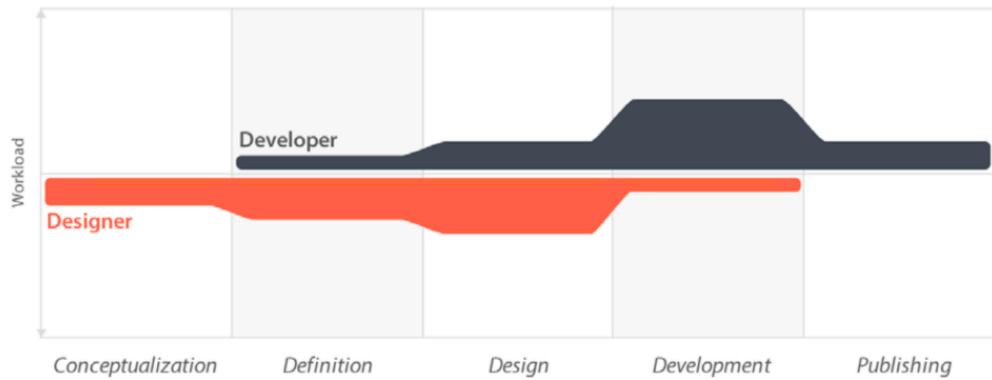


1 Introduzione

Processo di creazione di applicazioni software che possono essere eseguite su dispositivi mobile. Le fasi di progettazione e sviluppo sono differenti rispetto alle classiche applicazioni desktop:

- Estremamente diverso dallo sviluppo di applicazioni desktop
- User Interface (*UI*) e User eXperience (*UX*) *mobile first*



1. **Concettualizzazione:** idea di un'applicazione + check sulla fattibilità;
2. **Definizione:** si definisco gli end user, le funzionalità;
3. **Design:** si creano i wireframe e i visual design;
4. **Sviluppo:** si sviluppa l'applicazione tramite programmazione e si effettuano i test;
5. **Pubblicazione:** deploy (web app) o pubblicazione su store (app mobile), si garantiscono eventuali aggiornamenti, ecc...

1.1 Tipologie di app

Stack moderni per mobile programming

- **Native App Development**

- iOS: Swift, Xcode, iOS SDK, UIKit
- Android: Java/Kotlin, Android Studio, Android SDK, Android framework

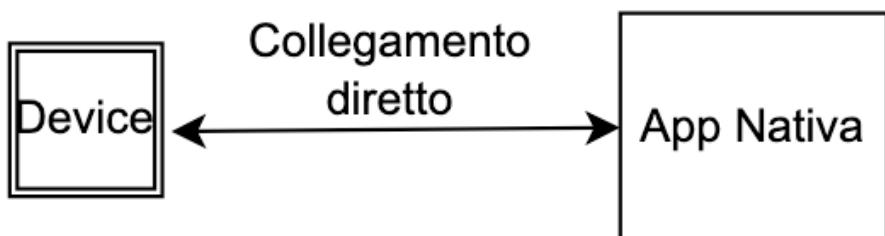
- **Web app**

- HTML, CSS, JavaScript
- UI framework: React, Vue.js, Svelte
- Container-like: Cordova, Ionic, PhoneGap

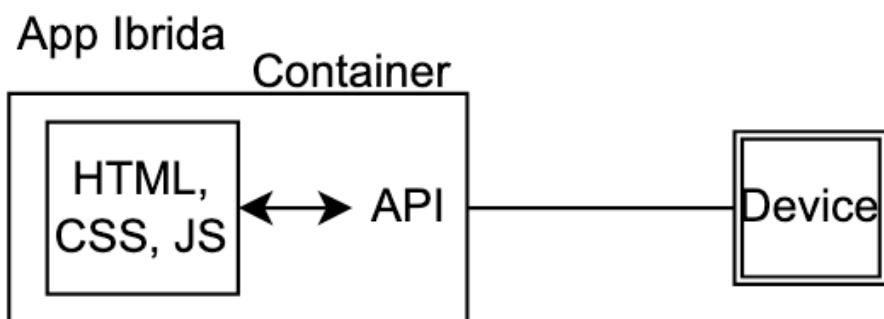
- **Cross-platform**

- Xamarin (outdated)
- React Native
- Flutter

- **App native:** app sviluppate per un particolare sistema operativo, utilizzando linguaggi di programmazione **nativamente supportati** e gli strumenti messi a disposizione dalla piattaforma stessa (Swift/Objective C e Xcode per ios, java/kotlin e android Studio per android). Garantiscono le massime prestazioni, avendo diretto accesso al device... ma il costo di sviluppo e mantenimento è molto elevato;



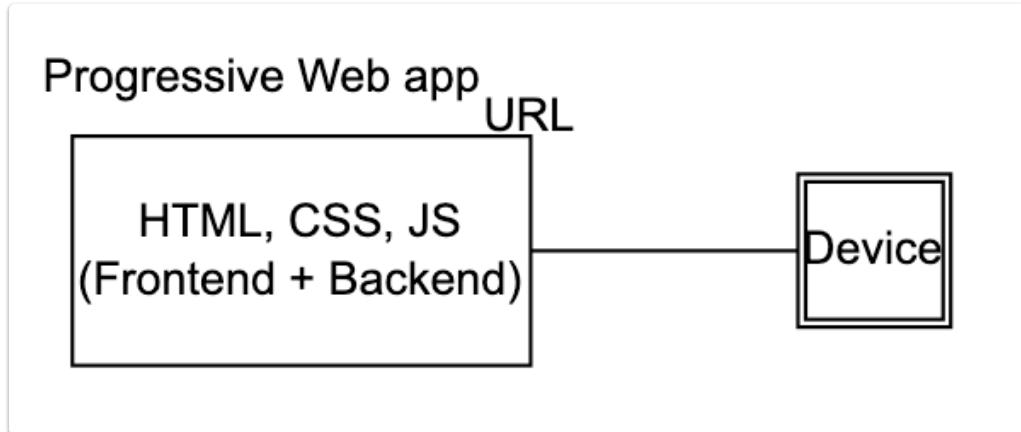
- **PWA - Progressive Web App:** applicazioni web basate su stack moderni. L'idea è quella di fornire un'esperienza simile a quella delle app native. Funzionano su qualunque device che presenta un **browser** standard. Le pwa possono anche essere containerizzate (app ibride) mediante framework appropriati che intermediano le funzionalità offerte dal sistema operativo;



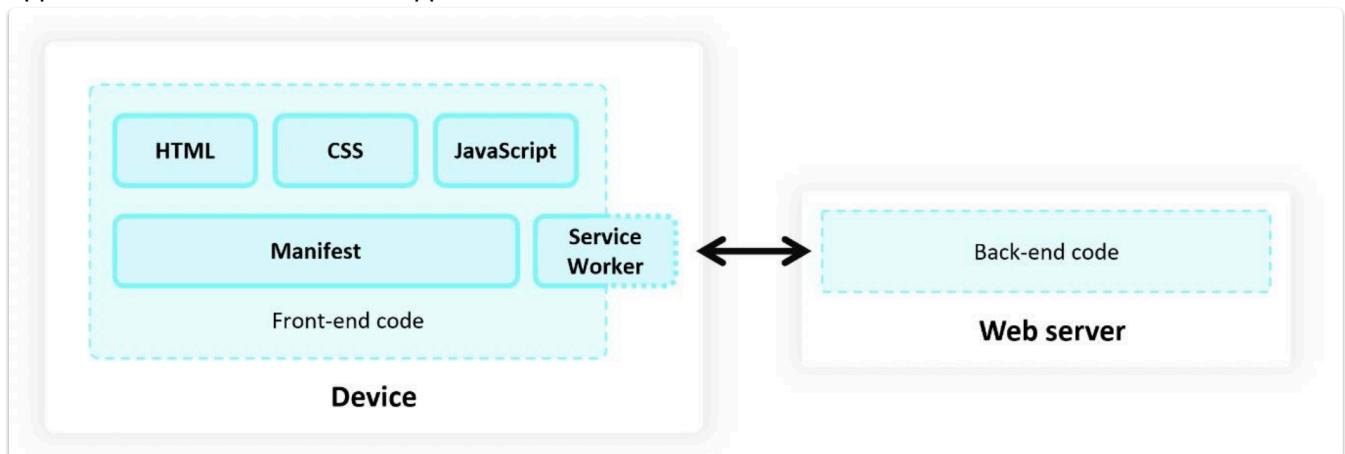
- **App Cross Platform:** combinano elementi delle app native e delle web app, generalmente sviluppate in javascript (react-native) o dart (flutter). Il codice scritto dal programmatore sarà poi convertito

in codice nativo per la specifica piattaforma;

2 PWA - Progressive Web App

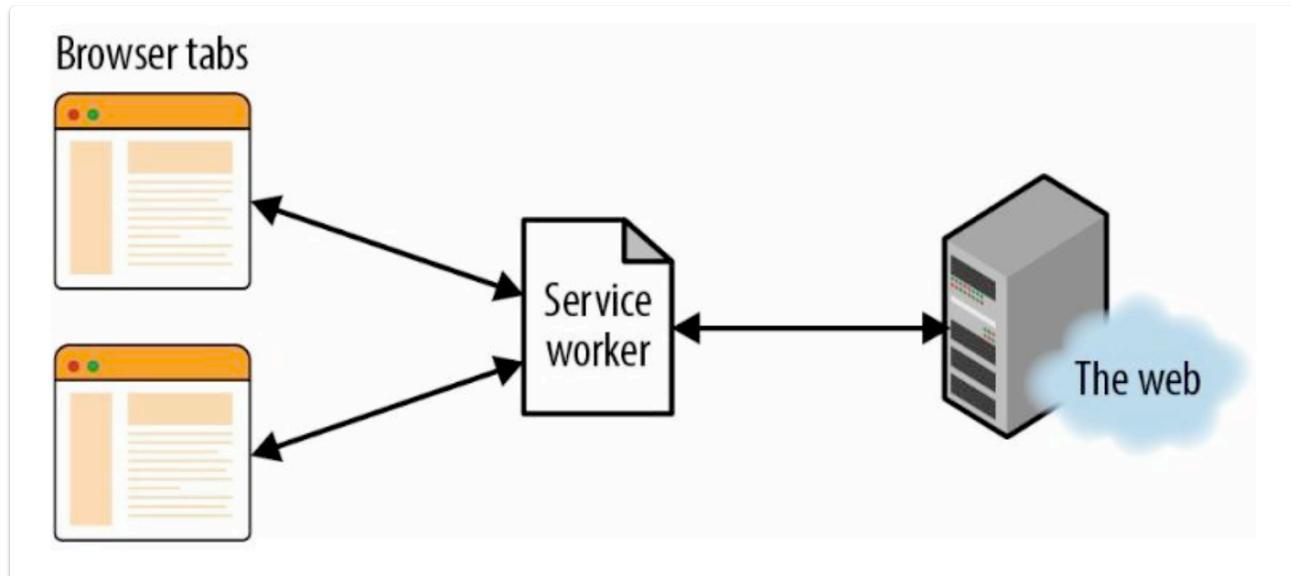


Applicazioni realizzate con tecnologie **web-oriented**. Si adattano alle capacità del device, possono essere eseguite in un **web browser** e possono essere **installate**. Una volta installate funzionano come delle normali applicazioni. Essendo delle web application offrono un **accesso limitato** alle features del device.



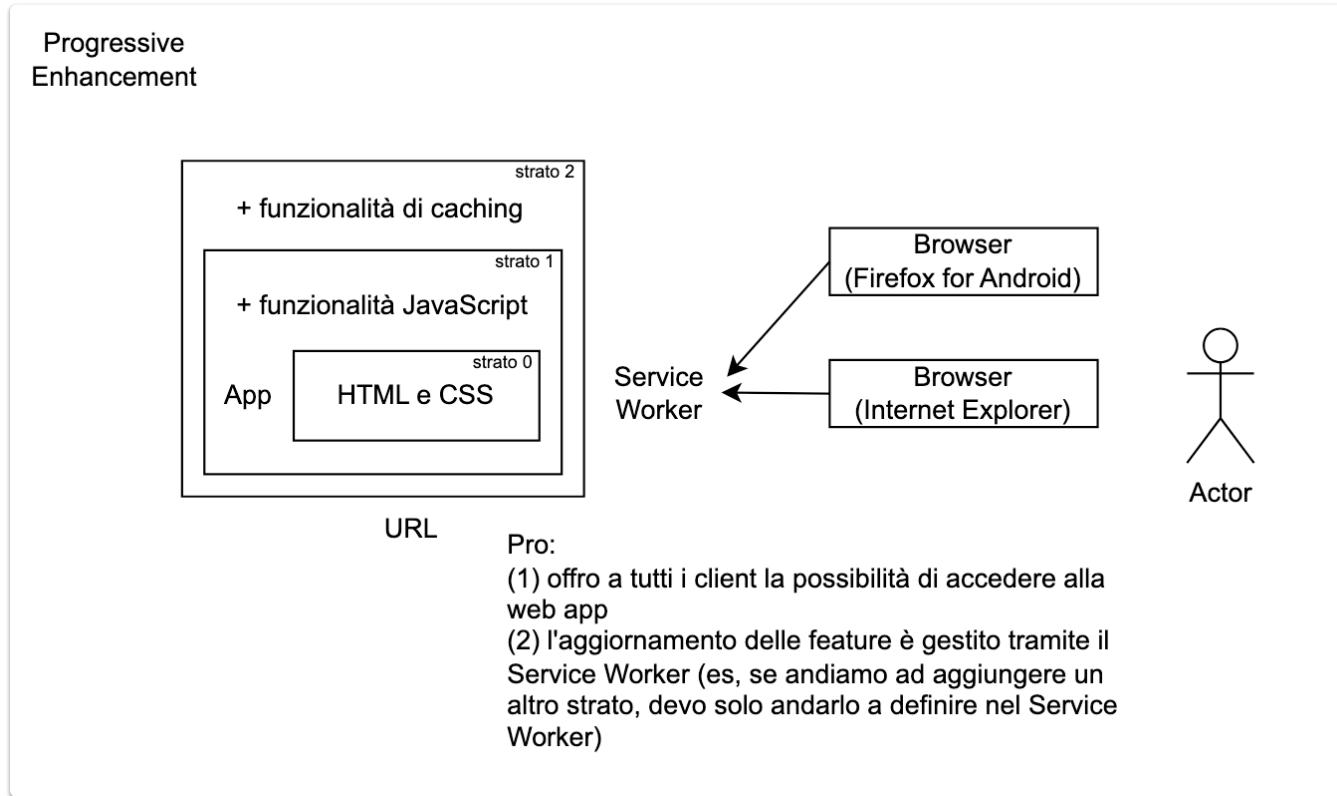
Generalmente una web-app si compone di:

1. **Back-end code**: endpoint necessari all'applicazione a recuperare il contenuto dinamico (`node.js`, `java`, `PHP`, ...);
2. **Front-end code**: risorse necessarie all'applicazione per poter essere installata sul device (`HTML`, `CSS`, `Javascript` e `json manifest`). Il `json manifest` serve a fornire una descrizione schematica (metadati) dell'applicazione all'app OS;
3. **Service worker**: specializzazione di un **Web Worker**: intercetta le richieste di rete e consente di implementare supporto offline, caching ed eseguire task in background. È necessario per l'installazione di una PWA. Sostanzialmente è uno script che controlla una o più pagine e vive in un layer intermediario tra il client (browser) e il server. È **indipendente dalla connessione**.



Supportano un metodo di sviluppo conosciuto come **Progressive Enhancement**, una strategia di sviluppo che consiste nel costruire l'applicazione a **strati**, partendo da una base semplice e accessibile a tutti, e aggiungendo funzionalità avanzate solo se il browser o il dispositivo dell'utente le supporta.

Si parte da un nucleo solido, contenente le **funzionalità basilari** dell'applicazione e funzionante anche su piattaforme **obsolete**... successivamente si aggiungono strati ulteriori che implementano funzionalità aggiuntive. Ogni utente vede e usa tutte le funzionalità che il suo **browser permette**, senza perdere l'accesso ai contenuti principali.



2.1 Web App Manifest

File JSON contenente i **metadati** dell'applicazione e necessario per l'**installazione** della PWA. Può essere in formato `.json` o `.webmanifest` e il contenuto deve essere `json` valido. I seguenti membri sono necessari:

1. `manifest_version`
2. `version`

3. name

```
{  
  "name": "My Sample PWA",  
  "lang": "en-us",  
  "short_name": "SamplePWA",  
  "description": "A sample PWA for testing purposes",  
  "start_url": "/",  
  "scope": "/",  
  "display": "standalone",  
  "theme_color": "#2f3d58",  
  "background_color": "#2f3d58",  
  "orientation": "any",  
  "icons": [  
    {  
      "src": "/icon512.png",  
      "sizes": "512x512"  
    }  
  ]  
}
```

2.2 Service Worker

2.2.1 Creazione del Service Worker

Creazione di un Service Worker

1. Registriamo un Service Worker

- navigator: browser
- Il register restituisce una promise
 - Se la promessa è "mantenuta", tutto è andato a buon fine e si esegue il then
 - Altrimenti, il catch

2. Definiamo un Service Worker

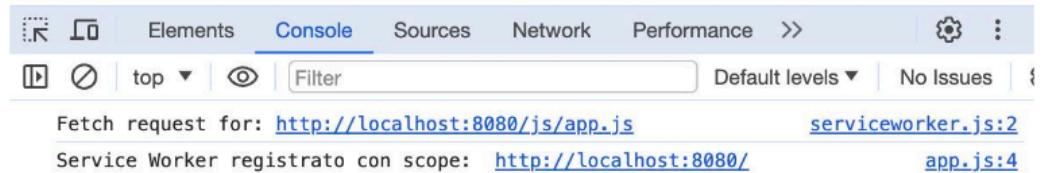
- Il listener cattura tutti gli eventi fetch
 - Gli eventi fetch sono eventi che vengono indirizzati al Service Worker (non alle normali pagine) e racchiudono le richieste effettuate dal browser

```
in /js/app.js
if ("serviceWorker" in navigator) {
  navigator.serviceWorker.register("/serviceworker.js")
    .then(function(registration) {
      console.log("Service Worker registrato con scope:",
        registration.scope);
    }).catch(function(error) {
      console.log("Registrazione Service Worker fallita:",
        error);
  })
}
```

```
in /serviceWorker.js
self.addEventListener("fetch", function(event) {
  console.log("Richiesta fetch per:", event.request.url);
});
```

4. Accediamo alla root tramite HTTP

Hello, World!



Creazione di un Service Worker



5. Intercettiamo una richiesta col service worker e restituiamone una nuova

```
in /serviceWorker.js
self.addEventListener("fetch", function(event) {
  if (event.request.url.includes("style.css")) {
    event.respondWith(
      new Response(
        'h1 { color: red; font-size: 4em; }',
        { headers: { 'Content-Type': 'text/css' } }
      )
    )
  }
});
```

Hello, World!



6. Creiamo un proxy "identità": restituisce la risorsa richiesta

```
in /serviceWorker.js
self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request)
  );
});
```

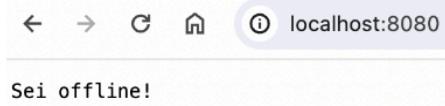
fetch restituisce la risposta includendola in una Promise.

L'API di quest'ultima fornisce due metodi:

- `then()`: se la Promise è *mantenuta*, si invoca questo metodo
- `catch()`: altrimenti, se vi è qualche problema, si invoca questo metodo

7. Modifichiamo il service worker in modo da restituire una nuova risposta se siamo offline

```
in /serviceWorker.js
self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return new Response("Sei offline!")
    })
  );
});
```



Il service worker ha uno scope limitato dettato dalla sua posizione, a livello di file, rispetto alla radice:

- se si trova sotto `/`, allora controlla tutte le richieste che originano in qualunque parte dell'app;
- se si trova in `/dir/` allora controlla le richieste che originano nella directory `/dir/`;

2.2.2 Caching - CacheStorage API

L'implementazione di un layer di caching può essere effettuata mediante un'API chiamata CacheStorage API : non è la cache del browser né l'application cache! La combinazione Service Worker e CacheStorage permette di far funzionare l'app offline.



- Ciclo di vita di un Service Worker
- Eventi fetch: catturati da un Service Worker nello stato *attivo*
- Per fare caching, catturiamo l'evento `install`
 - Avviene una sola volta
 - Subito dopo il `register`, e subito prima dell'attivazione
 - Permette di fare caching di tutti i file che vogliamo rendere disponibili offline
 - È utile anche per arrestare l'attivazione di un Service Worker nel caso in cui vi siano problemi
 - Se qualcosa va storto durante il caching, possiamo arrestare l'attivazione
 - Alla prossima richiesta, il browser riproverà ad attivare il Service Worker



```
// MEMORIZZAZIONE - in serviceWorker.js
self.addEventListener("install", function(event) {
  // waitUntil consente di estendere il tempo di vita dell'evento fin quando la Promisse non si realizza
  event.waitUntil(
    // caches è l'endpoint dell'API
    // open consente di accedere alla cache con il nome specificato
    // la cache è una mappa
    caches.open("la-mia-cache").then(function(cache) {
      return cache.add("/index-offline.html");
    })
  );
});
```

```
// RECUPERO - in serviceWorker.js
self.addEventListener("fetch", function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      // restituzione in caso la richiesta non vada a buon fine
      // match(X) verifica l'esistenza della chiave 'X' in una
      // qualunque cache dell'applicazione
      return caches.match("/index-offline.html")
    })
  );
});
```

```
// Caching di piú risorse
var CACHED_URLS = [
  "/index-offline.html",
  "/css/style.css",
]
self.addEventListener("install", (event) => {
  event.waitUntil(
    caches.open("la-mia-cache").then((cache) => {
      return cache.addAll(CACHED_URLS);
    })
  );
});
```

```

);
});

// Recupero con matching specifico
self.addEventListener("fetch", (event) => {
    event.respondWith(
        fetch(event.request).catch(() =>{
            return caches.match(event.request).then((response) => {
                if (response)
                    return response;
                else if (event.request.headers.get("accept").includes("text/html"))
                    return caches.match("/index-offline.html");
            });
        })
    );
});

```

3 Javascript e Node.js

Linguaggio ad alto livello interpretato con supporto agli **oggetti** e alla programmazione **funzionale**. Il core di js definisce le API mini per lavorare con i dati primitivi, il resto (eg. I/O) è a cura dell'host enviroment.

Per l'interfacciamento con il sistema operativo è possibile utilizzare alcuni **framework** tipo `node.js` o `deno`.

`Node.js` (server-side javascript o javascript runtime) è un'alternativa moderna alla creazione di shell script, il quale include l'eseguibile `node` e il package manager `npm`.

Node presenta una console integrata chiama `REPL`.

3.1 Eseguire un programma tramite Node

1. Start up: Node inizializza l'ambiente di runtime
2. Load and parse: il file viene parserizzato, il codice sorgente è convertito in una rappresentazione comprensibile per l'engine V8 di Javascript contenuto in Node;
3. Execution: il codice javascript viene eseguito linea dopo linea;
4. Exit: dopo l'ultima riga, Node effettua un clean delle operazioni in pending (eg. I/O) e termina il processo

3.2 Programmazione asincrona in JS

JS è event-driven:

- Client-side: si aspetta l'interazione da parte dell'utente
- Server-side: si aspetta le richieste

La programmazione asincrona si fonda sul concetto di **promises**, oggetti che rappresentano un risultato *not-yet-available* di un'operazione asincrona. Le promise possono esistere in 4 stati diversi:

- **Fulfilled** (mantenuta, soddisfatta): l'operazione è andata a buon fine;
- **Rejected** (non mantenuta, rigettata): l'operazione non è andata a buon fine;
- **Pending** (in attesa): operazione né fulfilled né rejected

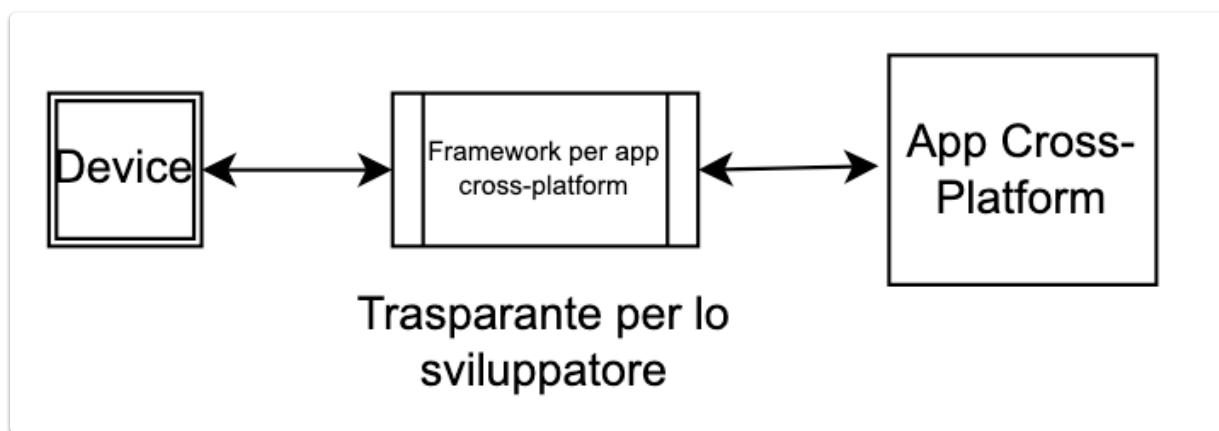
- **Settled** (completata): fulfilled o rejected

JavaScript: Promise

- Un esempio è l'API fetch
 - Si usa per fare richieste di rete
 - È parte delle Web API del browser, non di JavaScript
 - Restituisce una Promise
 - Se soddisfatta, restituisce una Response

```
fetch('https://prova.com/api/endpoint')
  .then(response => {
    if (!response.ok)
      throw new Error('Risposta di rete errata');
    return response.json();
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error('Errore:', error);
  });
});
```

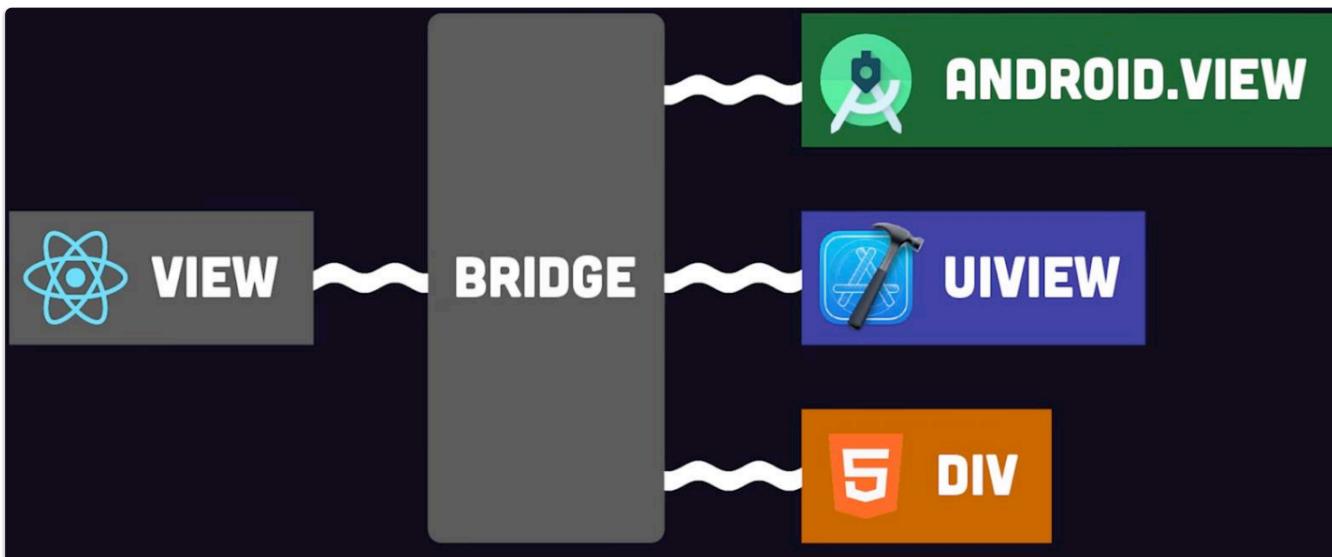
4 App Cross Platform



- React-Native
- Flutter

5 React-Native

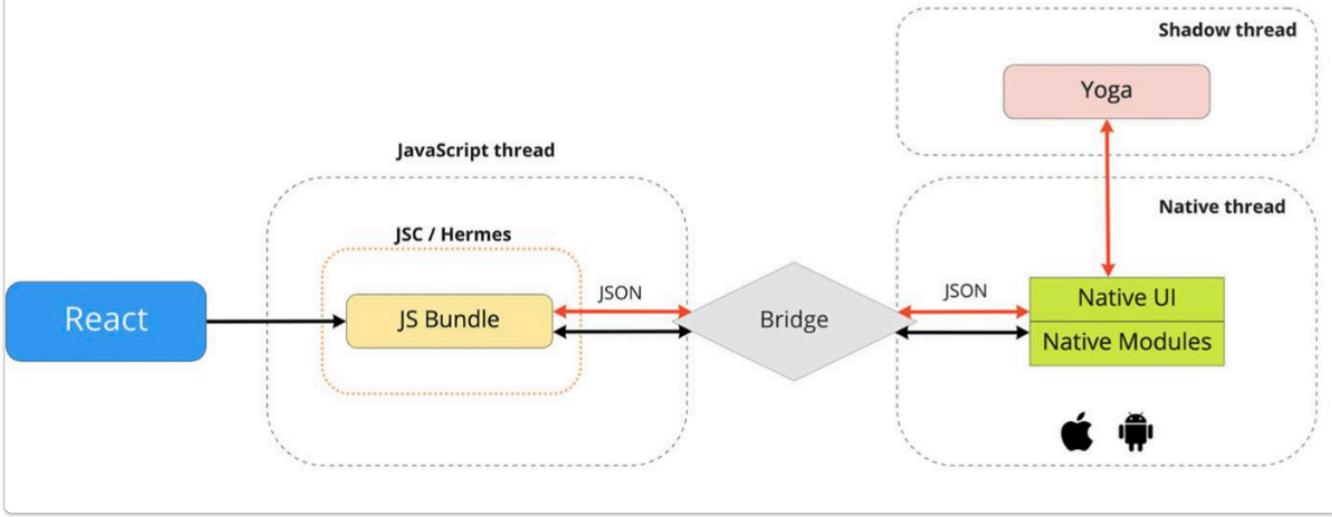
Framework sviluppato da **Meta** (basato su libreria `React`) per lo sviluppo di applicazioni native scrivendo codice `javascript`. A runtime, i componenti `react-native` vengono renderizzati nei corrispettivi **componenti nativi** della piattaforma target.



Adotta una filosofia *"Learn Once, Write Anywhere"*. Nonostante sia il framework più utilizzato per lo sviluppo mobile risulta essere leggermente immaturo rispetto a Cordova o Flutter e si tratta di un'astrazione costruita sulle API delle piattaforme target: se cambiano React-Native deve essere aggiornato.

5.1 Anatomia

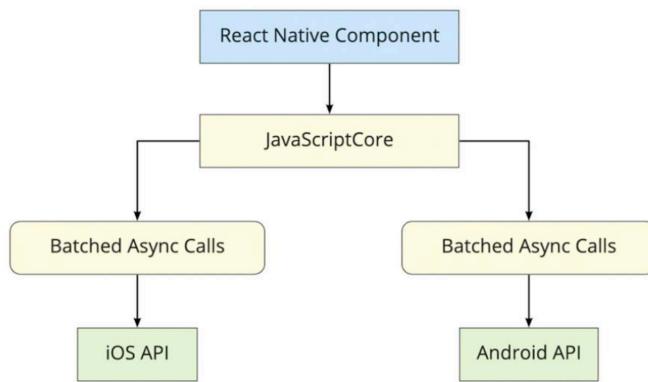
- Architettura di React Native



Under the hood



- Visione ad alto livello
 - La libreria React è presente nell'engine JavaScriptCore
 - React Native comunica asincronicamente con l'OS mobile
 - Questa comunicazione permette l'accesso alle API per i widget nativi (es, componenti)
 - React Native renderizza i componenti tramite queste API
 - Lo stesso vale per altri target, es tvOS, Android TV, Windows, macOS, e anche web



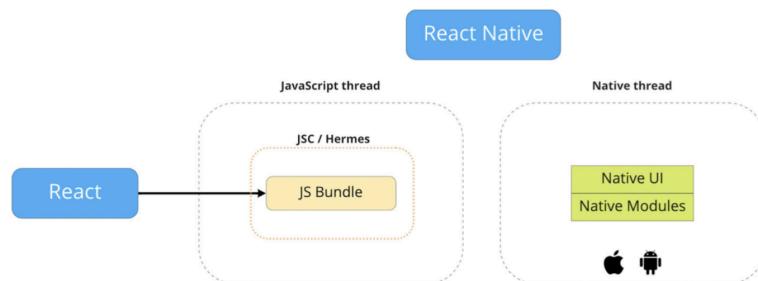
Il codice `js` viene eseguito da un motore primario chiamato *JavascriptCore*, il quale comunica con il sistema operativo mediante un bridge (*Batched Async Calls*). Il bridge invia messaggi batch asincroni alle API native del device target, le quali hanno il compito di creare e gestire i **widget nativi**.

- **La libreria React Native**

- Componenti nativi tradotti grazie a due livelli:
 - JavaScript thread
 - Native thread

- **Livello JavaScript**

- Contiene una JavaScript virtual machine (*Hermes*)
- Esegue il codice, invoca le API native del target, etc
- Vi è un singolo thread
 - La componente logica dell'app sarà eseguita in questo thread
- L'app è pacchettizzata in un singolo file tramite *Metro*
 - Traduce anche il codice JSX in JS

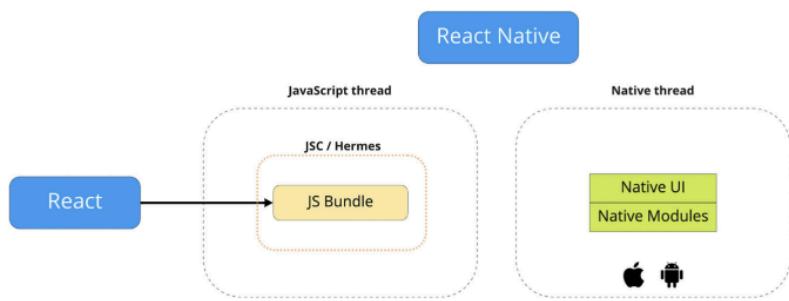


React-native è basato su 2 thread principali: *javascript thread* e *native thread*. Il javascript thread:

1. Il codice React/js viene pacchettizzato in unico file (`js bundle`) mediante il tool *metro*;
2. Il `js bundle` viene eseguito dalla JVM *Hermes* contenuta nel *js thread*;
3. Quando serve, il JS thread comunica con il **Native thread** tramite il **bridge**

- **Livello Native**

- Qui si esegue il codice nativo
- React Native implementa questa parte in codice nativo per ogni piattaforma (es, Java per Android)
- Composto da moduli che comunicano con l'SDK Android oppure iOS
- Espone una API che unifica le varie piattaforme, che può essere invocata dal JavaScript thread

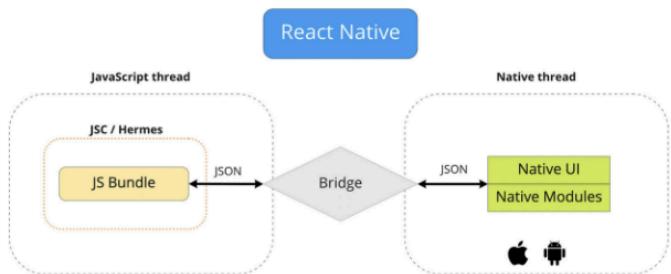


Il **native thread** invece esegue codice nativo e pertanto è implementato per ogni piattaforma. È composto da moduli che comunicano con il SO ed espone un'API che unifica le varie piattaforme che può essere invocata dal Javascript Thread.

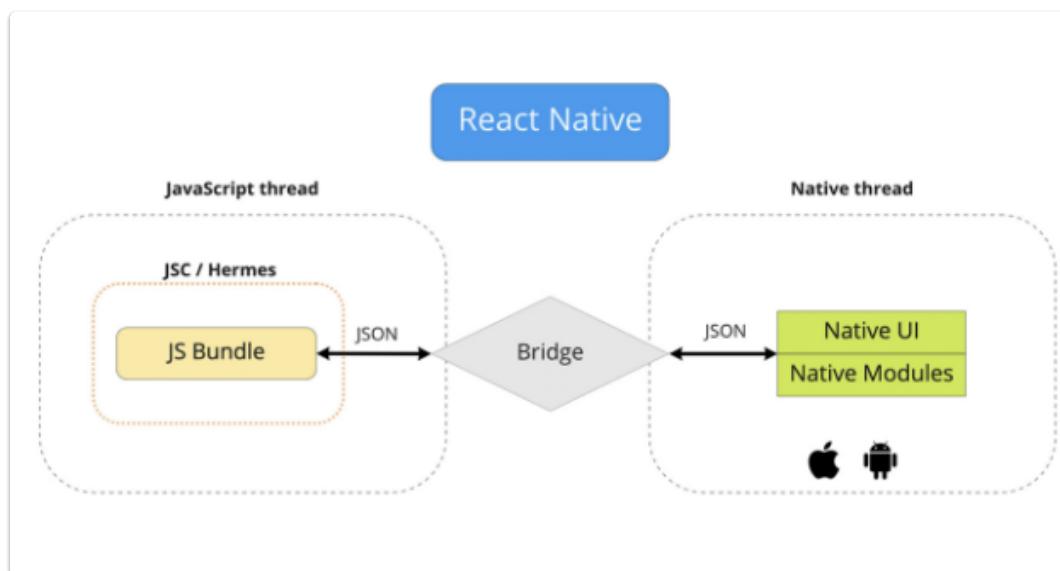
Under the hood



- Comunicazione tra i due livelli
 - Un bridge basato su una coda asincrona
 - I dati vengono serializzati, convertiti in JSON e passati nella coda; infine, vengono deserializzati
 - Simile ad una architettura client-server
 - Tutti gli eventi e le azioni si basano sui messaggi JSON asincroni
 - *Una singola codebase, diverse piattaforme*
 - *Bottleneck se l'app implementa business logic complicate*



I due livelli comunicano mediante un **coda**: i dati vengono serializzati, convertiti in `JSON` e passati alla coda, infine **deserializzati**. È una sorta di implementazione di architettura client-server e come tale presenta anche problematiche di bottleneck:

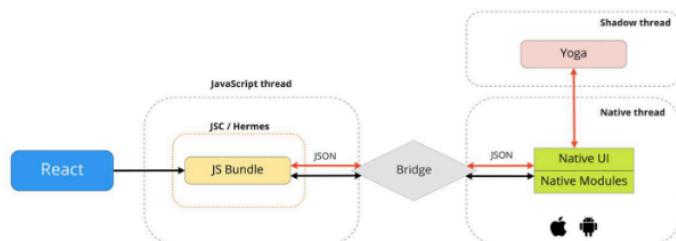


Supponendo di avere una lista molto grande di elementi da scrollare, quando l'evento `onScroll` avviene nel thread native, l'informazione è **comunicata asincronamente** al thread javascript... il primo tuttavia **non aspetta il feedback** del secondo e quindi potrebbe esserci un **ritardo** nel rendering degli elementi della lista.

La soluzione comune è quella di utilizzare i widget appositi per il rendering delle liste come `FlatList` oppure unificare l'interfaccia:

• Unificare l'interfaccia

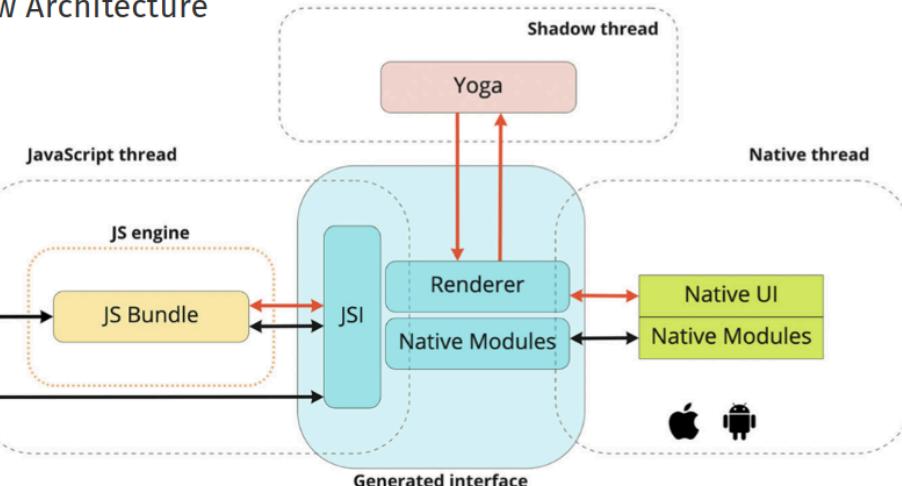
- In JS, abbiamo degli oggetti con delle proprietà di stile (*style property*)
- Per assicurare la consistenza, si ha un thread separato chiamato Shadow
 - Ricalcola il layout dell'applicazione usando l'engine Yoga
 - Tutti i calcoli relativi alla formazione dell'interfaccia dell'applicazione sono effettuati in questo thread
 - I risultati sono poi inviati al thread Native, responsabile del rendering



È previsto il rilascio di una nuova architettura di React-Native chiamata "*New Architecture*".

• Architettura (futura) di React Native

- Anche chiamata New Architecture



5.2 Pilastri fondamentali di React Native

1. Typescript
2. Core e Native Components
3. React in Native: JSX, Components, Props, State

5.2.1 Typescript

Estensione di Javascript sviluppata da Microsoft. Si tratta di un superset tipizzato di JS che integra un **compile-time type checker** basato su **static checking** (identificazione degli errori senza il bisogno di eseguire il programma) e **static type checking** (identificazione basta sul tipo di valori).

Il comportamento a runtime è preservato dal fatto che il compilatore TS crea codice **plain JS** senza informazioni sui tipi.

5.2.1.1 Definizione dei tipi

```

const user: User = {
  name: "Hayes",
  id: 0,
}
  
```

Il tipo viene espresso mediante un'interfaccia specificata dalla sintassi `: tipo`.

La stessa sintassi può essere usata per annotare i parametri di ritorno di una funzione:

```
function deleteUser(user: User){}

function getAdminUser(): User {}
```

Sono disponibili di default i seguenti tipi:

1. `boolean`
2. `bigint`
3. `null`
4. `number`
5. `string`
6. `symbol`
7. `undefined`
8. `any` : qualunque tipo
9. `unknown` : assicura che prima di poter essere usato si dichiari il tipo effettivo
10. `never` : valore che non può accadere
11. `void` : funzione che ritorna `undefined` o nessun tipo

```
const jsonParserUnknown = (jsonString: string): unknown => JSON.parse(jsonString);
const myOtherAccount = jsonParserUnknown(`{ "name": "Samuel" }`);
```

```
myOtherAccount.name;
```

Qui il tipo è `unknown`; non possiamo usarlo se il tipo non è stato ancora dichiarato a TypeScript. Questo assicura un design dei tipi up-front (es, se utilizzo una API, devo esplicitare i tipi di cui avrò necessità)

```
type User = { name: string };
```

```
const myUserAccount = jsonParserUnknown(`{ "name": "Samuel" }`) as User;
```

```
myUserAccount.name;
```

OK! `myUserAccount` è di tipo `User`

TypeScript: esempio never



```
type Shape = "circle" | "square" | "triangle";  
  
function getArea(shape: Shape): number {  
    switch (shape) {  
        case "circle":  
            return Math.PI * Math.pow(2, 2);  
        case "square":  
            return 4 * 4;  
        case "triangle":  
            return 0.5 * 3 * 4;  
        default:  
            const _exhaustiveCheck: never = shape;  
            return _exhaustiveCheck;  
    }  
}
```

Il tipo never è utile in diversi casi, tra il marking di **unreachable code**, ovvero segnare parti di codice che non dovrebbero mai essere eseguite.

Esempio: qui controlliamo il valore della variabile shape; il controllo è **esaustivo** (controlliamo tutti i possibili casi) e quindi non dovremmo **MAI raggiungere** il default.

5.2.1.1.1 Composizione di tipi

I tipi possono essere composti mediante le unions o i generics.

- Unions

```
type WindowStates = "open" | "closed" | "minimized";  
type LockStates = "locked" | "unlocked";  
type PositiveOddNumbersUnderTen = 1 | 3 | 5 | 7 | 9;
```

Union type – il valore può essere uno tra diversi tipi

type definisce alias di tipi; è simile all'interface ma

- Può definire union e intersection type
- Non è estendibile come una interface
- Può definire diversi alias per gli stessi tipi primitivi

```
function getLength(obj: string | string[]) {  
    return obj.length;  
}
```

Il tipo di una variabile x può essere ottenuto tramite typeof x

• Generics

```
type StringArray = Array<string>;
type NumberArray = Array<number>;
type ObjectWithNameArray = Array<{ name: string }>;

// Possiamo anche definire i nostri tipi
// affinché utilizzino generics
interface Backpack<Type> {
  add: (obj: Type) => void;
  get: () => Type;
}

// declare indica al compilatore che la definizione
// della variabile non avviene in questo scope ma
// da qualche altra parte (es, un file che importiamo)
// In pratica, è una definizione senza implementazione
declare const backpack: Backpack<string>;
// myType qui è una stringa
const myType = backpack.get();
```

5.2.1.1.2 Duck Typing

TypeScript supporta [duck typing](#) quindi l'equivalenza dei tipi è basata sulla loro [struttura](#). Se due oggetti hanno la stessa struttura, allora sono considerati dello stesso tipo.

```
interface Point {
  x: number;
  y: number;
}

// Accetta un parametro di tipo "Punto"
function logPoint(p: Point) {
  console.log(`x: ${p.x}, y: ${p.y}`);
}

// definisco un oggetto con 2 attributi: x, y (NON È DI TIPO PUNTO!)
const point = { x: 12, y: 26 };
logPoint(point); // Nessun errore!

const point2 : Point = { x: 15, y: 10 };
logPoint(point2);
```

5.2.2 React - Componenti Core e Nativi

Un [componente](#) è un pezzo di codice riusabile ed estensibile ed è l'elemento base su cui si fonda [React](#). La [View](#) è l'elemento base della programmazione mobile, un elemento rettangolare in grado di mostrare del contenuto. Tutto è una view!

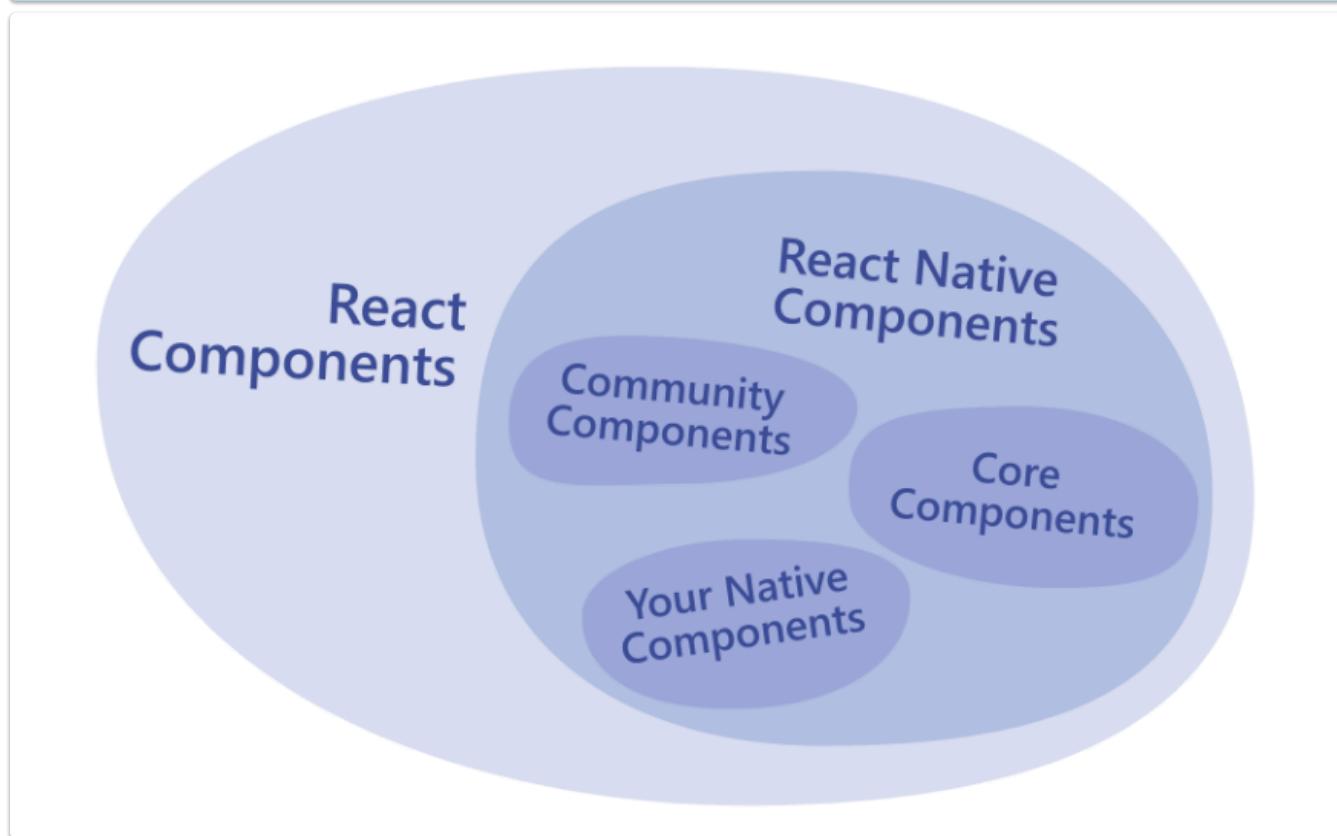
In [React Native](#), le view vengono definite e gestite tramite [Javascript](#) usando i [componenti React](#).

- A **runtime**, React Native interpreta (tramite js bridge) questi componenti e **crea dinamicamente le view native** corrispondenti per la piattaforma in uso (iOS o Android).

I componenti React Native (come `<View>` e `<Text>`) sono denominati **Native Components** perché rappresentano l'interfaccia Javascript diretta per interagire con e controllare le view native sottostanti. I Native Components vengono convertiti dal JS bridge nei componenti del device target.

React-Native fornisce un set di Native Component chiamati **Core Component**.

| RN Native UI Component | View Android | View iOS | Su Web | Descrizione |
|---------------------------------|---------------------------------|-----------------------------------|---|--|
| <code><View></code> | <code><ViewGroup></code> | <code><UIView></code> | Un <code><div></code> non scrollabile | Un container che supporta layout in flexbox, gestione del tocco e controlli di accessibilità |
| <code><Text></code> | <code><TextView></code> | <code><UITextView></code> | <code><p></code> | Visualizza stringhe di testo e gestisce alcuni eventi legati al tocco; comprende lo stile |
| <code><Image></code> | <code><ImageView></code> | <code><UIImageView></code> | <code></code> | Visualizza una immagine |
| <code><ScrollView></code> | <code><ScrollView></code> | <code><UiScrollView></code> | <code><div></code> | Un container generico che supporta lo scrolling |
| <code><TextInput></code> | <code><EditText></code> | <code><UITextField></code> | <code><input type="text"></code> | Casella di testo |



5.2.2.1 Anatomia di un componente

Un componente è una funzione che restituisce un elemento React .

```
// La funzione ritorna il componente "Text"
const Hello = () => {
  return <Text>Hello, World!</Text>
}
```

```

const world = () => 'World';
const Hello = () => {
  const hello = 'Hello';
  return <Text>{hello}, {world()}!</Text>;
}

```

La sintassi del `return` è **JSX**, un'estensione di Javascript che permette di scrivere elementi di markup nel codice JS. Gli oggetti verranno poi trasformati in plain Javascript.

5.2.2.2 Anatomia di un'app

```

import React from 'react';
import {Text} from 'react-native';

const world = () => 'World';
const Hello = () => {
  const hello = 'Hello';
  return <Text>{hello}, {world()}!</Text>;
}

export default Hello;

```

`export <nome_componente>` permette di "esportare" il componente e renderlo **disponibile** all'interno dell'applicazione.

`export default <nome_componente>` rappresenta l'esportazione **principale**, ovvero "l'oggetto più importante" che quel file offre.

5.2.2.3 Composizione e riuso di componenti

```

import React from 'react';
import {Text, View} from 'react-native';
const World = () => {
  return <Text>World again!</Text>;
};
const Hello = () => {
  return (
    <View>
      <Text>Hello, World!</Text>
      <World />
      <World />
      <World />
    </View>
  );
};
export default Hello;

```

`<View>` contiene e renderizza `<Text>` e `<TextInput>`. Un componente che renderizza altri componenti viene detto **parent component**; il componente renderizzato è detto **child component**.

5.2.2.4 Props

Le **props** sono proprietà dei componenti che permettono la loro **personalizzazione**. Rappresentano un oggetto a cui il componente può accedere.

```

// definizione della prop
type PlanetProps = {
  name: string;
};

// componente figlio, accetta un unico argomento chiamato "props", di tipo
// PlanetProps
const Planet = (props: PlanetProps) => {
  return (
    <View>
      <Text>
        Hello,
        {props.name}! // accedo al campo name della prop
      </Text>
    </View>
  );
};

// componente padre
const Universe = () => {
  return (
    <View>
      <Planet name="Earth" />
      <Planet name="Pluto" />
      <Planet name="Mars" />
    </View>
  );
};
export default Universe;

```

Quasi tutti i core components possono essere personalizzati mediante props. Le props consentono inoltre la comunicazione tra widget padre-figlio.

5.2.2.5 State

Con stato si intende la possibilità di dare memoria ai componenti; si tratta di una prop che permette di tenere traccia di informazioni che potrebbero cambiare nel tempo. Per associare uno `State` ad un componente si usa una speciale funzione `hook` chiamata `useState`.

Gli hook sono funzioni React disponibili durante il rendering e sono caratterizzate dal prefisso `use`. Formalmente possiamo pensare agli hook come a delle funzioni che danno accesso alla memoria interna di react: restituiscono il dato in questione e un metodo per modificare tale dato.

```

import { useState } from 'react';

const Counter = () => {
  // useState associa al componente una variabile di stato che mantiene
  // i dati tra i vari render ed una state setter function che consente
  // di aggiornare la state variable
  const [count, setCount] = useState(0);
  return (
    <View>
      <Text>Il conteggio è: {count}</Text>
      <Button
        onPress = {() => { setCount(count + 1) }}
        title = 'Aggiungi 1'
    
```

```

        />
    </View>
);
}
const App = () => {
    return <Counter />;
}

```

`useState(d)` accetta un valore di default e restituisce una coppia `(stateVar, stateSetFun)`, la prima rappresenta la variabile di stato, la seconda il suo metodo setter.

5.3 Navigation

React-Native non offre un modo nativo per implementare la navigazione, è necessario utilizzare un pacchetto della community chiamato `React Navigation`.

`Navigator` implementa la navigazione tra le schermate. Esistono diverse tipologie di navigazione:

1. `Stack` - a pila
2. `Drawer` - a cassetto con il classico hamburger menu
3. `Tab` - a schede

5.3.1 Stack Navigation

```

import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack';

const Stack = createNativeStackNavigator();

function App() {
    return (
        <NavigationContainer>
            <Stack.Navigator initialRouteName="About">
                <Stack.Screen name="Welcome"
                    component={WelcomeScreen} />
                <Stack.Screen name="About"
                    component={AboutScreen} />
            </Stack.Navigator>
        </NavigationContainer>
    )
}

function AboutScreen() {
    return (
        <View style={{ ... }}>
            <Text>Some info about the MobProg course ... </Text>
        </View>
    );
}

```

1. `createNativeStackNavigator` restituisce un oggetto con 2 proprietà: `Screen` e `Navigator`. Sono oggetti che vengono usati per configurare il navigatore... `Navigator` deve contenere gli elementi `Screen` come figli per definire le rotte... le rotte sono specificate direttamente come props. È possibile stabilire una rotta iniziale di default con `initialRouterName`

2. `NavigationContainer` : componente che gestisce il navigation tree e contiene il navigation state.
Solitamente si renderizza nell'entry point dell'applicazione;

Per muoversi tra gli screens (per esempio al click di un pulsante) è necessario passare la `navigation prop` al componente in questione ed utilizzare `navigation.navigate(route)` per navigare verso la rotta specificata.

```
function WelcomeScreen({ navigation }) {
  return (
    <View style={{ ... }}>
      <Text>Welcome!</Text>
      <Button
        title="Vai alla pagina About"
        onPress={() => navigation.navigate("About")}
      </Button>
    </View>
  );
}
```

Navigation implementa altri 2 metodi:

- `goBack()` , effettua il pop dello stack e quindi consente di tornare alla schermata precedente;
- `popToTop()` : consente di tornare all'inizio dello stack rimuovendo gli screen intermedi;

5.3.2 Passaggio di parametri alle rotte

I dati passati alle route vengono chiamati `params` e devono essere incapsulati in un oggetto (`p`) che viene passato come parametro al metodo `navigation.navigate(route, p)` . La prassi è passare un oggetto **JSON-serializzabile**. Per accedere ai parametri si usa `route.params` .

```
function AboutScreen({ route, navigation }) {
  // route.params è l'oggetto contenente i parametri
  const { foo, bar } = route.params;

  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text>Informazioni</Text>
      <Text>foo: { JSON.stringify(foo) }</Text>
      <Text>bar: { JSON.stringify(bar) }</Text>
      <Button
        title="Vai ancora ad About"
        onPress={() => {
          // qui non passo alcun valore per bar, quindi sarà undefined
          // e nel prossimo AboutScreen non apparirà alcunché
          navigation.push('About', {
            foo: 24,
          });
        }}
      />
      <Button title="Vai al Welcome" onPress={() => navigation.navigate('Welcome!')} />
      <Button title="Vai indietro" onPress={() => navigation.goBack()} />
    </View>
  );
}
```

Si possono specificare anche parametri iniziali con la prop `initialParams` :

```

function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Welcome">
        <Stack.Screen
          name="Welcome"
          component={WelcomeScreen}
          options={{ title: "Il mio Welcome Screen" }}
          initialParams={{ foo: 41, bar: "Valore iniziale di default" }}
        />
        <Stack.Screen name="About" component={AboutScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}

```

5.3.3 Navigator innestati

I navigator possono essere innestati...

- Navigator Stack che contiene 3 screen: Home, Profile e Settings
- Lo screen Home contiene un navigator Tab che mostra due screen: Feed e Messages

```

const Stack = createNativeStackNavigator();
const Tab = createBottomTabNavigator();
function Home() {
  return (
    <Tab.Navigator>
      <Tab.Screen name="Feed" component={Feed} />
      <Tab.Screen name="Messages" component={Messages} />
    </Tab.Navigator>
  );
}

function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen
          name="Home"
          component={Home}
          options={{ headerShown: false }}
        />
        <Stack.Screen name="Profile" component={Profile} />
        <Stack.Screen name="Settings" component={Settings} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}

```

Proprietà dei Navigator innestati:

- Ogni navigator mantiene la propria cronologia di navigazione,
- Ogni navigator ha le proprie options,
- Ogni navigator ha i propri params,

- Le azioni di navigazione vengono gestite dal navigator corrente, e passate al padre se questo non è possibile,
- Metodi specifici di un navigator sono disponibili anche nei navigator innestati

6 Flutter

Mobile-first SDK (completo) sviluppato da [Google](#) che include al suo interno:

- Engine di rendering ([Skia](#)),
- Componenti UI,
- Framework per testing,
- Routing,
- Hot reload

A differenza di `react-native` [non implementa nessun layer intermedio](#) (ciò garantisce un buon incremento delle prestazioni) e il framework implementa direttamente le interfacce di `UI` (widget). Inoltre, il motore di rendering '[Skia](#)' disegna pixel per pixel la UI sullo schermo del device, senza doversi interfacciare con il sistema operativo.

Rispetto alle altre soluzioni presenti sul mercato, `flutter` risulta essere particolarmente [performante](#) in quanto, non utilizzando alcun layer intermedio, compila direttamente l'applicativo in linguaggio macchina, il quale dialoga direttamente con il sistema operativo.

6.1 Introduzione a Dart

`Dart` è un linguaggio di programmazione creato da [Google](#) nel 2011 con l'obiettivo di sostituire `Javascript`, considerato [poco performante](#) e soggetto a problematiche di [sicurezza](#). Gli aspetti fondamentali del linguaggio sono:

1. Performance;
2. Curva di apprendimento;
3. Compilazione AoT (Ahead-of-time): trasformazione del codice sorgente in binario. Utilizzata per la distribuzione di applicazione binarie per `iOS` ed `Android`;
4. Compilazione JIT (Just-in-time): compilazione eseguita a runtime. L'hot-reload si basa su di essa;

In fase di sviluppo (debug), il codice `Dart` viene eseguito da una particolare virtual machine chiamata '[Dart VM](#)', fornita direttamente con l'SDK ufficiale del linguaggio (tale SDK è fornito in bundle con `flutter`).

A livello [tecnico](#) le caratteristiche peculiari:

- Linguaggio platform independent;
- Linguaggio compilato;
- Supporta programmazione concorrente
- [Object Oriented](#)
- Fortemente [tipizzato](#) ma con supporto alla tipizzazione dinamica e alla null safety

6.1.1 Variabili

La definizione del tipo è opzionale ma fortemente consigliata per mantenere l'approccio type-safe:

```
keyword <tipo_dato> nome;
```

L'operatore `?` consente di definire [variabili nullable](#):

```
String? a = null;
```

L'operatore `_` consente di rendere **privato** un elemento:

```
int _counter = 0;
```

6.1.2 Debug e Assert

Dart supporta la modalità di debug e l'utilizzo dei breakpoint. In tale fase tornano utili le asserzioni, le quali consentono di intercettare una condizione non soddisfatta che interromperebbe il flusso con un errore:

```
void main() {
    var num = 1;
    assert(num != 1);
}
```

6.1.3 Funzioni e parametri

```
tipo Ritorno nome_funzione([lista_parametri]){
    corpo
}
```

Si possono definire anche parametri opzionali:

- tramite `[]`: `String helloworld(String str1, [String str2])`
- tramite `{}` per gestire i parametri mediante i relativi **nomi**: `String helloworld(String str1, {String str2})`. Ciò consente anche di non rispettare l'ordine di definizione

La keyword `required` consente di specificare dei parametri mandatori. Dal momento che in Dart tutto è un oggetto, è anche possibile usare funzioni come parametri di altre funzioni:

☰ Example

```
void main(){
    exec( (){
        return "Hello World";
    });
}
void exec(Function f){
    print(f());
}
```

6.1.3.1 Funzioni lambda

Sono funzioni che non hanno un nome.

☰ Example

```
void main(){
    exec(helloworld);
}
```

```

void exec(Function f){
    print(f());
}

String helloworld(){
    return "Hello World";
}

```

6.1.3.2 Arrow Syntax

Sintassi compatta per la definizione di una funzione:

```
tipo_ritorno nome_funzione() => corpo_funzione
```

☰ Example

```
int sum(int a, int b) => a+b;
```

6.1.4 Ereditarietà e mixin

Dart di default supporta il principio dell'ereditarietà: una classe può estendere una superclasse mediante la keyword `extends`. Il linguaggio tuttavia **non supporta l'ereditarietà multipla**, la quale può comunque essere ottenuta mediante `mixin`.

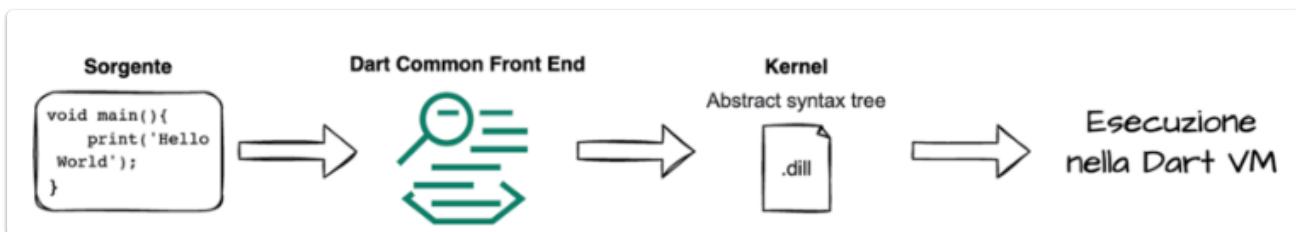
6.1.5 Esecuzione

Dart mette a disposizione 2 modalità di esecuzione del codice: *from-source* e tramite *snapshot*. Utilizzando flutter tuttavia, la compilazione (eventualmente anche in release) sarà gestita completamente dal framework.

6.1.5.1 Esecuzione from-source e compilazione JIT - Just-In-Time

La prima fa uso della **compilazione JIT**: la VM esegue il codice compilandolo *just-in-time*. Utilizzando la compilazione JIT, la VM non compilerà tutto il codice dell'applicativo alla sua esecuzione, ma solamente **metodi e funzioni che dovranno essere 'appena' eseguiti**.

Tale tipologia di compilazione si pone a metà strada tra un **approccio interpretato** ed una **compilazione statica** in quanto, nel primo caso il software viene eseguito completamente a runtime, senza eseguire alcuna compilazione; nel secondo caso invece il software viene prima **compilato e successivamente eseguito**.



Tale compilazione fa uso di un **layer intermedio** chiamato 'kernel', creato in fase di compilazione durante l'analisi del codice sorgente. In particolar modo il componente dell'SDK chiamato "*Dart Common Front End*" (**CFE**) effettua il **parsing** del sorgente e crea il file ***kernel AST*** (file `.dll`), il quale può essere eseguito dalla VM.

Un qualunque sorgente Dart può essere eseguito da terminale mediante: `dart run nome_file`. Alternativamente è possibile compilare manualmente e successivamente eseguire il file compilato `.dll`:

```
dart compile kernel nome_file.dart  
dart run nome_file.dll
```

6.1.5.2 Snapshot

Nel caso dell'esecuzione tramite **snapshot**, il compilatore effettua una **copia dell'heap** (la memoria della *DartVM*), con un elenco di oggetti ed altre informazioni utili ad un avvio più veloce. Con questa tipologia di esecuzione infatti, il codice non viene analizzato dal *CFE*.

6.1.5.2.1 Snapshot JIT

In questo frangente l'applicativo può essere avviato partendo direttamente da un **binario**, come se fosse un oggetto serializzato. Lo snapshot JIT infatti contiene tutte le classi e funzioni già compilate... lo snapshot è tuttavia dipendente dall'architettura!

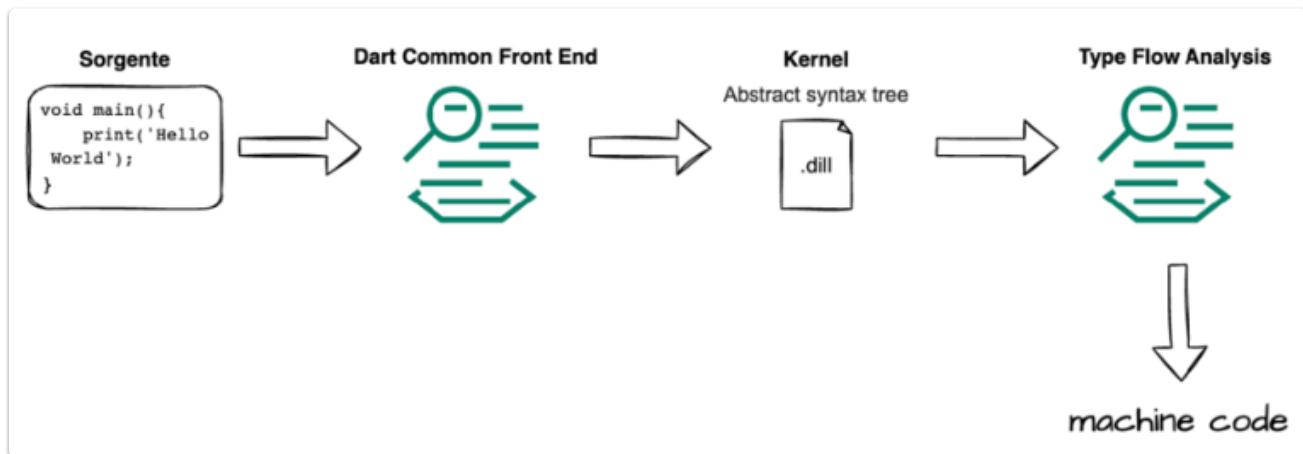
Uno snapshot può essere creato da terminale mediante:

```
dart compile jit-snapshot nome_file.dart  
dart run nome_file.jit
```

6.1.5.3 Compilazione AoT - Ahead-of-time

Con la compilazione AoT il codice viene **compilato del tutto in nativo**, dopo l'analisi TFA - *Type Flow Analysis*. Si tratta di un'analisi statica del codice *kernel AST* col fine di eliminare le porzioni di codice non utilizzate ed implementando importanti **ottimizzazioni**.

Il binario AoT sarà pertanto molto **più leggero e performante** rispetto a quello JIT e pertanto è la modalità da preferire in caso di distribuzione di un'eventuale release.



6.1.5.3.1 Snapshot AoT

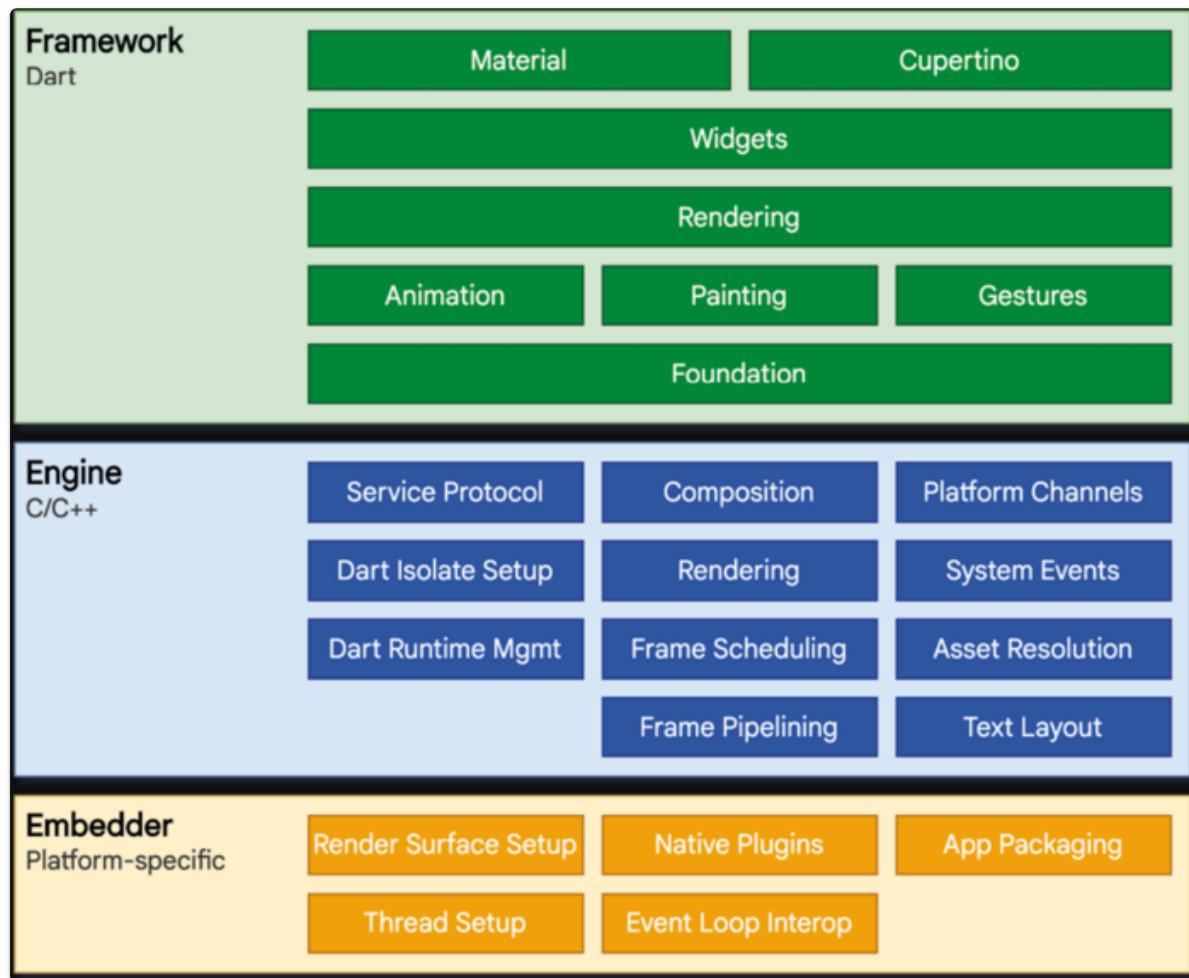
È possibile creare degli snapshot anche in modalità AoT. La differenza peculiare risiede nel fatto che tale snapshot **non contiene nessun runtime** e pertanto è necessario utilizzare un tool esterno per avviare il software. Per tale ragione sono anche molto più leggere delle istantanee JIT.

```
dart compile aot-snapshot nome_file.dart
```

6.2 Architettura Flutter

L'architettura `Flutter` si compone di 3 strutture:

- Framework;
- Engine;
- Embedder



6.2.1 Framework

Rappresenta l'interfaccia con l'utente e contiene tutti gli strumenti per lo sviluppo di applicazioni in `Flutter`.

- "*Material*" e "*Cupertino*" sono librerie che consentono di implementare i relativi design;
- "*Widgets*" rappresenta tutti gli elementi utilizzabili per la creazione di interfacce grafiche;
- "*Rendering*: si occupa della trasformazione dei widget in pixel (rendering);
- "*Foundation*" è la libreria Core che fornisce le primitive

6.2.2 Engine

Strato intermedio scritto in `C++`, costituisce il motore di `flutter` in quanto lavora a runtime e fornisce tutte le API di basso livello per l'interfaccia con la rete, I/O, animazioni ecc....

6.2.3 Embedder

Punto di contatto tra `Flutter` e il SO. È un **layer specifico per piattaforma**, scritto in:

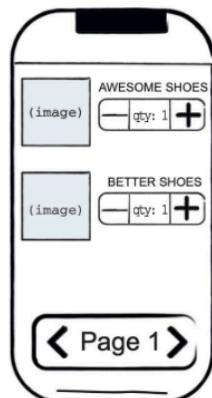
- `java` e `c++` per android;
- `objective-c/c++` per ios;
- `c++` per Linux e Windows

Ha il compito di **dialogare con le API fornite dal sistema operativo** per il calcolo delle superfici di rendering, la gestione degli eventi (Event Loop), dei thread... e di tutte l'interazione reale dell'applicativo con il device.

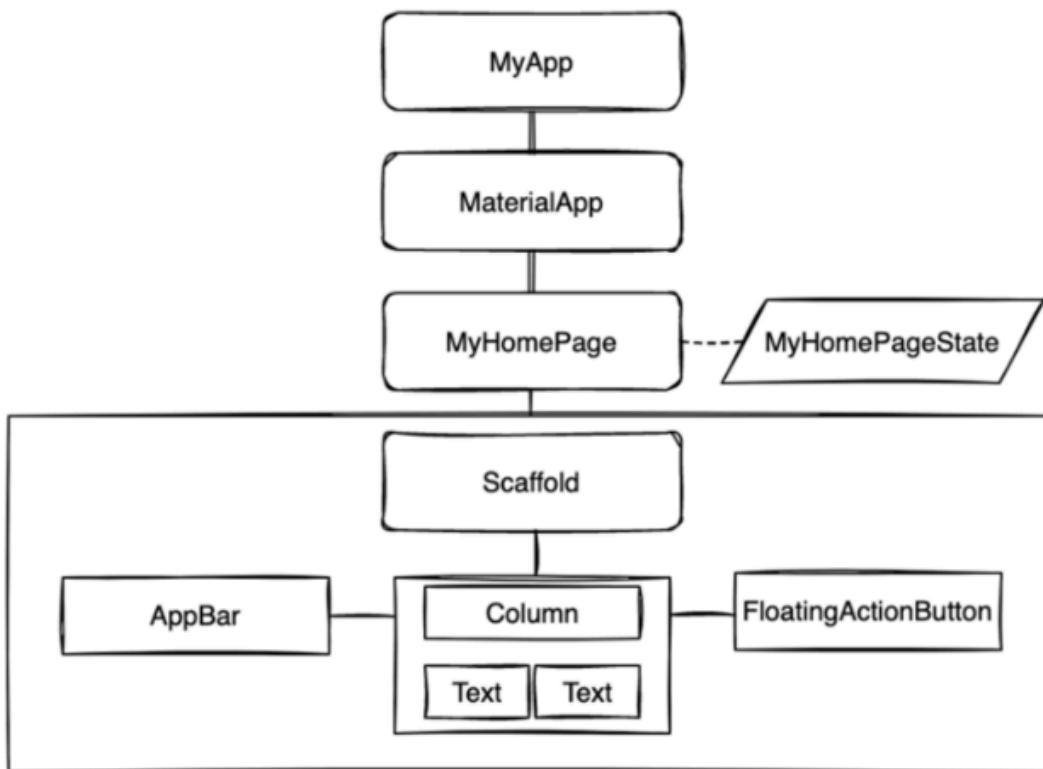
6.3 Widget

La creazione di un'app flutter consiste nella creazione di una mobile user interface mediante componenti chiamati **widget**: ogni pezzo dell'app è un widget.

- Esempio app carrello della spesa
- L'app è standard
 - Mostra una lista di prodotti,
 - Un prodotto può essere aggiunto o rimosso
- Tutto ciò che appare in questa app è un **widget**.
 - Oltre ai widget, le uniche classi che scriveremo in Flutter serviranno a specificare la logica dell'applicazione
- I widget possono avere uno **stato**
 - Quando lo stato di un widget cambia, un **feedback** permette al framework di capire cosa cambiare nel rendering dell'applicazione

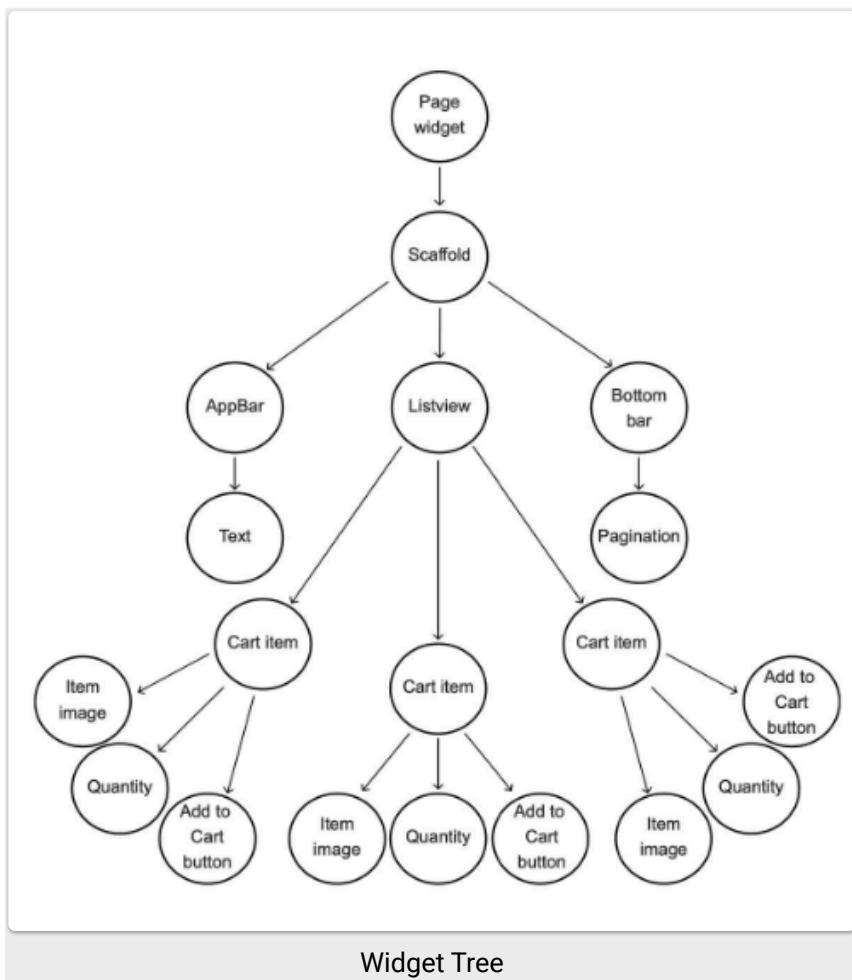


```
build(BuildContext context) {  
  return Column(  
    //...  
    Image(),  
    Text("BETTER SHOES"),  
    //...  
    IconButton(  
      icon: Icon(Icons.chevron_left),  
    ),  
    Text("Page $page_num"),  
    //...  
  ); // column  
}
```



Struttura di un'app generata dal framework

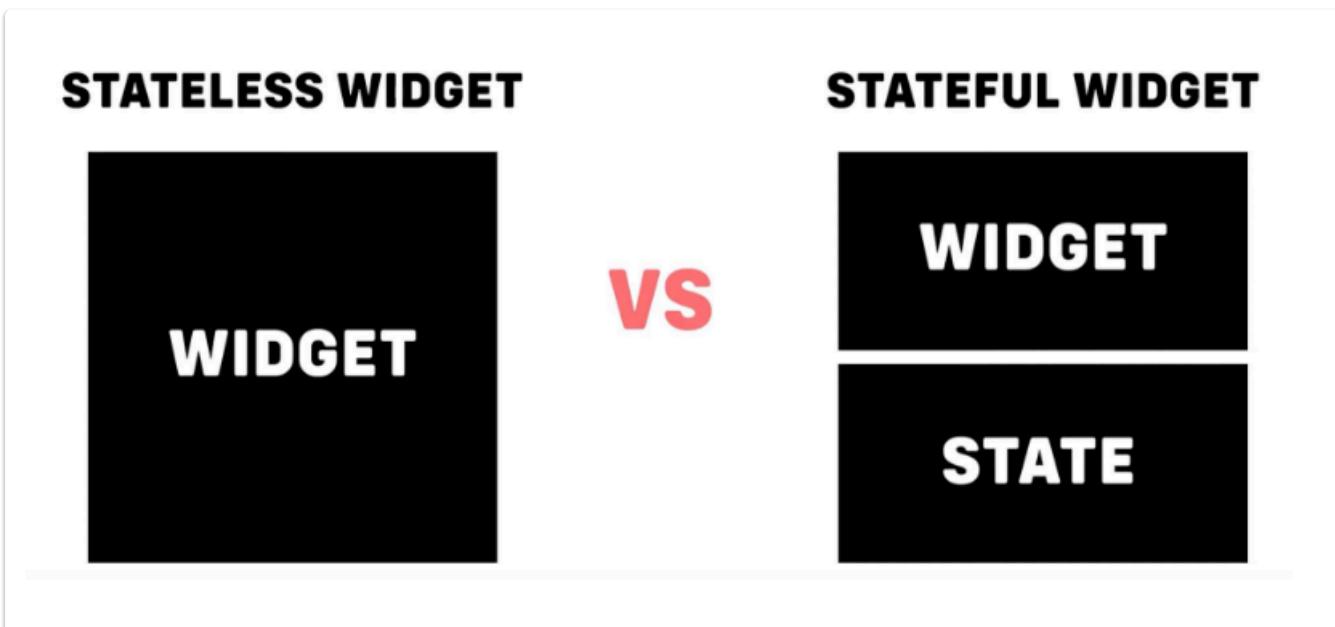
In **Flutter** qualunque cosa è un **widget**... l'intera applicazione può essere rappresentata mediante un **albero** (**widget tree**) composto da widget annidati ed un entry-point (`MyApp`).



Widget Tree

6.3.1 Widget const

I widget definiti `const` non verranno **ricostruiti** in quanto sono definiti costanti e quindi, per definizione, **immutevoli**.



6.3.2 Stateless Widget

Si tratta di **widget immutabili** che non conservano uno stato... dedicati principalmente a mostrare contenuto ed UI.

Riceve uno o più parametri dal widget genitore che memorizza in campi `final`. Quando viene invocato il

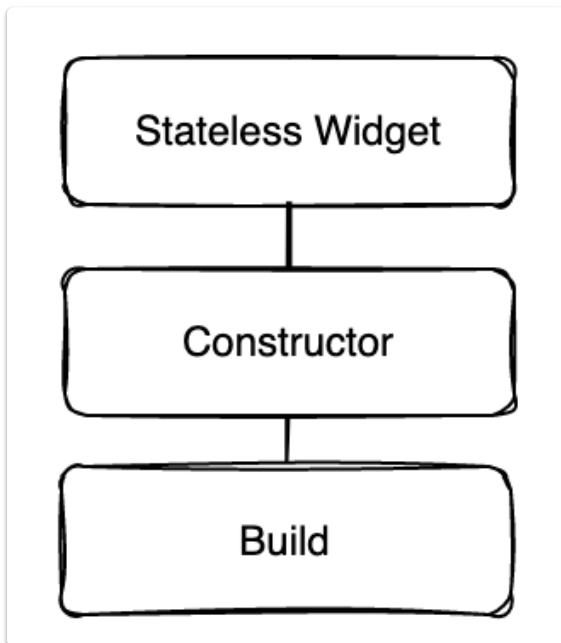
`build()`, utilizza queste informazioni per derivare i parametri da passare ai figli.
Possono essere eliminati senza alcuna perdita di informazione.

6.3.2.1 Lifecycle stateless widget

Con life-cycle di un widget si intende il suo processo di **building** ed **updating**.

Il **ciclo di vita** di un widget stateless è molto semplice dal momento che non è presente lo stato associato. Il ciclo di vita corrisponde alle 3 fasi:

1. Viene **creata la classe** del widget;
2. Viene istanziato il widget mediante il **costruttore**;
3. Il widget viene costruito mediante il metodo `build()` ;



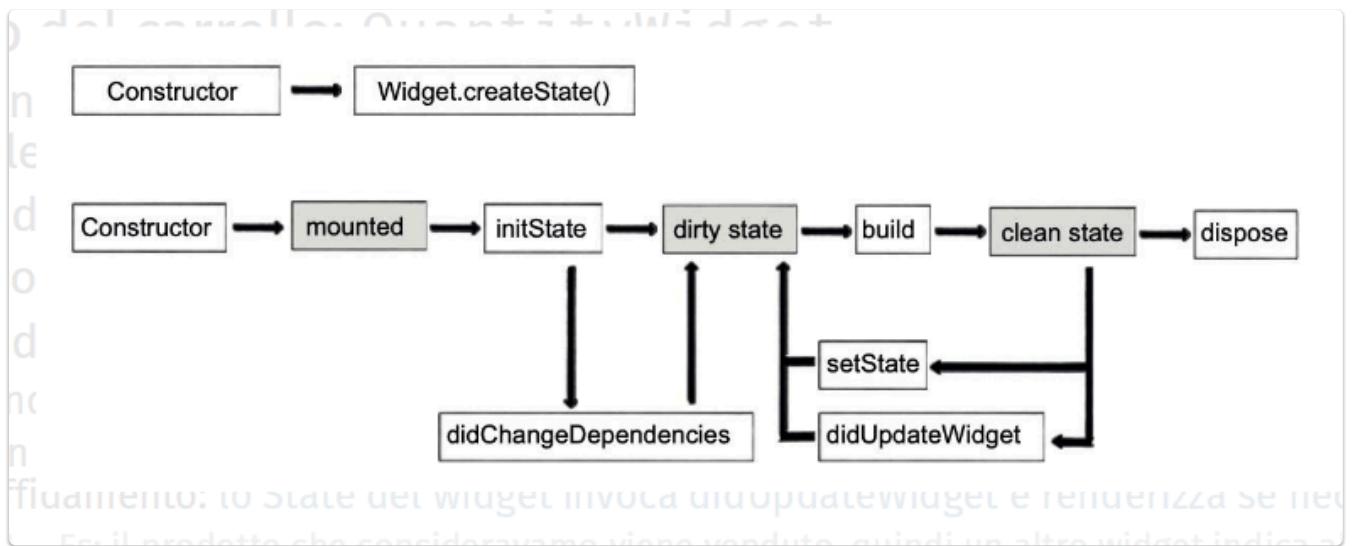
6.3.3 StatefulWidget

Widget dinamici che consentono di gestire i **cambiamenti** dell'interfaccia che possono avvenire, per esempio, tramite l'interazione dell'utente con la UI o un evento asincrono. Il widget rimane sempre **immutable**, quello che cambia è lo **stato** associato.

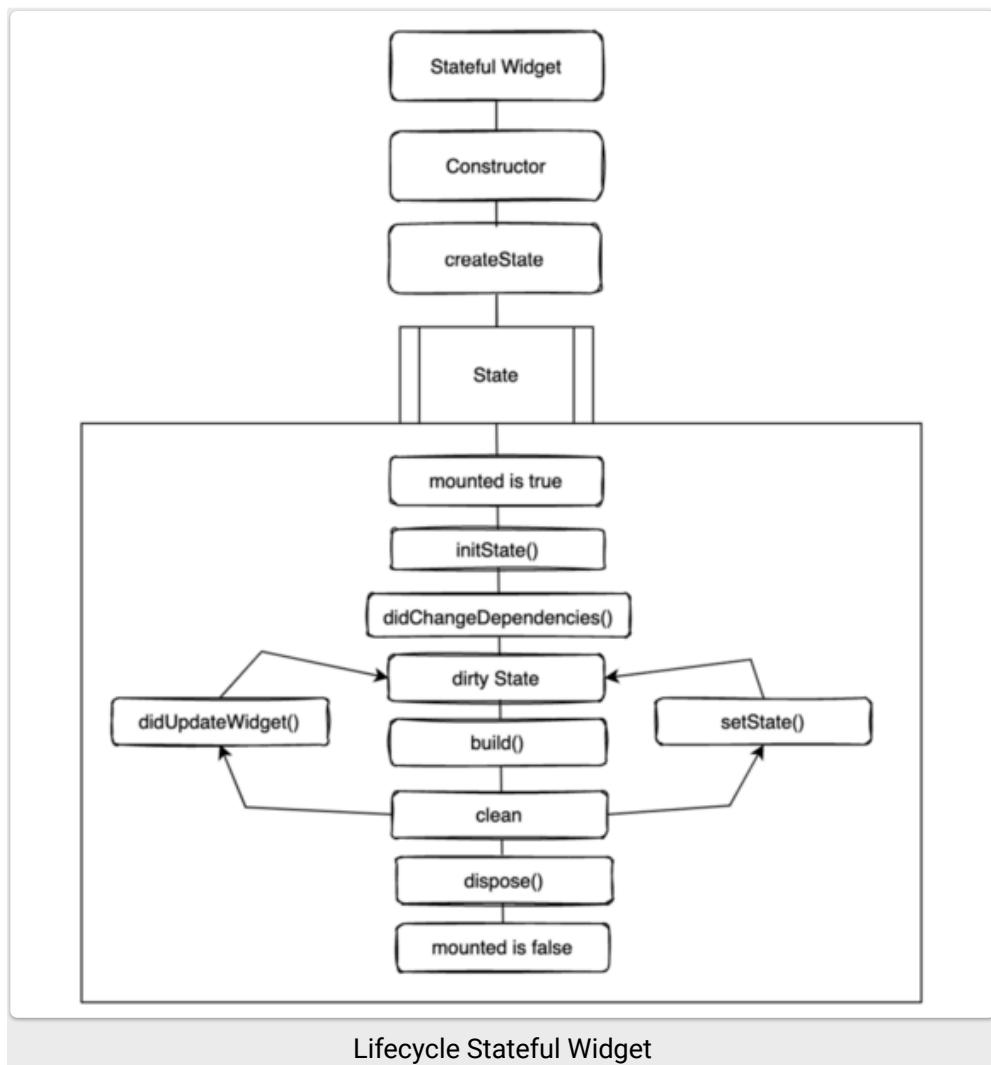
Si usa il metodo `createState()` per uno stato mutabile, associato al widget. Alla creazione di un nuovo stato occorre associare una classe del tipo `_MyHomePageState`, che si occuperà della gestione dello stato. Tale classe avrà il proprio metodo `build` e il proprio `BuildContext`, oltre al metodo `setState()` che ha il compito di notificare al framework una notifica in corso d'opera. Flutter si occuperà quindi di reindirizzare nuovamente il widget per mostrare le modifiche sullo schermo.

6.3.3.1 Lifecycle StatefulWidget

Negli StatefulWidget lo stato ha un suo **ciclo di vita** ben definito. Il ciclo di vita è gestito dall'**interazione** tra l'istanza dello `StatefulWidget` e l'oggetto `State`.



- Il widget è creato ed istanziato mediante il suo **costruttore**;
- Viene invocato il metodo `createState()` per la **creazione dello stato**. L'oggetto `State` è stato creato;
- Non appena l'oggetto `State` viene associato ad un `BuildContext` e quindi inserito nel widget tree, il suo flag booleano `mounted` viene impostato su `true`. Se la property è settata su `true`, allora il widget è **presente nel widget tree**;
- Viene invocato il metodo `initState()`. Il metodo sarà invocato solamente una volta dopo la creazione dello stato.
- `didChangeDependencies` verrà eseguito quando **cambia una dipendenza** del widget, ad esempio un widget genitore oppure una `MediaQuery`, il `Theme` ...
- `dirty State` è uno stato interno di Flutter il quale indica che l'oggetto `State` è "sporco", ovvero ha **subito delle modifiche** che devono essere riportate sull'interfaccia grafica. Il framework infatti non ha ancora aggiornato il processo di rendering (lo farà all'invocazione del metodo `build()`);
- Quando il `dirty state` viene aggiornato, si invoca il **processo di build**. Il framework **ricostruisce il widget** basandosi sullo stato corrente dell'oggetto `State`
- `clean` è uno stato interno che indica la corretta esecuzione del metodo `build()`;
- Viene invocato il metodo `dispose()` che **termina il ciclo di vita del widget** rimuovendo definitivamente widget e stato del widget tree e liberandone le risorse



Lifecycle Stateful Widget

Ciclo di vita di un widget



- Il processo di *building* e *updating* di un widget è detto il suo *life cycle*
- Esempio del carrello: *QuantityWidget*
 1. Quando navighiamo sulla pagina contenente il widget, Flutter crea l'oggetto, il quale crea lo State associato al widget;
 2. Il widget viene inserito nel tree view e il metodo initState viene invocato.
 3. Dopo l'inizializzazione dello stato, Flutter costruisce il widget (e lo renderizza)
 4. Il widget ora resta in attesa di tre possibili eventi
 1. Andiamo su un'altra pagina: **lo stato può essere eliminato**;
 2. Un altro widget ha aggiornato o modificato una configurazione su cui il nostro widget fa affidamento: **lo State del widget invoca didUpdateWidget e renderizza se necessario**
 1. Es: il prodotto che consideravamo viene venduto, quindi un altro widget indica al QuantityWidget di disabilitarsi poiché non può aggiungere quel prodotto.
 3. Il widget viene premuto: **invoca setState e aggiorna lo stato interno del widget. Implica un rebuild e un rendering da parte di Flutter.**

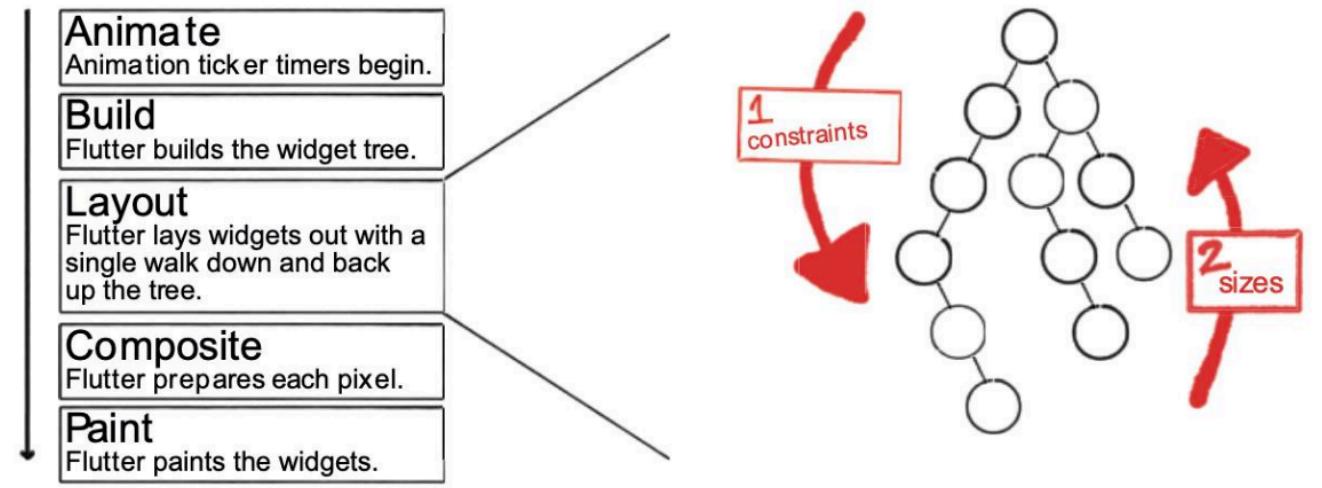
6.3.4 Widget Rendering/Updating

- Update;
- Rendering;

6.3.4.1 Processo di update

- **Il processo di update ad alto livello**
 1. Avviene una interazione con i widget
 - Es: un utente tocca un bottone
 2. L'app invoca `setState` sul callback
 - Es: `Button.onPressed`
 3. Lo state del Button è ora *dirty*
 - Esso indica a Flutter di doverlo ricostruire
 - Una nuova versione del widget verrà costruita
 4. Il nuovo widget rimpiazza il vecchio nel widget tree
 5. Flutter renderizza il widget tree aggiornato

6.3.4.2 Processo di rendering



1. Parte l'animation ticker

- Un sottoprocesso che controlla in quale lasso di tempo i widget devono essere aggiornati (es, se si muovono)

2. Il widget tree viene costruito,

3. L'organizzazione dei widget viene definita (layout),

- Questo passaggio è fatto in **tempo lineare** rispetto al n. di widget
- Il passaggio verso il basso definisce i *vincoli* (il genitore indica al figlio vincoli sulla posizione, dimensione, etc)
- Il passaggio verso l'alto definisce i valori effettivi (garantiti rispettare i vincoli)

4. Ogni singolo pixel viene preparato (rastering),

5. I widget vengono mostrati a schermo.

6.3.4.3 Notifica dei cambiamenti negli stati

La notifica del cambiamento di uno stato può attraversare il widget tree verso l'**alto**, trasportata mediante **callback**. Lo stato corrente invece viene trasmesso verso il basso agli **Stateless Widget**. La notifica del cambiamento di stato viene effettuata mediante il metodo `setState()` (il quale marca il widget come **dirty**, flutter dovrà quindi ricostruirlo) tuttavia, per applicazioni più complesse si possono percorrere strade differenti:

6.3.4.3.1 Callback

Tramite callback è possibile propagare lo stato di un widget genitore, verso i suoi figli, sfruttando gli argomenti del costruttore.

```

// Il Widget Genitore (gestisce lo stato)
class ParentWidget extends StatefulWidget {
  const ParentWidget({super.key});

  @override
  State<ParentWidget> createState() => _ParentWidgetState();
}

class _ParentWidgetState extends State<ParentWidget> {
  int _counter = 0; // Lo stato del genitore

  // Questo è il metodo callback che il figlio chiamerà
  void _incrementCounter() {
    setState(() {
      _counter++; // Aggiorna lo stato del genitore
    });
  }

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            'Contatore: ${_counter}', // Il genitore mostra il suo stato
            style: const TextStyle(fontSize: 24),
          ),
          const SizedBox(height: 20),
          // Passiamo la funzione _incrementCounter al widget figlio
          ChildButton(onPressed: _incrementCounter),
        ],
      ),
    );
  }
}

// Il Widget Figlio (esegue un'azione che influisce sullo stato del genitore)
class ChildButton extends StatelessWidget {
  // Dichiariamo un campo finale di tipo VoidCallback (una funzione senza argomenti e
  // senza ritorno)
  final VoidCallback onPressed;

  const ChildButton({super.key, required this.onPressed});

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      // Quando il bottone viene premuto, chiamiamo la funzione che ci è stata passata
      // dal genitore
      onPressed: onPressed,
      child: const Text('Incrementa Contatore'),
    );
  }
}

```

6.3.5 BuildContext

Il metodo `build()` ha il compito di creare un nuovo `nodo` all'interno del widget-tree, rappresentante il widget che si sta creando. Prende come argomento un oggetto `BuildContext` che inietta nella funzione `builder` il `"context"` che rappresenta il collegamento tra il widget effettivo e il nodo creato nel widget-tree. Il `"context"` è infatti l'oggetto che consente di interagire con l'albero dei widget.

:≡ Example

Un esempio è quello di un widget che accede, tramite `"context"` ad un'informazione definita nel widget `MaterialApp`.

```
main() {
    runApp(const MyApp());
}

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

    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            theme: ThemeData(primaryColor: Colors.lightBlue[800]),
            home: Scaffold(
                body: Center(
                    child: Text(
                        "Hello World",
                        textDirection: TextDirection.ltr,
                        style: TextStyle(color: Theme.of(context).primaryColor)
                    )
                )
            );
    }
}
```

Il `BuildContext` può essere visto come ad un riferimento alla posizione di un widget nell'albero dei widget. Ogni volta che si invoca il metodo `build()` infatti viene passato un `BuildContext`.

6.3.5.1 Liste

- `ListView`: widget per **liste verticali ed orizzontali**, utilizzato generalmente quando ogni elemento è formato da icona e testo (`ListTile`). Il parametro `scrollDirection` permette di stabilire la **direzione degli elementi**;
- `GridView`: widget per la creazione di una griglia di elementi disposti a matrice. Presenta diversi costruttori, il più usato è `GridView.count()` che consente di specificare il numero di righe o colonne;

Il metodo `List.generate(n, f)` consente di generare `n` widget tramite `f`.

6.3.5.1.1 Liste eterogenee

1. Creare un **data source** con oggetti di tipo **differenti** mediante una classe astratta `ListItem`
2. **Convertire** la data source in una **lista di widget**,

```

abstract class ListItem {
    Widget buildTopPart(BuildContext context);
    Widget buildBottomPart(BuildContext context);
}

class DateItem implements ListItem {
    final String date;

    DateItem(this.date);

    @override
    Widget buildTopPart(BuildContext context) {
        return Text(
            date,
            style: Theme.of(context).textTheme.headlineSmall,
        );
    }

    @override
    Widget buildBottomPart(BuildContext context) => const SizedBox.shrink();
}

```

```

class MessageItem implements ListItem {
    final String user;
    final String body;

    MessageItem(this.user, this.body);

    @override
    Widget buildTopPart(BuildContext context) => Text(user);

    @override
    Widget buildBottomPart(BuildContext context) => Text(body);
}

```

```

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

    const MyApp({super.key, required this.items});

    @override
    Widget build(BuildContext context) {
        const title = 'Lista Eterogenea';

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

                        return ListTile(
                            title: item.buildTopPart(context),
                            subtitle: item.buildBottomPart(context),
                        );
                    },
                ),
            );
        }
    }
}

```

```

void main() {
    runApp(
        MyApp(
            items: List<ListItem>.generate(
                1000,
                (i) => i % 6 == 0
                    ? DateItem('Data: $i')
                    : MessageItem('User: $i', 'Testo del messaggio: $i'),
            ),
        ),
    );
}

```



6.3.5.1.2 Aggiornamento dinamico di liste

Significa **manipolare** il relativo **data source** mediante una funzione che invoca `setState`. Il metodo `build` si occupa di convertire il data source in una lista di widget. Il metodo `ListView.builder()` renderizza i widget della lista ricorsivamente solo quando sono **visualizzati** a schermo.

```

import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            title: 'Aggiungi alla lista',
            home: MyHomePage(),
        );
    }
}

```

```

class MyHomePage extends StatelessWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  final List<String> _items = ['Item 1'];

  void _addItem() {
    setState(() {
      int nextNum = _items.length + 1;
      _items.add('Item $nextNum');
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('La mia lista'),
      ),
      body: ListView.builder(
        itemCount: _items.length,
        itemBuilder: (context, index) {
          return ListTile(
            title: Text(_items[index]),
          );
        },
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _addItem,
        tooltip: 'Aggiungi',
        child: Icon(Icons.add),
      ),
    );
  }
}

```

6.4 Navigation

Flutter fornisce un [sistema completo per la navigazione](#). Vi sono 2 possibilità:

- Istanza della classe `Navigator`;
- Istanza della classe `Router`, per app più complesse.

6.4.1 Navigator

`Navigator` consente di muoversi tra le schermate seguendo un approccio "navigation history stacks", organizzando i vari screen in una [pila](#) (memorizza uno stack di oggetti `Route`). Per navigare su una nuova schermata basta fare il push nello stack:

```

Navigator.of(context).push(MaterialPageRoute(
  builder: (context) => const NewPage())
);

```

MaterialPageRoute è una sottoclasse di `Route` che specifica l'**animazione** di transizione dallo schermo corrente a quello pushato.

Analogamente, per tornare ad una pagina precedente occorre effettuare un `pop`.

6.4.2 Tab Navigation

Flutter mette a disposizione il widget `TabController`.

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: DefaultTabController(
        length: 3,
        child: Scaffold(
          appBar: AppBar(
            bottom: const TabBar(
              tabs: [
                Tab(icon: Icon(Icons.directions_car)),
                Tab(icon: Icon(Icons.directions_transit)),
                Tab(icon: Icon(Icons.directions_bike)),
              ],
            ),
            title: const Text('Tabs Demo'),
          ),
          body: const TabBarView(
            children: [
              Icon(Icons.directions_car),
              Icon(Icons.directions_transit),
              Icon(Icons.directions_bike),
            ],
          ),
        ),
      );
  }
}
```

6.4.3 Passare dati ad un Route

Naturalmente è possibile passare **dati** da una `Route` all'altra, sfruttando i parametri del costruttore del widget. La `Route` infatti è un semplice widget...

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

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

void main() {
  runApp(
    MaterialApp(
      title: 'Passing Data',
      home: TodosScreen(
        todos: [
          Todo(title: 'Buy milk', description: 'Milk'),
          Todo(title: 'Buy bread', description: 'Bread'),
          Todo(title: 'Buy eggs', description: 'Eggs'),
        ],
      ),
    ),
  );
}
```

```

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

```

```

class TodosScreen extends StatelessWidget {
    const TodosScreen({super.key, required this.todos});

    final List<Todo> todos;

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: const Text('Todos'),
            ),
            body: ListView.builder(
                itemCount: todos.length,
                itemBuilder: (context, index) {
                    return ListTile(
                        title: Text(todos[index].title),
                        // Quando un utente tocca la ListTile, si naviga verso
                        // il DetailScreen; nota bene: qui instanziamo un DetailScreen
                        // e al contempo passiamo il todo corrente ad esso
                        onTap: () {
                            Navigator.push(
                                context,
                                MaterialPageRoute(
                                    builder: (context) => DetailScreen(todo: todos[index]),
                                ),
                            );
                        },
                    );
                },
            );
        );
    }
}

```

```

class DetailScreen extends StatelessWidget {
    // Il todo è necessario
    const DetailScreen({super.key, required this.todo});

    // Memorizziamo qui il todo
    final Todo todo;

    @override
    Widget build(BuildContext context) {
        // Visualizziamo le informazioni del todo
        return Scaffold(

```

```

        appBar: AppBar(
            title: Text(todo.title),
        ),
        body: Padding(
            padding: const EdgeInsets.all(16),
            child: Text(todo.description),
        ),
    );
}
}

```

6.4.4 Restituire dati da una Route

È possibile anche restituire dati da una `Route` sfruttando il metodo `pop` di `Navigator` e specificando i parametri di ritorno all'interno del metodo stesso:

```

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

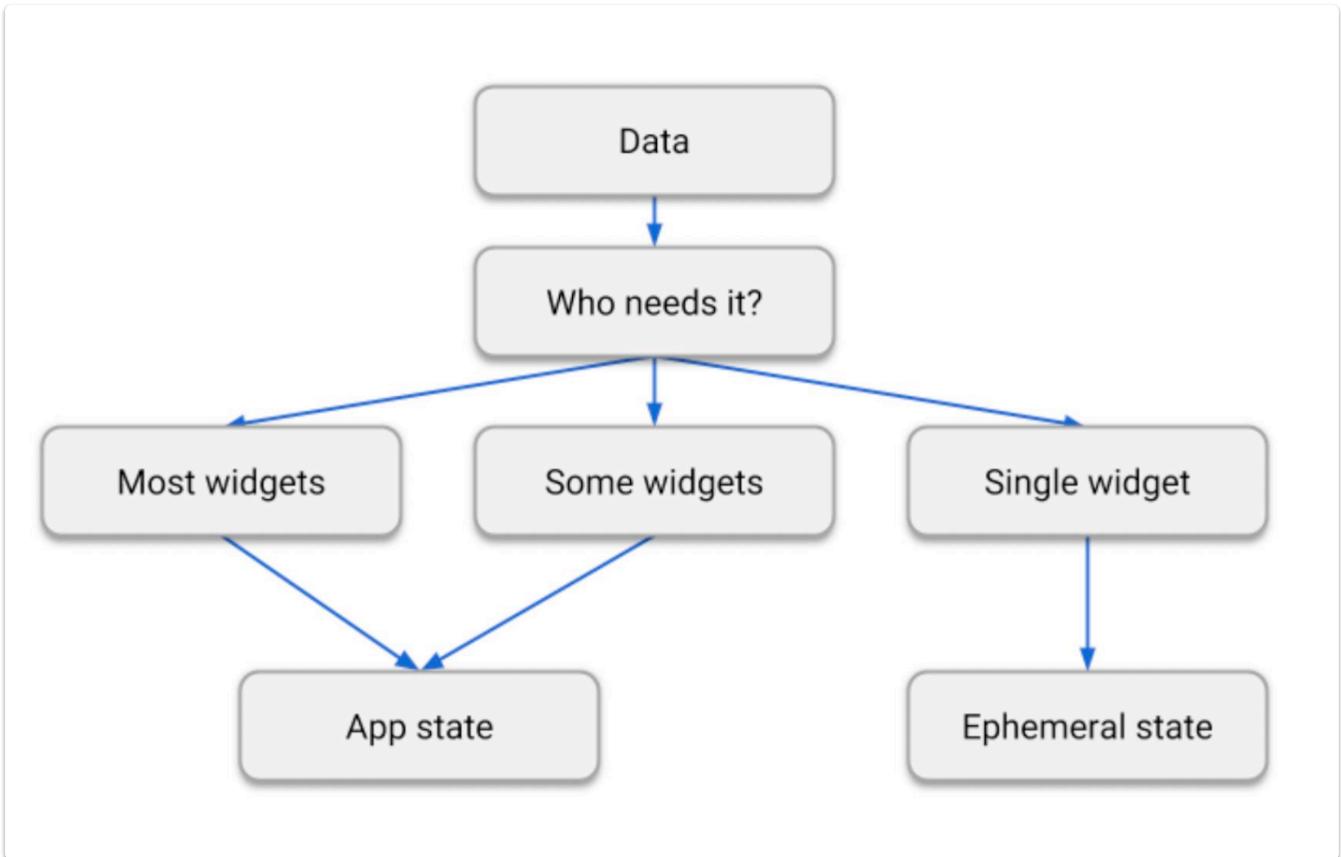
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: const Text('Scegli un\'opzione'),
            ),
            body: Center(
                child: Column(
                    mainAxisAlignment: MainAxisAlignment.center,
                    children: <Widget>[
                        Padding(
                            padding: const EdgeInsets.all(8),
                            child: ElevatedButton(
                                onPressed: () {
                                    // Chiude lo schermo e restituisce Yep!
                                    Navigator.pop(context, 'Yep!');
                                },
                                child: const Text('Yep!'),
                            ),
                        ),
                        Padding(
                            padding: const EdgeInsets.all(8),
                            child: ElevatedButton(
                                onPressed: () {
                                    // Chiude lo schermo e restituisce Nope.
                                    Navigator.pop(context, 'Nope.');
                                },
                                child: const Text('Nope.'),
                            ),
                        ),
                    ],
                ),
            );
    }
}

```

6.5 Stati

Flutter supporta 2 tipologie diverse di stati:

- **Ephemeral State:** stato del **singolo widget**, utilizzato dagli `StatefulWidget`;
- **App State:** stato **condiviso** all'interno di **tutta l'applicazione**. Si può usare per le informazioni di login, carrello della spesa, impostazioni dell'app...



Se uno stato è usato da più widget, conviene utilizzare `App State` ... che deve essere inserito come nodo del widget tree, sopra ai widget che lo utilizzano. Alla costruzione di un widget, recuperiamo l'App state corrente, e lo usiamo per costruire il widget stesso.

6.5.1 Definizione dell'App State

L'app state va inserito, dal punto di vista del widget tree, **sopra** ai widget che lo utilizzano (buona norma definirlo nell'entry-point). Alla costruzione di un widget, si **recupera l'app state** corrente e lo si usa per costruire lo stesso widget.

Per notificare le modifiche di un App state avvenute nei widget figli usiamo un `Provider` (si tratta di uno state manager), il quale si occupa di fornire ai widget un modo di **accedere** e **modificare** l'App state. Un provider necessita generalmente di:

- `ChangeNotifier`: classe che fornisce notifiche di cambiamento a chi si abbona ad essa (i listener). Definisce l'App state;
- `ChangeNotifierProvider` widget che fornisce un'istanza di `ChangeNotifier` ai figli;
- `Consumer`: widget che accetta un `ChangeNotifier` e permette di accedere all'App State;

```
// lib/counter_model.dart

import 'package:flutter/foundation.dart';

class CounterModel extends ChangeNotifier {
  int _count = 0;

  int get count => _count;
```

```
void increment() {
    _count++;
    notifyListeners();
}

import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'counter_model.dart';

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

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

    @override
    Widget build(BuildContext context) {
        return ChangeNotifierProvider(
            create: (context) => CounterModel(),
            child: const MaterialApp(
                title: 'Flutter Demo',
                home: HomePage(),
            ), // MaterialApp
        ); // ChangeNotifierProvider
    }
}

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

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: const Text('Provider Example App'),
            ), // AppBar
            body: Center(
                child: Consumer<CounterModel>(
                    builder: (context, counter, child) {
                        return Text(
                            'Counter: ${counter.count}',
                            style: Theme.of(context).textTheme.headlineMedium,
                        ); // Text
                    },
                ), // Consumer
            ), // Center
            floatingActionButton: FloatingActionButton(
                onPressed: () =>
                    Provider.of<CounterModel>(context, listen: false).increment(),
                tooltip: 'Increment',
                child: const Icon(Icons.add),
            ), // FloatingActionButton
        ); // Scaffold
    }
}
```

6.5.2 Watch

La classe `Consumer` presenta diverse alternative di implementazione... una di queste è l'utilizzo dei metodi `read`, `watch` e `select` presenti direttamente nel `context`:

- `Read`: consente l'accesso al modello dati (`ChangeNotifier`) senza scatenare processi di build del widget al cambio del modello;
- `Watch`: consente di monitorare cambiamenti di stato nel modello e di procedere al build del widget;
- `Select`: consente di usare la logica di `watch` specificamente per un elemento del modello

6.6 Persistenza dei dati

Flutter supporta diverse metodologie per garantire la persistenza dei dati:

- Shared Preferences
- Lettura/Scrittura su file
- Database

6.6.1 Shared Preferences

Modalità più **semplice** per data storage... consente di salvare i dati in modalità **chiave/valore**. Da utilizzare per **piccole collezioni** di dati. Si possono utilizzare solamente **tipi primitivi** e non è garantito che i dati persistano tra i vari riavvii dell'app.

Si utilizza un'istanza condivisa di `SharedPreferences` che fornisce diversi metodi getter e setter.

```
// Carica le SharedPreferences per l'app
final prefs = await SharedPreferences.getInstance();

// Salva il valore del contatore nella memoria persistente con la chiave 'counter'
await prefs.setInt('counter', counter);
```

```
// Ottieni le SharedPreferences
final prefs = await SharedPreferences.getInstance();

// Leggi il valore del contatore dalla memoria persistente
// Se non esiste, restituisce 0 come valore di default
final counter = prefs.getInt('counter') ?? 0;
```

```
// Ottieni le SharedPreferences
final prefs = await SharedPreferences.getInstance();

// Rimuovi la coppia chiave-valore 'counter' dalla memoria persistente
await prefs.remove('counter');
```

6.6.2 Lettura/Scrittura su file

Per manipolare file, si combina il plugin `path_provider` con la libreria `dart:io`. Si seguono i seguenti passaggi:

1. Si definisce un path locale per il file;
2. Si crea un riferimento al file;

3. Si scrive/legge il file

Il package `path_provider` consente di accedere alle **directory comuni** del file-system del dispositivo che si sta utilizzando, senza doversi preoccupare di formattare correttamente il percorso. In particolare supporta **2 tipologie di accessi**:

- `Temporary Directory`: **cache** che l'OS è libera di cancellare;
- `Documents Directory`: directory per i file dell'app a cui solo la stessa può accedere. Viene cancellata quando l'app viene **disinstallata**.

```
import 'package:path_provider/path_provider.dart';
// Accesso alla documents directory
Future<String> get _localPath async {
    final directory = await getApplicationDocumentsDirectory();
    return directory.path;
}
```

Una volta ottenuto il riferimento alla directory, possiamo creare il file file tramite la classe `File` del package `dart.io`.

```
Future<File> get _localFile async {
    final path = await _localPath;
    return File('$path/counter.txt');
}

Future<File> writeCounter(int counter) async {
    final file = await _localFile;
    return file.writeAsString('$counter');
}
```

```
Future<int> readCounter() async {
    try {
        final file = await _localFile;
        // Read the file
        final contents = await file.readAsString();
        return int.parse(contents);
    } catch (e) {
        // If encountering an error, return 0
        return 0;
    }
}
```

6.6.3 Database (SQLite)

Per molti dati più elevati la scelta più comune è quella di utilizzare un database **SQLite**. Si tratta di un database utilizzato estensivamente nelle applicazioni mobile grazie alla possibilità di essere "embeddato" all'interno dell'applicazione stessa.

Il plugin più utilizzato per l'interazione con SQLite è `sqflite`.

| |
|---------------------|
| ☒ 1 Error in region |
| ☒ 1 Error in region |