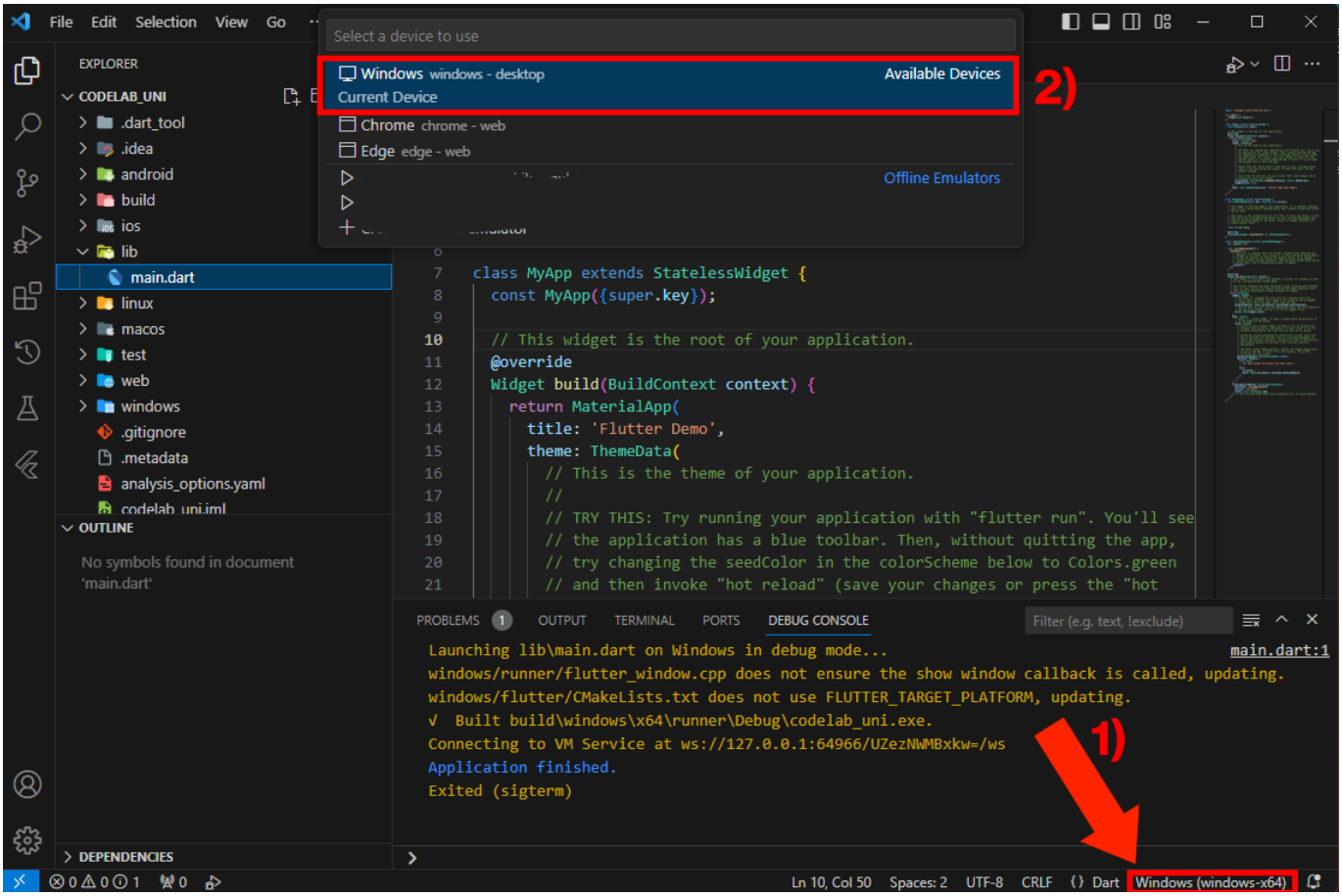
> **Nota:** VS Code mostrerà una finestra chiedendo se può considerare attendibile il contenuto della cartella. Selezionare Si. Se premi no verrà disabilitata la possibilità di usare Flutter nella cartella.



> Se non appare la voce **Flutter: New Project** consultare https://code.visualstudio.com/docs/editor/workspace-trust

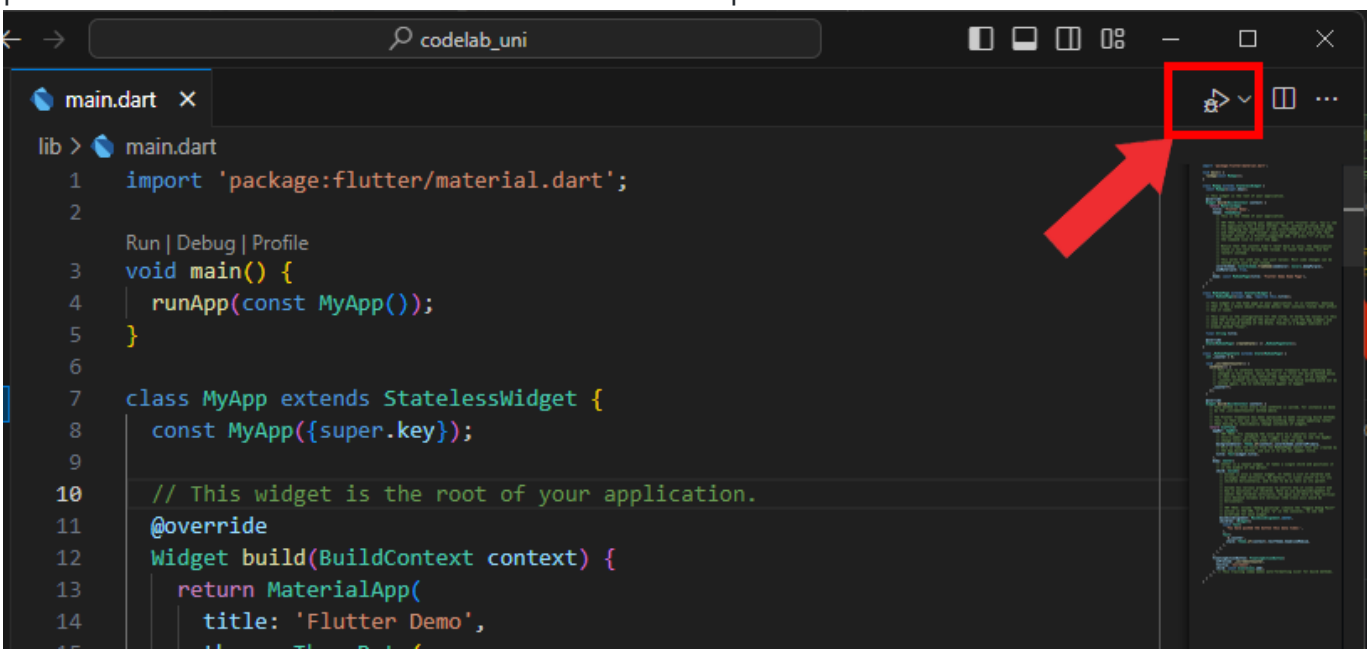
Selezioniamo il Target Device

Per selezionare il **target device**, (1) premere nell'angolo in basso a destra di VS Code. (2) Selezionare quindi **Windows - desktop** che dovrebbe comparire tra i device disponibili.



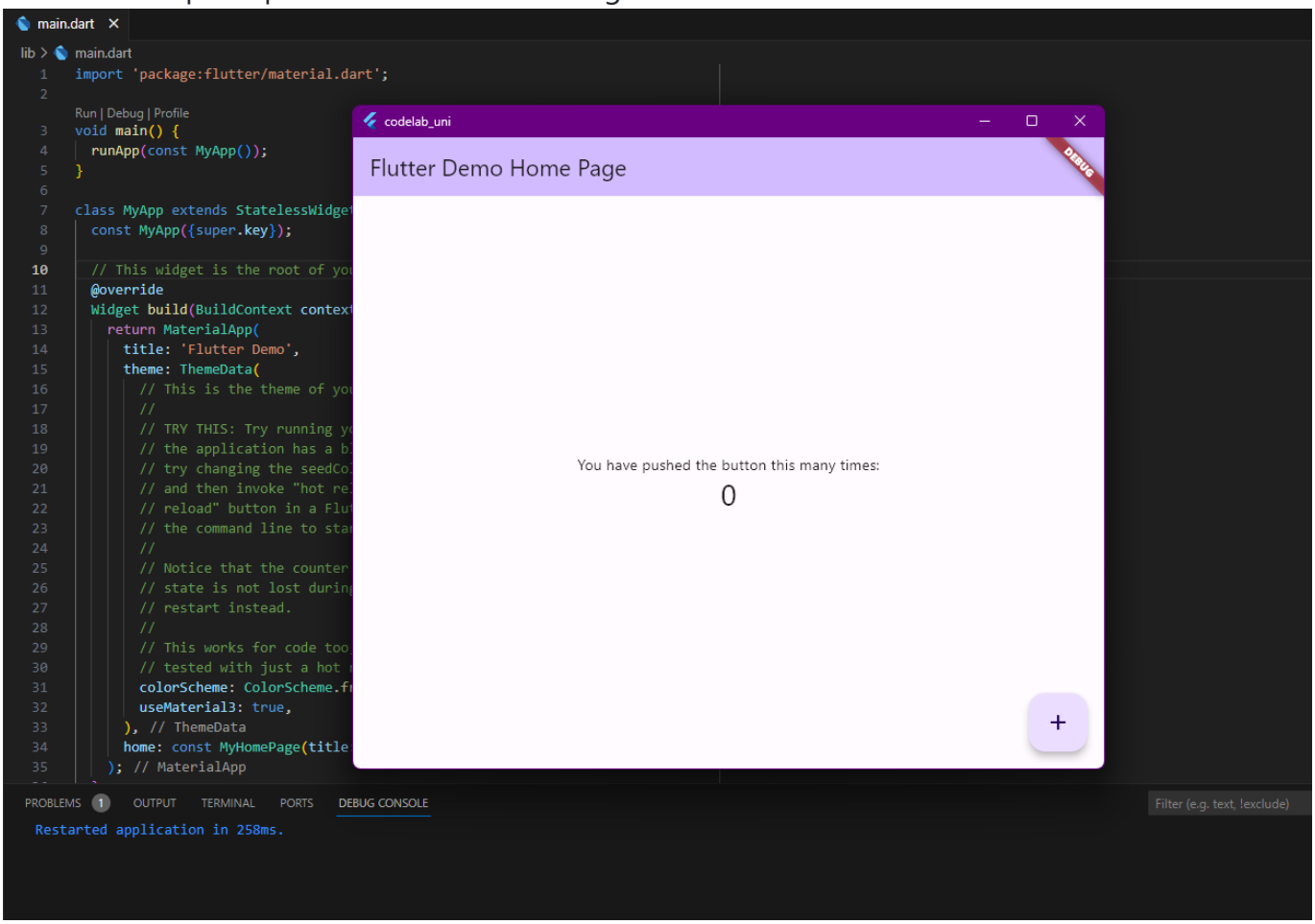
Avviamo l'app(Debug)

Per avviare l'app in modalità Debug assicurarsi che sia aperto il file `lib/main.dart` e premere il pulsante in alto a destra della finestra di VS Code o premere F5



Dopo circa un minuto l'app verrà avviata l'app in modalità debug. Potremo visualizzare quindi la

demo Flutter preimpostata mentre viene eseguita nella finestra.



Idea dell'app e codice partenza

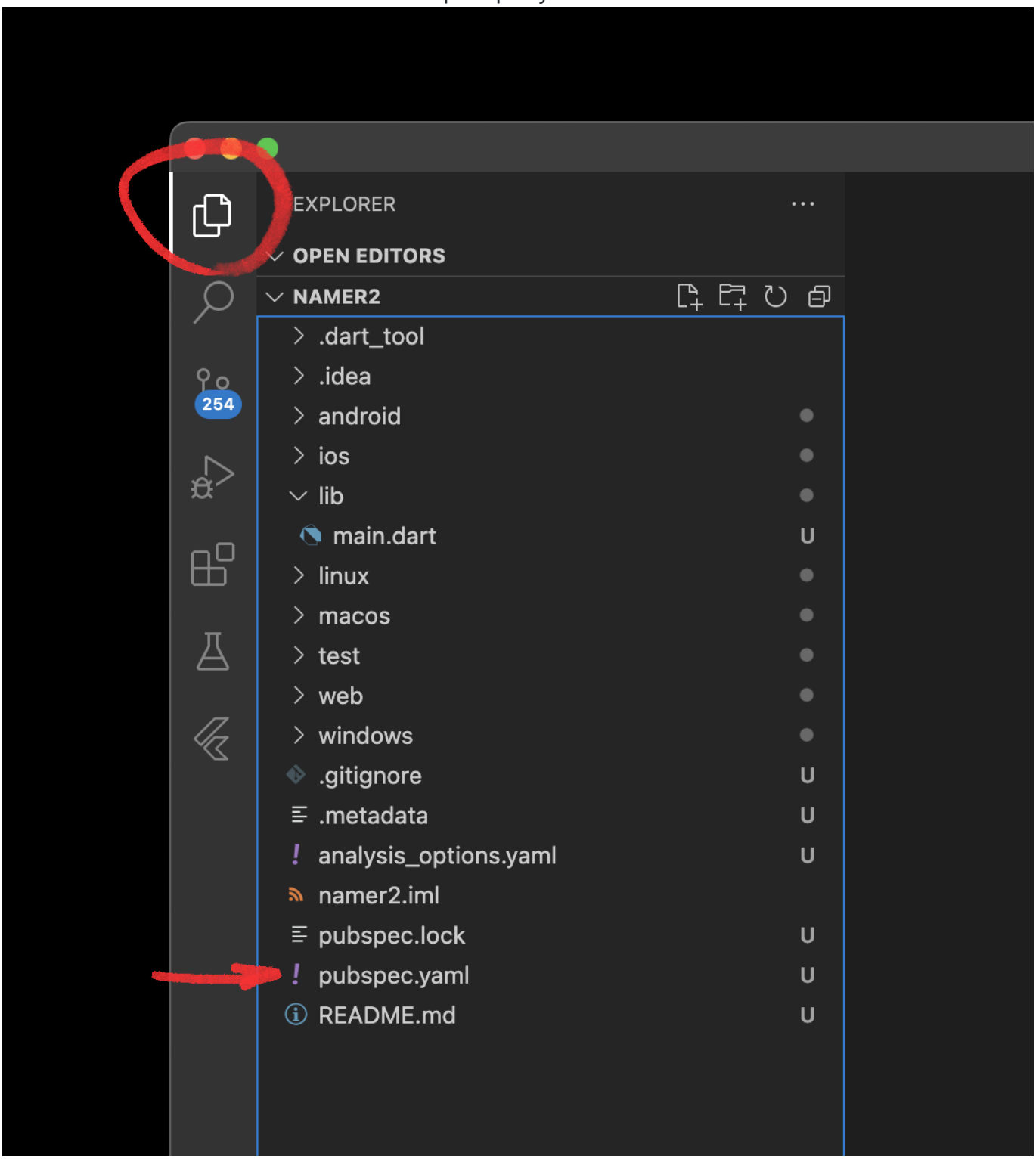
Oggi vogliamo creare un'app in cui se si preme un pulsante viene generata una stringa di caratteri formata da due parole del dizionario inglese.

Prepariamo i file del progetto

Ora sovrascriveremo il contenuto di 3 file con del codice di partenza per l'app.

pubspec.yaml

Iniziamo sostituendo il contenuto del file pubspec.yaml con il codice sotto:



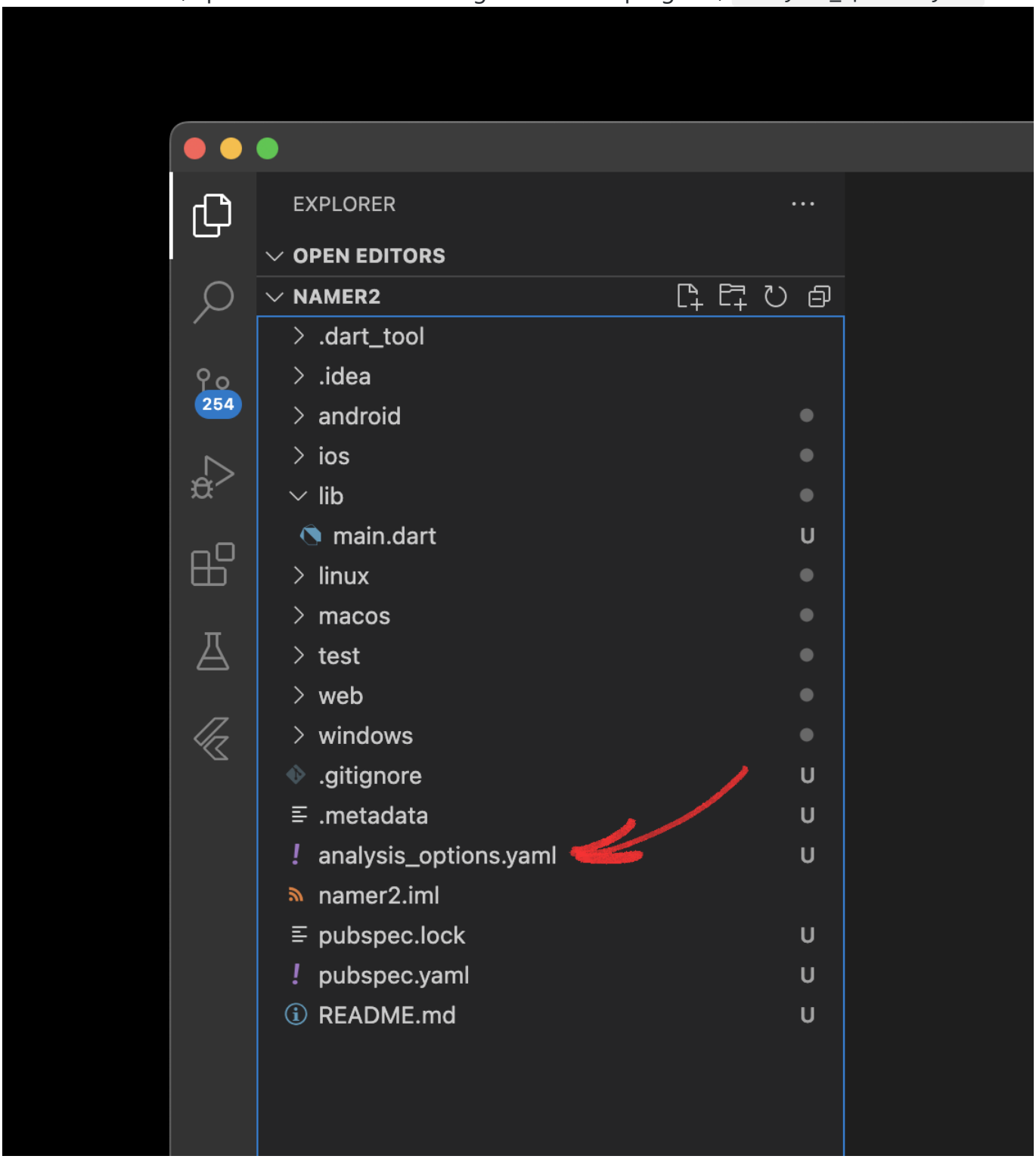
```
name: namer_app
description: A new Flutter project.

publish_to: 'none' # Remove this line if you wish to publish to pub.dev

version: 0.0.1+1

environment:
  sdk: ^3.1.1

dependencies:
  flutter:
    sdk: flutter

  english_words: ^4.0.0
  provider: ^6.0.0

dev_dependencies:
  flutter_test:
    sdk: flutter

  flutter_lints: ^2.0.0

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

Il file `pubspec.yaml` contiene le informazioni di base sulla tua app, come la versione corrente, le dipendenze con le loro versioni in uso e varie cose che non vedremo. Le dipendenze si potranno importare in un progetto semplicemente inserendo il nome e la versione di fianco.

Una volta sostituito il contenuto salvare.

analysis_options.yaml

Successivamente, apriamo l'altro file di configurazione nel progetto, `analysis_options.yaml`.



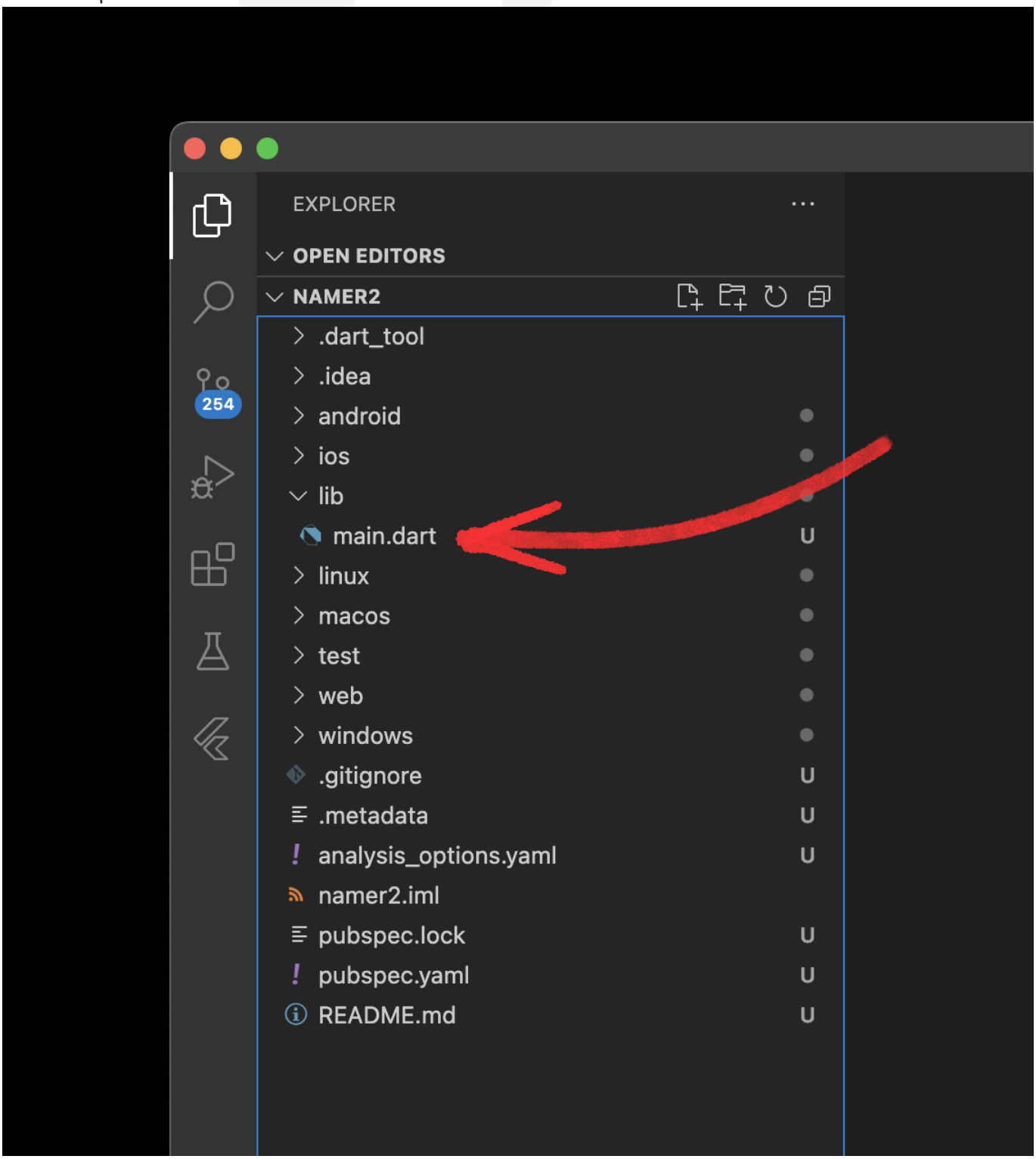
E sostituiamo il contenuto del file con questo.

```
include: package:flutter_lints/flutter.yaml

linter:
  rules:
    avoid_print: false
    prefer_const_constructors_in_immutables: false
    prefer_const_constructors: false
    prefer_const_literals_to_create_immutables: false
    prefer_final_fields: false
    unnecessary_breaks: true
    use_key_in_widget_constructors: false
```

Quello che abbiamo fatto con questa configurazione è sviluppare con Flutter in modo più rilassato senza che l'analizzatore statico del codice usi delle regole troppo restrittive. (A differenza dei test dinamici che vengono eseguiti) Sappiamo che gli analizzatori statici nel testing software vengono anche spesso usati per migliorare la qualità del codice, identificare bug e applicare migliori pratiche di programmazione, e che questi agiscono senza che venga eseguito il programma/codice. Essendo un primo progetto di prova quindi sarà rilassare questo analizzatore. Semmai doveste produrre un software e questo fosse scritto per essere pubblicato in modo vero e proprio allora potete rendere l'analizzatore più rigoroso di come abbiamo fatto.

Una volta sostituito il contenuto salvare.

Adesso apriamo il file `main.dart` nella cartella `lib`



E anche qui sostituiamo il contenuto di questo file con il codice seguente e salviamo il file

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

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

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (context) => MyAppState(),
      child: MaterialApp(
        title: 'Namer App',
        theme: ThemeData(
          useMaterial3: true,
          colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepOrange),
        ),
        home: MyHomePage(),
      ),
    );
  }
}

// Classe che rappresenta lo stato dell'app(Dati)
class MyAppState extends ChangeNotifier {
  // questo metodo restituisce la stringa della coppia di parole
  var current = WordPair.random();
}

class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    var appState = context.watch<MyAppState>();

    return Scaffold(
      body: Column(
        children: [
          Text('A random idea:'),
          Text(appState.current.asLowerCase),
        ],
      ),
    );
  }
}
```

Queste 50 righe di codice rappresentano l'intera app di partenza.

Sviluppiamo l'app

Primo Hot Reload

Avviamo nuovamente l'applicazione tramite l'icona in alto a destra o premiamo F5 per avviare l'app e aspettiamo qualche secondo prima che si avvi. Per testare questa funzionalità famosa di Flutter, ovvero l'**hot reload** modifichiamo il parametro dell'oggetto Text() in una nuova stringa

```
// ...

    return Scaffold(
      body: Column(
        children: [
          Text('A random AWESOME idea:'),  // ← Example change.
          Text(appState.current.asLowerCase),
        ],
      ),
    );

// ...
```

dopo aver modificato la stringa nell'oggetto del primo `Text()` se salverete il file (con `Ctrl+S` o `Cmd+S`) noteremo come l'app cambia immediatamente ma la parola casuale rimane la stessa. Quindi viene iniettata esattamente solo la modifica del widget modificato e gli stati del resto degli altri widget resta invariato. Come notiamo la parola non cambia e questo significa che non viene ricompilato tutto il codice(la parola viene generata nello stato dell'app definito con MyAppState). Notiamo quindi che questa funzionalità per i programmatori è un'ottimo modo per aumentare di molto il workflow dello sviluppo di un'app.

Aggiungiamo un Pulsante

Sempre nel main aggiungiamo il bottone nella parte inferiore del Widget Column(il widget che incolonna una lista di Widget).

```
// ...

    return Scaffold(
      body: Column(
        children: [
          Text('A random AWESOME idea:'),
          Text(appState.current.asLowerCase),

          // ↓ Aggiungi questo.
          ElevatedButton(
            onPressed: () {
              print('button pressed!');
            },
            child: Text('Next'),
          ),

        ],
      ),
    );

// ...
```

Salvando la modifica, l'app si aggiorna nuovamente: viene visualizzato un pulsante e, quando fai clic su di esso, la *console di debug* in VS Code mostra un **button pressed!** come stampa di debug.

Corso accelerato ma capiamo bene

Per quanto questo sia un piccolo corso accelerato di Flutter è importante dare un'occhiata al codice

```
// ...

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

// ...
```

Nella parte superiore del file troveremo la funzione `main()`. Che dice a Flutter solo di eseguire l'app definita in `MyApp` che è un Widget.

```
// ...

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (context) => MyAppState(),
      child: MaterialApp(
        title: 'Namer App',
        theme: ThemeData(
          useMaterial3: true,
          colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepOrange),
        ),
        home: MyHomePage(),
      ),
    );
  }
}

// ...
```

La classe `MyApp` estende `StatelessWidget` che è il modo per caratterizzare una classe in modo che sia visto come un Widget da Flutter. I widget sono gli elementi da cui crei ogni app Flutter. Come puoi vedere, anche _MyApp _ è un widget.

Il codice `MyApp` configura l'intera app ed è abbastanza standard. Qui dentro solitamente viene creato lo stato dell'app (di cui parleremo dopo), gerarchicamente questo è la radice di tutti gli altri Widget(L'app non è altro che un albero di Widget). In `MyApp` si assegna un nome all'app, si definisce il tema visivo e si imposta il widget "home", come il punto di partenza della tua app. Ad esempio come punto di partenza ci possiamo inserire un Widget Homepage che abbiamo creato.

Stato app

Lo stato dell'app(business logic) è contenuto in:

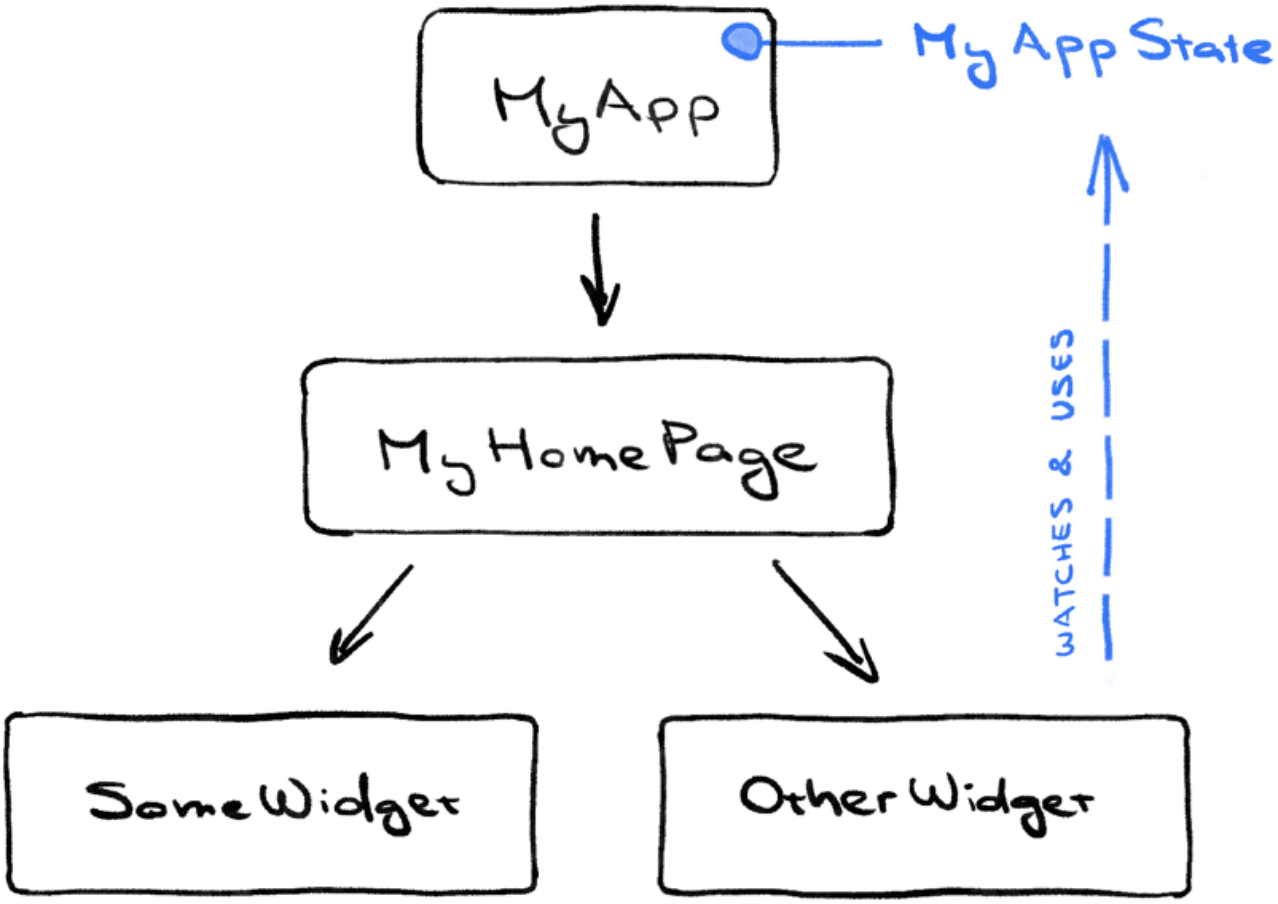
```
// Classe che rappresenta lo stato dell'app(Dati)
class MyAppState extends ChangeNotifier {
  // questo metodo restituisce la stringa della coppia di parole
  var current = WordPair.random();
}
```

Noi che siamo programmatori sappiamo che per il bene del software, della sua gestione e manutenzione dovremmo separare la logica di business (business logic) dalla presentazione (layout).

Questa dimostrazione a cui partecipate resterà molto semplice ma sappiate che ci sono molti modi per gestire lo stato di un'app in flutter (solitamente le aziende o i gruppi di sviluppatori ne stabiliscono uno da usare)

Noi useremo `ChangeNotifier` che è uno dei più facili da spiegare

- In questo momento la classe `MyAppState` definisce i dati necessari per il funzionamento dell'app. Al momento contiene solo una singola variabile con l'attuale coppia di parole casuali.
- La classe `MyAppState` estende `ChangeNotifier`, il che significa che può *notificare* ad altri i propri *cambiamenti* . Ad esempio, se la coppia di parole corrente cambia, alcuni widget nell'app che noi scegliamo vogliamo che lo sappiano.
- Lo stato viene creato e fornito all'intera app utilizzando un `ChangeNotifierProvider` (vedere il codice sopra in `MyApp`). Ciò consente a qualsiasi widget innestato nell'app di accedere ai dati dello stato dell'app.



Widget Homepage

```
// ...

class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {           // ← 1
    var appState = context.watch<MyAppState>();  // ← 2

    return Scaffold(                             // ← 3
      body: Column(                              // ← 4
        children: [
          Text('A random AWESOME idea:'),        // ← 5
          Text(appState.current.asLowerCase),    // ← 6
          ElevatedButton(
            onPressed: () {
              print('button pressed!');
            },
            child: Text('Next'),
          ),
        ],                                       // ← 7
      ),
    );
  }
}

// ...
```

Abbiamo `MyHomePage` che è il widget che abbiamo creato e che abbiamo scelto come punto di partenza della nostra app. Ogni riga numerata sottostante corrisponde a un commento con numero di riga nel codice sopra:

1. Ogni widget definisce un `build()` metodo che viene chiamato automaticamente ogni volta che le circostanze del widget cambiano in modo che il widget sia sempre aggiornato.
2. `MyHomePage` il Widget che stiamo creando vogliamo che osservi le modifiche che vengono effettuate sullo stato utilizzando il metodo `watch()` per farlo teniamo traccia di queste modifiche in questo modo.
3. Ogni widget, in questo caso `MyHomePage` ha un metodo `build` che deve restituire un widget o (più tipicamente) un *albero* annidato di widget. In questo caso, il widget di livello superiore è `Scaffold`. Non useremo lo `Scaffold` in questo codelab, ma è un widget utile e si trova nella stragrande maggioranza delle app Flutter.
4. `Column` è uno dei widget di layout più basilari in Flutter. Prende un numero qualsiasi di figli e li incolonna in una tutti dall'alto verso il basso. Ovviamente potremo modificare i parametri del widget che spesso sono progettati per modificare la visualizzazione del widget, ad esempio potremmo cambiare il modo in cui la colonna è centrata.
5. `Text` è quello che abbiamo modificato prima.

6. Questo secondo `Text` widget come parametro prende `appState` e accede all'unico membro di quella classe, ovvero `current` che per ora è l'unica variabile dello stato della nostra app in Flutter. Utilizza un getter `asLowerCase` (il getter è come un metodo che restituisce un tipo di dato) che restituisce la stringa della coppia di parole in minuscolo.
7. Notiamo come il codice Flutter fa un uso molto massiccio di virgole finali. Non è necessario che questa particolare virgola sia qui, perché button è l'ultimo elemento della lista dei figli di `Column` . Tuttavia è generalmente buona norma utilizzare le virgole finali.

Primo comportamento

Adesso, collegheremo il pulsante allo stato dell'app, facendone cambiare i valori dello stato e ridisegnando i Widget che osservano lo stato dell'app. Torniamo nella sezione di codice che rappresenta lo stato dell'app ovvero all'interno di `MyAppState` e aggiungiamo questa porzione di codice

```
// ...

class MyAppState extends ChangeNotifier {
  var current = WordPair.random();

  // ↓ Add this.
  void getNext() {
    current = WordPair.random();
    notifyListeners();
  }
}

// ...
```

Il nuovo `getNext()` metodo riassegna alla variabile `current` una nuova parola casuale `WordPair` . Una volta assegnata la parola chiama anche `notifyListeners()` (un metodo `ChangeNotifier)` che garantisce che chiunque guardi `MyAppState` venga avvisato. In questo caso chi osserva `MyAppState` è la variabile `var appState = context.watch<MyAppState>();` che verrà avvisata. Subito dopo verrà ridisegnata l'app a partire dal Widget radice, scendendo fino ai Widget foglia. E in questo caso verranno ridisegnati i Widget con le nuove modifiche osservate nello stato dell'app `MyAppState` .

Tornando al nostro bottone quello che facciamo adesso in questo piccolo pezzo di codice sarà chiamare il metodo `getNext` dal parametro di tipo callback del pulsante . In Flutter, il parametro `onPressed` nei widget dei bottoni è utilizzato per specificare la funzione o il blocco di codice che deve essere eseguito quando il pulsante viene premuto (cliccato).

```
// ...

    ElevatedButton(
      onPressed: () {
        appState.getNext();  // ← This instead of print().
      },
      child: Text('Next'),
    ),

// ...
```