

UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Corso di Laurea Triennale in Informatica

Tesi di Laurea

PROGETTAZIONE E SVILUPPO DI
UN'APPLICAZIONE MOBILE
CROSS PLATFORM
PER IL MONITORAGGIO
DI DATI NAUTICI

Relatore:

Prof. IVAN SCAGNETTO

Laureando:

FEDERICO ZANARDO

Correlatore:

Prof. FRANCESCO TREVISAN

ANNO ACCADEMICO 2019-2020

Alla mia famiglia e ai miei cari per esserci sempre stati nei momenti difficili.

Ad Alessia per il suo amore,
per aver sempre creduto in me e per esserci sempre stata nel momento del bisogno.

Agli amici che non se ne sono mai andati.
A tutte le persone che mi sono state vicine in questo percorso.

”Quando qualcosa è abbastanza importante,
la fai anche se le probabilità di successo non sono a tuo favore.”

- Elon Musk

Introduzione

Il lavoro descritto in questa tesi nasce dall’esperienza di tirocinio interno svolto presso il Dipartimento di Scienze Matematiche, Informatiche e Fisiche dell’Università degli Studi di Udine. L’obiettivo principale di questo tirocinio è stato quello di realizzare un’**applicazione mobile cross-platform** per il monitoraggio dei dati provenienti da un server installato su una barca a vela, in tempo reale. Il contesto d’uso di questa applicazione è la **regata**. Un altro obiettivo molto importante del tirocinio è stato quello di approfondire il mondo dello sviluppo di applicazioni cross-platform, valutandone pregi e difetti concretamente durante la realizzazione dell’applicazione.

L’applicazione va ad integrare un progetto già avviato dall’Università degli Studi di Udine: il progetto **UniUD Sailing Lab**. È un progetto di ricerca del Dipartimento Politecnico di Ingegneria e Architettura dell’Università.

In passato sono state sviluppate sia un’applicazione Web che una per dispositivi Android per il medesimo scopo. L’applicazione Android riuscì a sopprimere alcuni svantaggi della versione Web, aumentando l’usabilità dell’applicazione, migliorando la fruizione delle informazioni e migliorando le prestazioni complessive dell’app. Inoltre permetteva di utilizzare le caratteristiche intrinseche presenti nel dispositivo, eliminando così le limitazioni imposte dal browser.

Tuttavia mancava una versione dell’applicazione per i dispositivi iOS. Pertanto si è deciso di implementare una nuova applicazione che, a partire da un unico codice sorgente, potesse essere distribuita per entrambi i sistemi operativi.

Indice

Introduzione	v
Elenco delle figure	xi
1 Introduzione al contesto di applicazione del progetto	1
1.1 Concetto base della regata	1
1.2 Regata costiera	1
1.3 Barca a vela	2
2 Architettura hardware e software	3
2.1 Hardware	3
2.1.1 Premessa	3
2.1.2 Raspberry	3
2.1.3 Ricevitore GPS	5
2.2 Software	5
2.2.1 Modello architetturale	5
2.2.2 Funzionalità implementate dal server	6
2.2.3 Argos	7
2.2.4 Neptune	9
3 Classificazione delle applicazioni mobili	13
3.1 Introduzione	13
3.2 Native	13
3.3 Web App	15
3.4 Progressive Web App	16
3.5 Hybrid	16
3.6 Cross-Platform	17
3.7 Considerazioni	19
3.8 Criteri per la scelta dell'approccio di sviluppo	20
3.8.1 Risorse economiche a disposizione	20
3.8.2 Time-to-market e copertura del mercato (reach)	21

3.8.3	Performance dell'applicazione	21
3.8.4	User Experience e interfaccia grafica	21
3.8.5	Funzionalità	22
3.9	Motivazione della scelta di un framework cross-platform	22
4	Dart	23
4.1	Introduzione al linguaggio	23
4.2	Paradigmi	24
4.2.1	Programmazione reattiva	24
4.2.2	Vantaggi e svantaggi della programmazione reattiva . . .	26
4.3	Compilazione ed esecuzione	26
4.3.1	Criteri per la classificazione dei linguaggi di programmazione	26
4.3.2	Tipizzazione	28
4.3.3	Compilazione ed esecuzione del codice	29
4.3.4	Compilatori ed interpreti	30
4.3.5	DartVM	31
4.4	Principali caratteristiche	31
4.4.1	Isolate	31
4.4.2	Event Loop	32
4.4.3	Future	33
4.4.4	Async e await	33
4.4.5	Stream	34
4.4.6	dynamic	34
4.4.7	Gestore dei pacchetti	35
4.5	Familiarità della sintassi	35
4.6	Confronto con JavaScript	36
4.6.1	Introduzione a JavaScript	36
4.6.2	Confronto	37
5	Flutter	39
5.1	Introduzione	39
5.2	Struttura del framework	39
5.2.1	Skia	40
5.3	Widget	41
5.3.1	Classificazione dei Widget	42
5.3.2	Costruzione dei Widget	43
5.4	State Management	44
5.4.1	setState	45
5.4.2	InheritedWidget	45
5.4.3	Scoped Model	46

5.4.4	Redux	46
5.4.5	Provider e BLoC	47
5.5	Hot Reload	47
5.6	Fuchsia	47
5.7	Ciclo di vita delle applicazioni	49
5.8	Pacchetti	49
5.9	Vantaggi e svantaggi di Flutter	50
5.10	Confronto con React Native	51
5.10.1	Introduzione a React Native	51
5.10.2	Confronto	52
6	Sviluppo dell'applicazione	55
6.1	Requisiti	55
6.2	Architetture utilizzate	56
6.2.1	Introduzione	56
6.2.2	Scelta dell'architettura	56
6.2.3	Provider	57
6.2.4	BLoC	62
6.2.5	Motivazioni della scelta	64
6.3	Architettura proposta	65
6.3.1	Problematiche del pattern BLoC	65
6.3.2	Introduzione del Repository pattern	65
6.3.3	Architettura modificata	66
6.4	Architettura implementata	67
6.4.1	Descrizione	67
6.4.2	Integrazione del Repository	68
6.5	Descrizione dettagliata dell'architettura implementata	69
6.5.1	Organizzazione delle classi e dei package	69
6.5.2	Modelli	72
6.5.3	Socket	77
6.5.4	Repository	78
6.5.5	BLoC	82
6.5.6	View	85
6.5.7	Widget comuni	107
6.5.8	Palette di colori	111
6.5.9	Temi	111
6.5.10	Main	113
6.6	Motivazioni delle scelte grafiche implementate	114
6.6.1	Concetti generali	114
6.6.2	Scelte stilistiche attuate	115
6.7	Ottimizzazioni	116

6.7.1	Provider e StreamBuilder	116
6.7.2	Annidamento degli StreamBuilder	116
6.7.3	Ottimizzazioni grafiche	117
7	Conclusioni	121
7.1	Sviluppi futuri	121
7.1.1	Notifiche push	121
7.1.2	Grafici e schermate dedicate ai dati	122
7.1.3	Aggiunta di componenti grafiche	124
7.1.4	Introduzione di nuovi temi	124
7.1.5	Autenticazione degli utenti	124
7.2	Considerazioni sull'approccio cross-platform	124
7.3	Considerazioni su Dart	125
7.4	Considerazioni su Flutter	126
7.5	Considerazioni personali	127
7.5.1	Scelta del framework	127
7.5.2	Progettazione e sviluppo dell'applicazione	128
A	Classi ed interfacce in Dart	129
B	AutomaticKeepAliveClientMixin	131
	Bibliografia	135

Elenco delle figure

1.1	Barca a vela Williab B	2
2.1	Raspberry Pi 3 Model B	4
2.2	Ricevitore GPS	5
2.3	Argos	8
2.4	Neptune - versione Web	10
2.5	Neptune - versione Android	12
3.1	Google Trends - Framework Cross-Paltform	17
3.2	StackOverflow - Framework Cross-Paltform	18
3.3	Confronto Native, Hyrbid, Web	19
4.1	Programmazione reattiva	25
4.2	Event loop	32
5.1	Architettura del framework Flutter	40
5.2	Widget Stateful e Stateless	44
5.3	State management - Redux	46
5.4	Flutter - Supporto sistemi operativi	48
6.1	Provider - Inizializzazione	59
6.2	Provider - Aggiornamento	60
6.3	Provider - Architettura complessiva	61
6.4	BLoC - Architettura complessiva	62
6.5	BLoC - Flusso di dati e di eventi	63
6.6	Architettura implementata	68
6.7	Screenshot - Navigation	88
6.8	Screenshot - Drawer e della schermata Settings	89
6.9	Screenshot - Dashboard	93
6.10	Screenshot - Drawer e della schermata Settings	97
6.11	Screenshot - Cache	98
6.12	Screenshot - Alert Dialog in Android e iOS della schermata Cache	99

6.13 Screenshot - Server	100
6.14 Screenshot - Controllo dello stato della connessione nella schermata Server	101
6.15 Screenshot - Salvataggio dei dati con successo nella schermata Server	103
6.16 Screenshot - Alert Dialog di errore nella schermata Server	104
6.17 Screenshot - Schermata per la scelta del tema	105
6.18 Screenshot - Dashboard e Navigation con tema ad alto contrasto	106
6.19 Screenshot - GridBox Widget	108
6.20 Screenshot - Location Widget	108
6.21 Screenshot - Location Widget nelle schermate Dashboard e Navigation	109
6.22 Screenshot - Page App Bar	110
6.23 Screenshot - Sensor Data App Bar	110
6.24 Screenshot - Schema di Event Queue e Microtask Queue	119
7.1 Sviluppi futuri - Dashboard	122
7.2 Sviluppi futuri - Location e Navigation	123

Capitolo 1

Introduzione al contesto di applicazione del progetto

1.1 Concetto base della regata

Una **regata** è una competizione tra imbarcazioni che possono essere a remi oppure a vela. Tale competizione consiste nel completare, una o più volte, un percorso prestabilito tramite un *bando di regata*[1]. La partenza delle imbarcazioni prevede un conto alla rovescia preceduto da dei segnali acustici (sirene) e visivi (bandiere o nel caso di competizioni notturne, segnali luminosi). Le imbarcazioni devono effettuare un percorso prestabilito da diverse boe che sono state posizionate in diversi punti del campo della regata. Il rappresenta il luogo in cui viene svolta la competizione. L'arrivo è contrassegnato da una linea immaginaria tracciata da due boe o da un cordino teso tra due sostegni galleggianti.

1.2 Regata costiera

Il campo della regata in questa categoria di competizioni può essere sia il mare che i laghi e i corsi d'acqua dolce, purché siano sufficientemente ampi. Nel contesto in cui verrà utilizzata l'applicazione, il campo della regata è il mare [2].

Il tracciato della regata è costituito dal posizionamento dei **waypoint** nel campo della regata, ovvero dei punti definiti nello spazio (coordinate geografiche) che rappresentano le posizioni relative ad ostacoli (possono essere boe o barche). È di fondamentale importanza posizionare correttamente i waypoint, in quanto un errato posizionamento di essi può generare degli scompensi durante la competizione. Tuttavia, durante la regata possono verificarsi delle



Figura 1.1: Barca a vela da regata *William B* [3]

condizioni non previste, in particolare condizioni meteo inaspettate, come del vento forte, che possono spostare fisicamente le boe nel campo di regata. Conseguentemente a questo fatto, i partecipanti hanno delle rotte differenti rispetto alla posizione effettiva della boa.

1.3 Barca a vela

Il lavoro svolto in questa tesi va a contribuire ad un progetto di ricerca dell'Università degli Studi di Udine: **UniUD Sailing Lab**. È un progetto di ricerca del Dipartimento Politecnico di Ingegneria e Architettura in cui viene utilizzata una barca a vela da regata. La barca **William B** è un modello *Archam-bault A35* ed è dotata di sensori ed hardware in grado di raccogliere e memorizzare dati in tempo reale sulla navigazione per il controllo e la gestione delle attività di bordo. L'applicazione mobile va a fornire un supporto all'equipaggio per le decisioni da prendere e le strategie da applicare durante la competizione.

Capitolo 2

Architettura hardware e software

2.1 Hardware

2.1.1 Premessa

Per l'implementazione dell'architettura hardware è stata utilizzata della componentistica semplice ed economica. Principalmente l'hardware utilizzato è un *Raspberry* ed un *ricevitore GPS*, per ricevere le coordinate geografiche della barca durante la competizione.

2.1.2 Raspberry

A bordo della barca è stato installato un **Raspberry Pi 3 Model B**, un computer di dimensioni ridotte (*single board computer*) che non necessita di eccessivi consumi di energia elettrica e dal costo contenuto. La particolarità di non avere della componentistica di alto livello permette di eseguire un qualsiasi sistema operativo basto su kernel Linux, fornendo un enorme vantaggio dal punto di vista del consumo energetico. Essendo la barca a vela il contesto in cui viene utilizzata questa *single board*, non vi è la possibilità di fornire grandi quantità di energia per sostenere il carico energetico di hardware di alto livello. Pertanto la durata della batteria è un aspetto di fondamentale importanza.



Figura 2.1: Raspberry Pi 3 Model B.

Il modello del Raspberry in questione ha le seguenti caratteristiche:

1. CPU Quad Core 1.2GHz Broadcom BCM2837 a 64bit
2. 1GB di RAM
3. Modulo WLAN BCM43438 e *Bluetooth Low Energy* (BLE) integrato
4. 10/100 Fast Ethernet
5. 2 porte USB 2.0
6. Uscita video HDMI
7. Lettore schede Micro SD integrato dove risiede il sistema operativo
8. Porta di alimentazione Micro USB

Il Raspberry deve ospitare un'applicazione server che fornirà dei servizi ai client, comunicare con i vari sensori installati nella barca a vela e mantenere attiva una rete WiFi a cui i vari client possono connettersi per accedere ai servizi del server. Quindi, il Raspberry ha anche il ruolo di *access point wireless*.



Figura 2.2: Ricevitore GPS collegabile via USB.

2.1.3 Ricevitore GPS

Per tracciare la rotta da percorrere è necessario ottenere le coordinate geografiche, fornite tramite il *servizio GPS* [4]. Pertanto viene utilizzato un ricevitore GPS a basso costo, collegabile via USB al Raspberry. L'utilizzo del GPS risulta essere lo strumento ideale in questo contesto, in quanto, trovandosi in un'ampia area, gli eventuali errori vengono mitigati e non compromettono la precisione della posizione determinata.

2.2 Software

2.2.1 Modello architetturale

Per questo progetto si è deciso di utilizzare un'architettura **Client-Server**. È un modello architettonico per sistemi distribuiti dove le funzionalità che offre il server sono presenti sotto forma di servizi disponibili ai client. I client accedono al server per usufruire dei servizi offerti.

Il principale vantaggio che si può trarre da questa architettura è la possibilità di accedere al server da più postazioni. In questo progetto risulta molto utile accedere al server da postazioni differenti o supportare connessioni multiple (utilizzare l'applicazione da mobile e l'applicazione Web). L'indipendenza tra server e client è un importante vantaggio in quanto è possibile apportare delle modifiche ad uno dei due *attori* senza influire sull'altro o su altre parti del sistema. I client devono conoscere il nome del server disponibile (o i no-

mi dei server nel caso in cui ve ne siano più di uno) e i servizi che offre. Il server invece non necessita di conoscere i client che si connettono a lui ed è in grado di gestire un numero arbitrario di connessioni. I client usufruiscono dei servizi offerti dal server tramite un protocollo di *richiesta-risposta*, dove un client invia una richiesta e rimane in attesa di una risposta da parte del server. In particolare, il protocollo implementato per questo progetto permette la comunicazione tra i due attori tramite l'utilizzo delle *socket*, un'astrazione che fornisce una API per l'architettura TCP/IP.

Lo svantaggio maggiore di questa architettura è la suscettibilità ad attacchi *Denial-of-Service (DoS)*, oltre ad essere soggetta ad eventuali guasti del server. Inoltre le prestazioni possono essere imprevedibili in quanto dipendono sia dal carico della rete, sia dal carico di lavoro che il server deve sostenere.

È necessario prestare attenzione alla separazione tra il concetto di *server fisico* e l'*applicazione server*: il server fisico è una macchina fisica dotata di componentistica in grado di poter supportare le funzionalità tipiche di un server (grandi quantità di risorse di calcolo e di memoria). L'applicazione server è un software che fornisce dei servizi per i client e che può essere installato indipendentemente che il dispositivo sia un server fisico o un normale personal computer. Un discorso analogo può essere fatto anche per i client.

2.2.2 Funzionalità implementate dal server

Comunicazione con i sensori

Una delle funzionalità principali che il server implementa è la comunicazione con i sensori installati sulla barca a vela. In particolare, il server raccoglie i dati in tempo reale, leggendoli in *modalità seriale*. La comunicazione tra il server ed i sensori avviene tramite la centralina **NMEA 183 Interface Output**, sviluppata dall'azienda *NKE Marine Electronics*. Questa centralina opera da intermediario tra i dati rilevati dai sensori ed il Raspberry. In particolare, la centralina converte i dati che provengono dai sensori in formato *ASCII* e li trasmette tramite la porta seriale al server, utilizzando il formato **NMEA 0183**, un formato standard di comunicazione utilizzato per applicazioni nautiche e per dati GPS.

Manipolazione dei dati raccolti

Una volta prelevati i dati dai sensori, questi vengono memorizzati in un *buffer*, pronti per essere inviati ad un qualsiasi client che ne fa richiesta. I dati vengono parsati in JSON, un formato per lo scambio di informazioni molto usato per le comunicazioni di rete. Inoltre, i dati prelevati dai sensori vengono utilizzati per la generazione di file di *log*, necessari per eseguire delle simulazioni a terra.

2.2.3 Argos

Sul Raspberry è stato installato il software **Argos**. Argos è un'applicazione lato server che offre dei servizi ai client che vi si connettono. È scritto in *C* e permette di raccogliere i dati registrati dai sensori installati sulla barca. Questi dati vengono notificati dai sensori al server ad intervalli regolari, in modo da averli sempre aggiornati ai fini delle decisioni che devono essere prese durante la competizione. Per rendere disponibili questi dati ai client, Argos apre una socket nella rete locale e rimane in ascolto sulla porta 4545: appena un client si connette, il server restituisce il set di dati formattato in JSON e chiude la comunicazione, ritornando in ascolto sulla porta.

Prima versione

La prima versione dell'applicazione server rimaneva in ascolto su due porte: la porta 4545 e la porta 4546. Tramite la porta 4545, venivano accettate le richieste dei client che volevano ottenere le informazioni riguardo ai sensori della barca in formato JSON. Tramite la porta 4546, invece, i client potevano inviare delle richieste in formato JSON, differenti rispetto alle prime. Queste richieste indicavano al server di svolgere determinate azioni. Quindi, le richieste venivano elaborate dal server e la risposta veniva fornita tramite la porta 4545.

Seconda versione

La seconda versione del server ha rimosso la funzionalità della porta 4546, ovvero, è stata tolta la possibilità di ricevere dei dati da parte dei client, di elaborarli e di restituire una risposta. Quindi, l'ultima versione permette soltanto di connettersi alla porta 4545 per ottenere i dati provenienti dai sensori della barca, parsati e formattati dal server.

Figura 2.3: La seconda versione del server *Argos* in esecuzione.

Dati rilevati dai sensori ed elaborati dal server

I dati che vengono rilevati dai sensori e poi inviati al server, vengono salvati su un file nel formato JSON. I sensori notificano periodicamente il server della disponibilità di nuovi dati: in questo modo l'utente avrà a disposizione sempre un set di dati aggiornati. Oltre ai dati forniti dai sensori, il server mette a disposizione degli ulteriori dati necessari per effettuare il *debugging*. Tali dati per il debugging vengono forniti dalla socket insieme a tutti gli altri dati dei sensori, senza la necessità di dover effettuare delle particolari richieste.

Di seguito verrà fornita una descrizione di alcuni dei dati che vengono raccolti dai sensori e poi resi disponibili dal server:

1. **A.W.S.** (Apparent Wind Speed): identifica la velocità del vento apparente o stimata;
 2. **A.W.A.** (Apparent Wind Angle): identifica l'angolo del vento apparente;
 3. **S.O.G.** (Speed Over Ground): indica la velocità effettiva della barca esente dagli effetti delle correnti marine;
 4. **C.O.G.** (Course Over Ground): indica la direzione vera della barca, esente dagli effetti delle correnti marine. Rappresenta la rotta che sta percorrendo la barca;

5. **M.H.** (Magnetic Heading): identifica il valore della direzione, segnalata dalla bussola di bordo;
6. **S.O.W.** (Speed Over Water): rappresenta la velocità della barca sull'acqua;
7. **T.W.S.** (True Wind Speed): indica la velocità attuale del vento;
8. **T.W.A.** (True Wind Angle): rappresenta l'angolo percepito del vento;
9. **I.H.** (Imu Heading): rappresenta la direzione data dalla IMU, l'angolo rispetto al Nord magnetico;
10. **C.ANG.** (Calypso Ang): indica l'angolo con cui il vento arriva sullo strumento di misurazione *Calypso*, ovvero, l'angolo misurato in base alla tacca segnata sullo strumento;
11. **C.AMP.** (Calypso Amp): identifica la forza del vento, misurata in nodi, dallo strumento *Calypso*;
12. **NWANG**: indica l'angolo del vento rispetto al Nord magnetico. È dato somma di *IMU Heading* e *Calypso Ang*;
13. **LATITUDE**: latitudine;
14. **LONGITUDE**: longitudine.

Nell'applicazione finale, questi dati vengono distribuiti nelle varie schermate. Alcune schermate mostreranno un piccolo set di questi dati per focalizzarsi su determinati aspetti come la navigazione, il vento o le coordinate GPS. La dashboard invece offre un quadro generale della situazione, illustrando tutti i dati appena descritti.

I dati elencati non sono tutti quelli che vengono forniti dalla socket: i dati illustrati sono soltanto quelli che vengono utilizzati all'interno dell'applicazione.

2.2.4 Neptune

Premessa

Il lavoro riguardo alla realizzazione dell'applicazione Web è stata dapprima affrontata dallo studente *Davide D'Osvaldo*. Un miglioramento di tale applicazione è stato effettuato prima da *Edoardo Polce* e successivamente da *Alessandro Martin*. Il mio lavoro ha preso in considerazione la versione dell'applicazione di quest'ultimo.

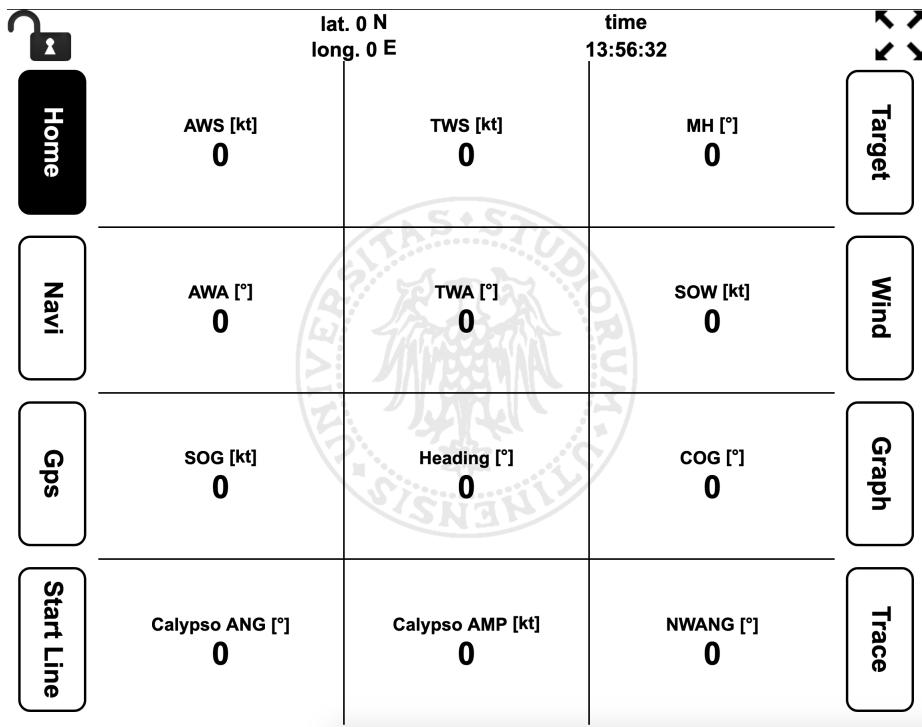


Figura 2.4: Schermata Home dell'applicazione Neptune nella versione Web.

Lo sviluppo dell'applicazione per dispositivi Android è stato effettuato dallo studente *Andrea Mazzocut*, di cui farò riferimento in questa sezione.

Obiettivo dell'applicazione

Neptune è un'applicazione lato client che permette di connettersi al server (tramite una socket alla porta 4545). Il client deve necessariamente essere connesso alla rete WiFi creata dal Raspberry. Ad intervalli regolari, l'applicazione interroga la socket per ottenere dei dati riguardanti i sensori installati sulla barca. Questi dati, in formato JSON, vengono mostrati in un'apposita interfaccia grafica. Nel corso di questo progetto, diversi studenti hanno partecipato alla realizzazione di differenti tipologie di applicazioni, per usufruire dei servizi offerti dal server Argos.

Applicazione web

L'obiettivo di questa applicazione è la creazione di una sorta di *computer di bordo* per essere utilizzato dall'equipaggio durante la competizione. L'accesso

a Neptune è permesso a qualsiasi dispositivo che possa connettersi via wireless al Raspberry, in quanto l'applicazione è accessibile tramite il *browser*.

L'applicazione è costituita da diverse schermate: in particolare, è possibile notare una schermata che presenta una serie di dati per avere un quadro generale di tutti i segnali provenienti dai sensori e delle altre schermate che permettono di monitorare dei dati specifici. Così facendo è possibile avere sia una panoramica generale dei dati rilevati dai sensori, sia un focus preciso su un piccolo set di dati.

Il software è stato sviluppato in HTML, CSS e JavaScript per la parte *frontend*, mentre per la parte *backend* è stato utilizzato PHP. Tuttavia questa versione dell'applicazione porta con sè diversi svantaggi, soprattutto a livello di uso pratico. Durante la regata, vengono utilizzati dei dispositivi mobili come smartphone e tablet, in modo da poterli maneggiare facilmente in ogni situazione e in ogni condizione. Di conseguenza, dispositivi di dimensioni maggiori rispetto a quelli citati sopra non vengono presi in considerazione. Per usufruire dell'applicazione Web è necessario quindi collegarsi via browser al sito web che ospita l'applicazione. L'esperienza utente per questa tipologia di software e per i dispositivi utilizzati non è ottimale, in quanto non vengono sfruttate appieno tutte le funzionalità native del dispositivo mobile.

Applicazione Android

Lo sviluppo di un'applicazione *nativa* per dispositivi mobili con sistema operativo *Android* è stata presa in considerazione per cercare di sopperire agli svantaggi della versione Web. L'applicazione nativa non ha l'intento di estendere o introdurre delle nuove funzionalità al servizio offerto, bensì l'intento è quello di rendere il servizio più facile da utilizzare e più reattivo in termini di prestazioni. Infatti, così facendo, è possibile sfruttare appieno le caratteristiche dei dispositivi mobili, permettendo un uso più naturale del servizio e riducendo anche i consumi delle risorse del dispositivo. Sulla base di queste considerazioni, è possibile personalizzare meglio il servizio: ad esempio, in questa versione dell'applicazione, è stato possibile introdurre la funzionalità per cambiare l'indirizzo IP, quando lo si necessita. Nella versione Web questo poteva essere fatto soltanto dal programmatore che ha realizzato l'applicazione. La funzionalità appena descritta non ha un'importanza maggiore rispetto alle altre già esistenti, ma aiuta a migliorare il servizio nel caso in cui si renda necessario dover svolgere un'operazione di questo genere.

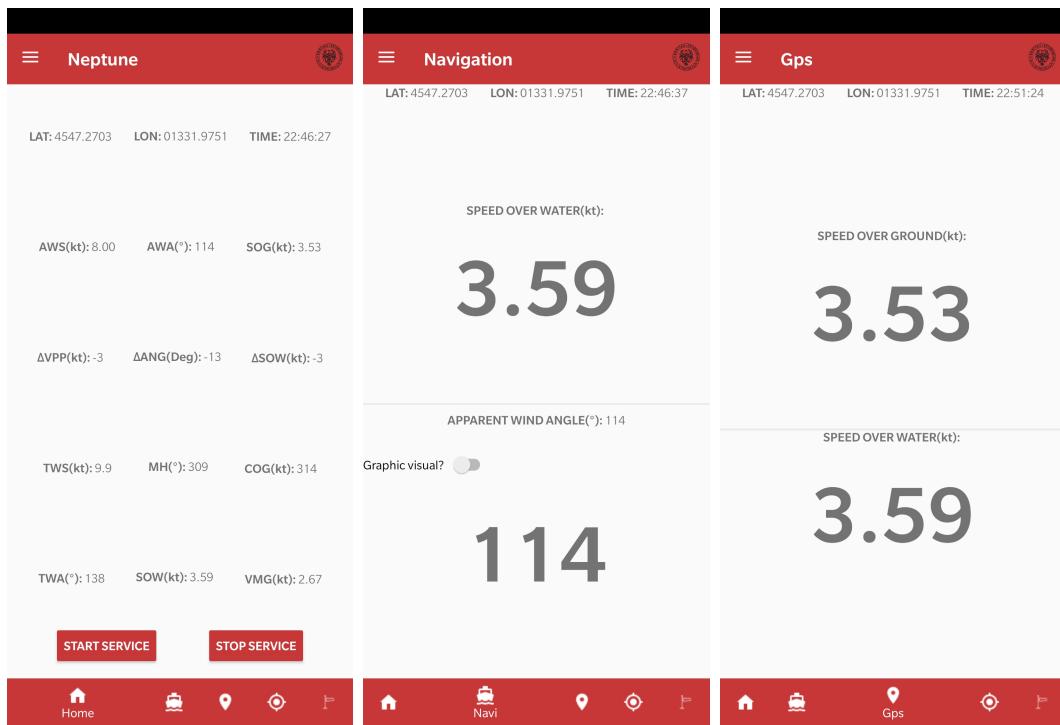


Figura 2.5: Alcune schermate dell'applicazione Android: schermate Home, Navigation e GPS (in ordine da sinistra).

Capitolo 3

Classificazione delle applicazioni mobili

3.1 Introduzione

In questo capitolo si procede all'esplorazione delle varie metodologie di sviluppo di applicazioni nate in questo ultimo decennio. Si vanno ad analizzare pregi, difetti e peculiarità di ogni approccio. Inoltre, si va a considerare quali possono essere i contesti d'uso in cui è preferibile utilizzare una metodologia rispetto ad un'altra.

3.2 Native

Nel corso di questo ultimo decennio, lo sviluppo di applicazioni *mobile* ha subito una profonda trasformazione ed evoluzione. Inizialmente lo sviluppo di applicazioni era possibile soltanto con i linguaggi di programmazione supportati dal sistema operativo. Di conseguenza era necessario conoscere le varie piattaforme (**Android**, **iOS** e **Windows Phone** sono i principali sistemi operativi per mobile che hanno caratterizzato questo mercato in questo decennio). Inoltre, per sviluppare applicazioni che potessero utilizzare appieno sia l'hardware che il software del dispositivo, era ed è tuttora necessario avere un'approfondita conoscenza di tutti i sistemi operativi su cui si vuole distribuire la propria applicazione.

Al giorno d'oggi i due competitor che coprono la quasi totalità del mercato mobile sono Android e iOS [5]. Essendo progettate per uno specifico sistema operativo, queste applicazioni sono ottimizzate e riescono a garantire delle prestazioni ottimali. Il linguaggio di programmazione utilizzato offre una serie di

funzionalità e librerie che permettono il completo accesso alla componentistica hardware del dispositivo (ad esempio, l'accesso ai sensori).

Possiamo osservare le considerazioni appena fatte da un punto di vista economico: se si vuole sviluppare un'applicazione e la si vuole distribuire sia su dispositivi Android che iOS, l'azienda dovrà assumere (almeno) uno sviluppatore per ogni sistema operativo. Più aumenta la complessità dell'applicazione, più persone dovranno essere aggiunte ai due team di sviluppo. Concettualmente, è necessario sviluppare *due codici sorgenti* della *medesima applicazione* per poterla distribuire su due *store* differenti (*Play Store* per Android e *App Store* per iOS). Questo comporta ad assumere diversi sviluppatori per realizzare quella che è concettualmente una singola applicazione.

Da un punto di vista tecnico, lo sviluppo della medesima applicazione da parte di due team che operano su ambienti differenti, portano a numerose difficoltà a livello di progettazione dell'applicazione e a livello organizzativo. Ad esempio, a parità di conoscenze ed esperienza dei singoli componenti dei team di sviluppo sulla piattaforma in cui operano (la conoscenza delle *hard skills*), l'introduzione di una nuova funzionalità può risultare semplice da sviluppare per un team, ma può diventare fonte di difficoltà per l'altro team. Questo può provocare dei rallentamenti nel rilascio della nuova funzionalità o può generare delle *asimmetrie* tra le due applicazioni, ovvero, una funzionalità può essere disponibile per un sistema operativo ma non nell'altro.

Volendo riassumere in punti i vantaggi e gli svantaggi dell'approccio nativo:

1. I vantaggi sono:

- (a) Le applicazioni native sono più veloci e più affidabili in termini di prestazioni. In generale sono più efficienti avendo a disposizione l'accesso alle risorse del dispositivo rispetto ad altri approcci di sviluppo;
- (b) Utilizzano l'interfaccia utente del dispositivo nativo, offrendo agli utenti un'esperienza più ottimizzata;
- (c) Avendo il pieno accesso all'hardware del dispositivo, può essere utilizzata un'ampia gamma di funzionalità, come ad esempio il giroscopio, la bussola, il Bluetooth, la posizione GPS ed altro ancora.

2. Gli svantaggi sono:

- (a) La non portabilità: una volta realizzata un'applicazione per uno specifico sistema operativo, questa non può essere utilizzata in altre piattaforme. Questo implica il bisogno di assumere diversi sviluppatori con conoscenze specifiche su uno specifico sistema operativo per

poter realizzare l'app. Le principali conseguenze di questo svantaggio sono l'aumento dei costi (sia in termini di tempo che economici) e la difficoltà nel manutenere l'applicazione;

- (b) Simmetria dei codici sorgenti: avendo più codici sorgenti per diversi sistemi operativi, si manifesta la difficoltà di mantenere aggiornati tutti i codici sorgenti nel momento in cui si vuole rilasciare una nuova funzionalità;
- (c) Ogni aggiornamento dell'applicazione deve essere scaricato manualmente dall'utente tramite l'apposito store;
- (d) L'applicazione contribuisce ad occupare lo spazio di memoria del dispositivo.

In origine, l'unico linguaggio permesso per la realizzazione di applicazioni per Android era **Java**. Oggi è possibile creare delle applicazioni anche con **Kotlin**. Analogamente, per quanto riguarda iOS all'inizio era permesso soltanto il linguaggio **Objective-C**, mentre, oggi è possibile sviluppare app anche con **Swift**, un linguaggio sviluppato appositamente dalla casa madre per i suoi dispositivi.

3.3 Web App

Tutte le considerazioni fatte a riguardo l'approccio nativo hanno portato allo sviluppo di applicazioni Web che fossero **mobile friendly**, ovvero, che potessero essere facilmente utilizzate anche da smartphone. L'intento principale è quello di sviluppare un'applicazione indipendentemente dalla piattaforma su cui verrà ospitata, ovvero, tramite l'utilizzo del *browser*, software installabile in qualsiasi dispositivo. Nonostante il nome possa trarre in inganno, queste applicazioni vengono denominate *Web app*. Le Web app non sono delle vere e proprie app, ma si cerca di rendere **responsive** il sito web, ovvero, di rendere il sito *mobile friendly*. Per *mobile friendly* si intende che la Web app deve avere un'interfaccia grafica che si adatti alle dimensioni dello schermo in maniera dinamica ed automatica. Sviluppando secondo questa concezione, è sufficiente implementare un unico codice sorgente per distribuire l'applicazione su qualsiasi dispositivo mobile e desktop (*Write once use everywhere*). L'utilizzo di queste applicazioni non comportano ad alcuna installazione e di conseguenza non incideranno sullo spazio di memoria del dispositivo. L'applicazione è sempre aggiornata all'ultima versione perché è sempre richiesta la connessione ad Internet per poterla usufruire. I costi per lo sviluppo sono abbattuti dal momento che il team che ha realizzato il sito web può anche sviluppare la versione per mobile, senza i problemi di distribuzione su piattaforme diverse.

Tuttavia non è possibile utilizzare appieno le risorse hardware e software che offre il dispositivo e di conseguenza l'ambito di applicazione di questa metodologia di sviluppo risulta essere limitato. L'interfaccia grafica e l'esperienza utente sono di basso livello e le applicazioni di questo genere sono molto limitate in termini di funzionalità.

Per realizzare queste applicazioni vengono utilizzati i linguaggi di programmazione per lo sviluppo di siti Web, ovvero, HTML, CSS, JavaScript e PHP.

In conclusione a questa sezione, possiamo notare che l'approccio nativo e quello delle Web app siano *diametralmente opposti*.

3.4 Progressive Web App

Un'evoluzione delle Web app sono le **Progressive Web App (PWA)**. Sono molto simili alle Web app, ovvero, mantengono alcune caratteristiche come la possibilità di sviluppare un unico codice sorgente per distribuire l'applicazione su qualsiasi dispositivo e la necessità di non doverla installare. Tuttavia, permettono di ricevere delle *notifiche push* e possono essere utilizzate anche in assenza di connessione ad Internet. Le PWA si appoggiano comunque al browser del dispositivo e quindi presentano le medesime limitazioni in termini di accessibilità delle Web app. Ad esempio, è possibile utilizzare il servizio GPS, ma non è possibile accedere alla fotocamera.

Per realizzare queste applicazioni vengono utilizzati i classici linguaggi per lo sviluppo di siti Web, ovvero, HTML, CSS e JavaScript. Per quanto riguarda i framework più comuni vi sono **Angular** (Google) e **ReactJS** (Facebook).

3.5 Hybrid

Le **app ibride** si pongono come un'evoluzione delle Web app e delle PWA e tentano di congiungere i vantaggi dello sviluppo nativo e delle PWA. Vengono create con i medesimi linguaggi per lo sviluppo Web, ma per funzionare necessitano l'installazione di un software sul device (*engine*). Questo software è uno strato che si interpone tra il dispositivo e l'applicazione ed è necessario per l'esecuzione del codice. L'*engine* permette di utilizzare in maniera più efficiente e profonda l'hardware del device, oltre ad effettuare il *rendering* della grafica. Ad esempio, è possibile realizzare applicazioni che facciano uso di sensori come l'accelerometro e il giroscopio.

Nonostante questi vantaggi, alcune funzionalità non possono essere realizzate e le prestazioni generali (compresa la *User Experience*) sono nettamente

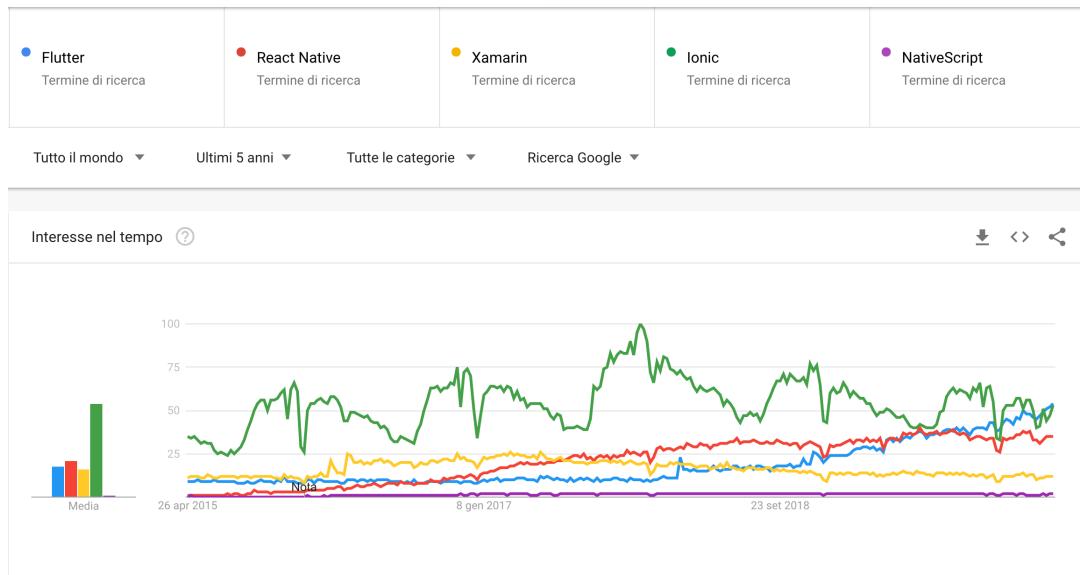


Figura 3.1: Trends dei framework cross-platform su Google Trends [6]

inferiori rispetto alle app native. Inoltre, rispetto alle Web app e alle PWA, queste applicazioni devono necessariamente essere pubblicate sullo store per poter essere installate sul dispositivo.

Il vantaggio fondamentale che si può ottenere è la possibilità di produrre un unico codice sorgente, utilizzando i classici linguaggi per lo sviluppo Web. Così facendo, è possibile distribuire l'applicazione su più dispositivi possibili, abbattendo in questo modo i costi.

I framework più comuni per la realizzazione di app ibride sono **Ionic** e **Apache Cordova**.

3.6 Cross-Platform

Le applicazioni **cross-platform**, anche dette *platform-independent*, sono applicazioni installabili su dispositivi con sistemi operativi differenti, senza la necessità di produrre un codice sorgente per ogni piattaforma. Sulla base di quanto appena detto, queste applicazioni possono essere confuse per applicazioni ibride, tuttavia hanno delle caratteristiche peculiari che le differenziano. In particolare, le app cross-platform utilizzano un *render engine* nativo che le permette di collegare il codice sorgente direttamente ai componenti nativi dei dispositivi. Così facendo, si aumentano notevolmente le prestazioni ed è possibile sfruttare meglio le risorse hardware rispetto alle applicazioni ibride. Le app

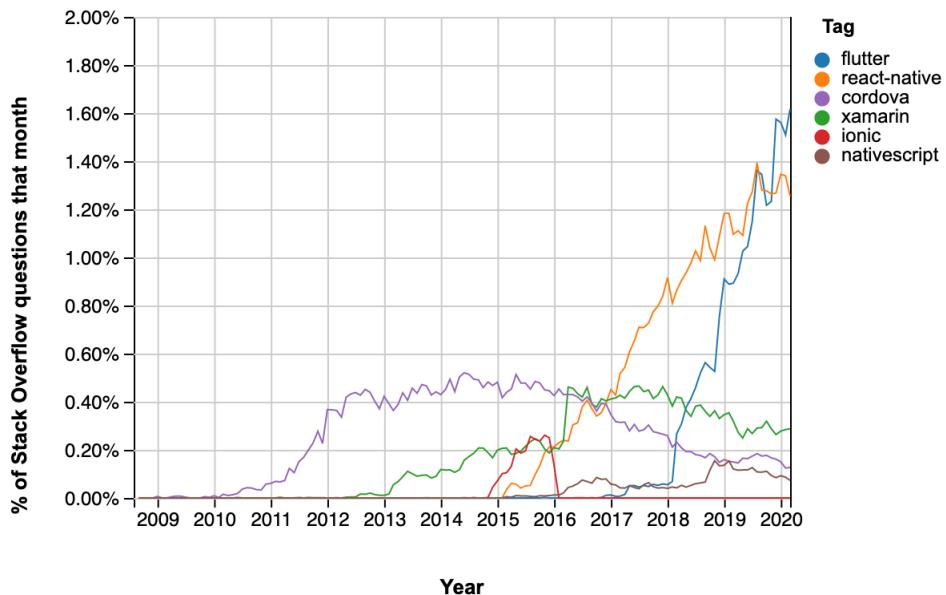


Figura 3.2: Trends dei framework cross-platform su StackOverflow [7]

ibride fanno uso, invece, di un render creato ad *hoc*. È comunque importante ricordare che soltanto le applicazioni native possono ottenere delle prestazioni ottimali: tutte le altre metodologie di sviluppo cercano di avvicinarsi ma non raggiungono i livelli di un approccio nativo.

In un'applicazione cross-platform è possibile implementare delle funzionalità particolari ed è possibile avere un interfaccia grafica fluida e reattiva, comparabile ad un'app nativa. Questo è l'approccio che più si avvicina alle applicazioni native e pertanto può essere considerata la metodologia più opportuna se si vuole sviluppare un'applicazione senza dover assumere due team di sviluppo differenti e senza sprecare le risorse economiche dell'azienda.

I framework più popolare in questo ambito di sviluppo sono **React Native** (Facebook), **Xamarin** (Microsoft) e **Flutter** (Google).

I linguaggi utilizzati variano da framework a framework: React Native utilizza *JavaScript*, Xamarin permette di scrivere codice in *C#* (linguaggio proprietario di Microsoft) e Flutter in *Dart* (linguaggio proprietario di Google).

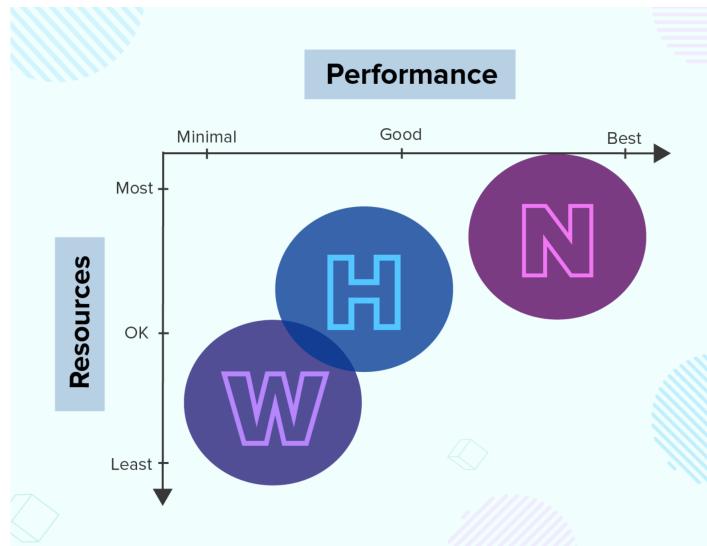


Figura 3.3: Confronto fra app Native, Hyrbid, Web [8].

3.7 Considerazioni

La decisione di esporre le varie metodologie di sviluppo non è stata presa in modo casuale. Si è voluto mettere in evidenza fin dall'inizio i due approcci in antitesi tra loro, ovvero, le *applicazioni native* e le *Web app*. A partire dalle *Web app*, si è proceduti verso un progressivo avvicinamento alle applicazioni native, pur tenendo conto che tutte le metodologie presentate, permettono di sviluppare un'app per più sistemi operativi con un unico codice sorgente. In questo modo si è proceduti ad illustrare tutte le possibilità offerte dal mercato per lo sviluppo di applicazioni mobili, considerando aspetti tecnici ed economici.

Dall'analisi effettuata, si possono suddividere gli approcci descritti principalmente in due categorie: *applicazioni native* e *applicazioni multipiattaforma*. *Web app*, PWA, Hybrid app e Cross-Platform app seguono la filosofia **"Write once use everywhere"** e **"Learn once apply everywhere"**, ovvero, la possibilità di realizzare un unico codice sorgente per distribuire l'applicazione su qualsiasi piattaforma, riutilizzando dei linguaggi di programmazione già esistenti, senza dover imparare uno specifico linguaggio per ogni singolo framework. I principi fondanti di questa categoria di approcci sono:

- 1. Manutenibilità:** è sicuramente il vantaggio più importante in termini di tempo e di risorse economiche risparmiate nel processo di realizzazione di un'applicazione. Sviluppando infatti un unico codice per l'applicazio-

ne, le attività di manutenzione, correzione e identificazione di eventuali malfunzionamenti risultano essere più veloci;

2. **Rapidità di sviluppo:** i tempi di realizzazione, dovendo scrivere un unico codice per tutte le piattaforme, si riducono notevolmente.

In conclusione, possiamo considerare che lo sviluppo di Web app (incluse le PWA) fanno un uso minimale delle risorse offerte dal dispositivo, mentre le applicazioni native fanno un largo uso delle risorse disponibili (Figura 3.3). Le applicazioni ibride si pongono come un compromesso tra la complessità delle applicazioni native e le forti limitazioni delle Web app. Le app cross-platform possono essere collocate vicino alle app native, sia per quanto riguarda la possibilità di utilizzare le risorse del dispositivo e sia dal punto di vista delle prestazioni, ma anche vicino alle Web app per quanto riguarda la semplicità di sviluppo e dell'uso di linguaggi comuni alla programmazione Web (Figura 3.3).

3.8 Criteri per la scelta dell'approccio di sviluppo

In questa breve sezione si prendono in considerazione alcuni degli aspetti e delle motivazioni che possono portare a scegliere un determinato approccio rispetto ad un altro [9].

3.8.1 Risorse economiche a disposizione

Quando si parla dello sviluppo di un'applicazione, un aspetto fondamentale e rilevante è l'*aspetto economico*. Nel contesto di una piccola azienda o di una *startup*, di solito sono disponibili pochi fondi per la realizzazione dell'applicazione. Pertanto, lo sviluppo nativo non è quello consigliato in quanto è necessario trovare uno sviluppatore che sia in grado di programmare sia per Android che per iOS o costruire due team di sviluppo per ogni sistema operativo. È evidente che questo approccio è il più *dispendioso*. In queste situazioni è molto importante concentrare tutti gli sforzi su un'unica piattaforma per ottenere un prodotto finale completo.

Se l'aspetto economico non è un problema, non è necessario comunque sviluppare delle app native. Ad esempio, se un'azienda ha già degli sviluppatori molto esperti in JavaScript, risulta inefficiente non sfruttare le potenzialità di questo team anche per lo sviluppo di un'applicazione per dispositivi mobile. Soprattutto, quando si hanno dei tempi stretti per la realizzazione del prodotto

e per la consegna, è consigliato di sfruttare appieno le risorse già presenti nell'ambiente aziendale, senza dover procedere con dei colloqui per assumere altri sviluppatori.

3.8.2 Time-to-market e copertura del mercato (reach)

La creazione di un'applicazione nativa richiede molto più tempo rispetto allo sviluppo di app ibride o PWA. Le tempistiche dettate dal mercato (**time-to-market**) influenzano le scelte anche di questo genere. Pertanto se si vuole realizzare un prodotto funzionante prima di un *competitor*, è necessario prendere in considerazione anche le vie alternative allo sviluppo nativo. Una volta che il prodotto è stato sufficientemente accettato dal mercato (quando ha superato la cosiddetta *massa critica*), allora si può pensare di ristrutturare l'applicazione e di scegliere altri approcci per gli sviluppi futuri.

Se parliamo in termini di **reach**, ovvero, di copertura del mercato con il proprio prodotto, è preferibile che l'applicazione possa essere distribuita nel più breve tempo possibile e su più *marketplace* possibili. Questo aspetto invita ad adottare un approccio ibrido o cross-platform, per poter generare con un unico codice sorgente, due versioni dell'app pronte per entrambi gli store.

3.8.3 Performance dell'applicazione

Se le prestazioni sono un fattore fondamentale ed un requisito indispensabile, l'approccio nativo è quello migliore di tutti. Tutti gli altri approcci introducono dell'*overhead*: si pensi ad esempio al bridge JavaScript di React Native per poter utilizzare gli elementi grafici nativi. Un'applicazione nativa ha il pieno accesso alle risorse hardware e software del dispositivo e in questo modo è possibile operare con precisione ed in profondità per ottenere dei vantaggi in termini di prestazioni e di ottimizzazioni.

3.8.4 User Experience e interfaccia grafica

Secondo una statistica del 2018 [10], quasi *una persona su tre* abbandona l'applicazione dopo la prima esperienza. Questo dato stimola la riflessione riguardo alla **UX (User Experience)** e la **UI (User Interface)**, aspetti fondamentali che devono essere curati se si vuole mantenere una buona *retention* dell'utente nell'applicazione. Le applicazioni native permettono meglio di qualunque altro approccio di realizzare delle animazioni dinamiche e degli scorrimenti fluidi. Dopo le applicazioni native, le app cross-platform sono quelle in grado di riprodurre al meglio una UX vicina a quella che può essere

implementata in ambiente nativo. Per quanto riguarda le applicazioni ibride e Web, l'esperienza va a mano a mano a degradarsi.

3.8.5 Funzionalità

Se l'applicazione che deve essere prodotta deve avere delle particolari funzionalità o deve utilizzare determinati sensori, l'approccio migliore è quello nativo perché è possibile accedere completamente a tutte le risorse del dispositivo. A partire dalle applicazioni cross-platform fino alle Web app, la possibilità di interagire con determinati componenti hardware installati nel dispositivo risulta essere sempre più difficile se non impossibile da attuare.

3.9 Motivazione della scelta di un framework cross-platform

Le motivazioni che mi hanno portato a scegliere di utilizzare un approccio cross-platform sono principalmente dovute per ragioni intrinseche del progetto da sviluppare. L'applicazione che deve essere realizzata necessita di visualizzare in *tempo reale* dei dati provenienti da una socket. In particolare si deve tener conto che l'applicazione debba essere *reattiva*. Inoltre si vuole rendere l'applicazione disponibile per più dispositivi mobili, con sistemi operativi differenti. Sulla base di queste considerazioni, l'approccio più plausibile da utilizzare è l'approccio cross-platform, in quanto:

1. Permette di creare un'unico codice sorgente per distribuire l'applicazione sia su Android che su iOS;
2. È l'approccio che più si avvicina a quello nativo, in termini di prestazioni e di accessibilità alla componentistica hardware del dispositivo;
3. È possibile realizzare un'interfaccia grafica che sia fluida e reattiva, grazie proprio alla maggior vicinanza con l'hardware del device.

La scelta della metodologia cross-platform rappresenta quindi un *trade-off* tra la possibilità di sviluppare un unico codice sorgente sia per Android che per iOS e le prestazioni dell'applicazione.

Capitolo 4

Dart

4.1 Introduzione al linguaggio

Dart è un linguaggio di programmazione ottimizzato per applicazioni multipiattaforma sviluppato da Google. Questo linguaggio è nato nel 2011 per lo sviluppo di applicazioni Web. Lo scopo di Dart è quello di risolvere i problemi strutturali di *JavaScript* che non possono essere risolti con l’evoluzione del linguaggio stesso. Queste affermazioni sono state scritte in una mail interna di Google, in cui viene definito *Dash* (il nome provvisorio di Dart quando il progetto era ancora in una fase embrionale) il linguaggio che dovrà ”*sostituire JavaScript come lingua franca per lo sviluppo web sulla piattaforma web aperto*”[11].

Dart offre delle prestazioni migliori, rende lo sviluppo più semplice e introduce migliori funzionalità legate alla sicurezza. Nonostante nei primi anni questo linguaggio non riuscì a riscuotere il successo che il team di Google sperava, l’azienda decise di utilizzarlo per lo sviluppo di applicazioni mobile ***platform-independent***, ovvero, per l’implementazione del framework *Flutter*. Inoltre, tramite questo linguaggio è possibile creare applicazioni anche per *desktop* (Windows, macOS, Linux/Unix-based e Fuchsia), lato *server* e per dispositivi *IoT*.

Gli obiettivi principali che il team di sviluppo di Google si è preposto di perseguire per la realizzazione di Dart, sono:

1. **Portabilità:** creare un linguaggio strutturato e flessibile per lo sviluppo di applicazioni per il Web e la possibilità di sviluppare applicazioni per tutti i device sul Web.;
2. **Produttività:** rendere Dart un linguaggio semplice e familiare ai programmatore, ovvero, con una *curva di apprendimento* bassa. Inoltre deve avere una sintassi chiara e concisa;

3. **Velocità:** i costrutti del linguaggio devono fornire delle alte performance ed un avvio veloce delle applicazioni;

In questo capitolo si andranno ad approfondire gli aspetti fondanti del linguaggio e gli aspetti più rilevanti, necessari per la realizzazione del progetto, senza elencare tutte le singole caratteristiche e funzionalità [14].

4.2 Paradigmi

Dart è un linguaggio *multiparadigma*, ovvero, è un linguaggio principalmente *orientato agli oggetti* ma offre anche le potenzialità della *programmazione funzionale*, della *programmazione imperativa* e della *programmazione event-driven*. Analogamente al linguaggio Java, fa uso di un *garbage collector*, ovvero, un meccanismo che libera la memoria occupata da oggetti che non sono più utilizzati dal programma. L'utilizzo di più paradigmi permette di ottenere un linguaggio flessibile e che unisce i vantaggi di ogni paradigma [15].

4.2.1 Programmazione reattiva

Oltre ai paradigmi già citati, Dart integra anche la *programmazione reattiva*, fondamentale per la realizzazione di applicazioni per dispositivi mobile prestanti, veloci ed efficienti [15]. Questo paradigma si fonda sulla nozione del cambiamento dei valori nel tempo, a seguito del verificarsi di un evento, e la conseguente propagazione di tali cambiamenti. Diverse fonti forniscono delle definizioni differenti:

1. **Reactive Manifesto:** Reactive Systems are Responsive, Elastic, Resilient, Message Driven [16];
2. **Wikipedia:** Reactive Programming is a programming paradigm oriented around data flows and the propagation of change [17].

Nonostante le differenze, si possono trarre dei concetti che vengono condivisi da entrambe le definizioni. In particolare, i concetti su cui si basa la programmazione reattiva, sono:

1. La nozione di *flusso di dati* e la *propagazione* dei cambiamenti nell'applicazione;
2. La possibilità di gestire facilmente dei *flussi asincroni*, sia che essi siano di dati o di eventi;

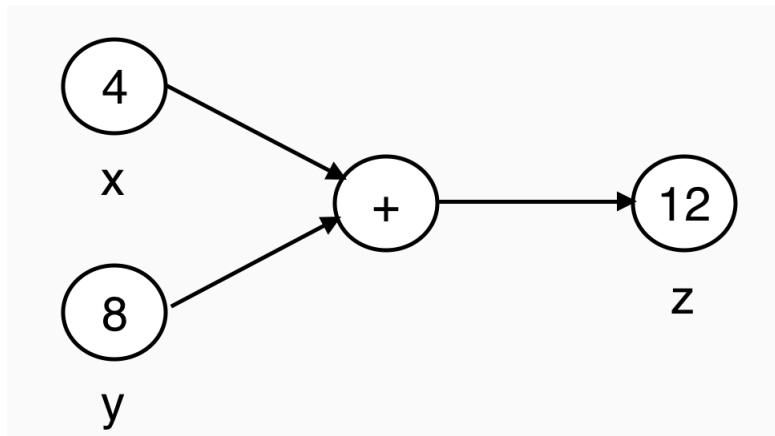


Figura 4.1: Rappresentazione della dipendenza delle variabili dell'esempio.

3. È un paradigma che ha una forte correlazione con la programmazione event-driven.

Dal punto di vista dello sviluppo, questo paradigma offre un importante vantaggio, in quanto il programmatore deve focalizzare la sua attenzione sul *che cosa deve fare* l'applicazione e può delegare la gestione al linguaggio su *quando deve essere fatto*. In questo modo, al verificarsi di un evento, i cambiamenti vengono propagati attraverso la rete delle **dipendenze computazionali** dal sottostante modello di esecuzione. Per mostrare concretamente i vantaggi della programmazione reattiva, si procede con l'esposizione di un esempio:

```

x = 4;
y = 8;
z = x + y;
  
```

La redazione di questo semplice codice con un linguaggio imperativo fa sì che il risultato di ***z*** sia 12 e che tale risultato non verrà modificato fino a quando non verrà assegnato esplicitamente un nuovo valore a tale variabile. In altri termini, ***z*** rappresenterà sempre la somma dei due valori iniziali. L'assegnamento di nuovi valori alle variabili iniziali non influenzano il risultato contenuto in ***z***. L'esecuzione di questo codice, scritto con un linguaggio che supporta la programmazione reattiva, mantiene aggiornato il valore di ***z*** ogni volta che almeno uno dei due addendi viene modificato: alla modifica di una delle due variabili, durante l'esecuzione del codice, viene automaticamente aggiornato il contenuto della variabile ***z***. Nel contesto della programmazione reattiva,

la variabile `z` è definita come *dipendente* dalle variabili `x` e `y`. È possibile rappresentare graficamente la relazione di dipendenza tra le variabili del codice di esempio (Figura 4.1) [17].

Il concetto di dipendenza viene realizzato attraverso l'implementazione di *ascoltatori*, che riescono ad intercettare i cambiamenti delle variabili da cui essi dipendono.

4.2.2 Vantaggi e svantaggi della programmazione reattiva

Il principale vantaggio che si può ottenere dalla programmazione reattiva è la facilità di sviluppo di un'applicazione, pur mantenendo distinta la *user interface (UI)* dalla *business logic*. È possibile generare del codice più compatto ed evitare di scrivere nel medesimo modulo o file sia codice per la UI e che codice per la business logic.

Tuttavia non è sempre vantaggioso l'utilizzo di questo paradigma, in quanto spesso rende complicata la leggibilità del codice. In particolare, segmentando l'interfaccia grafica in piccoli componenti, è possibile che per ogni componente debba essere associato un modulo della business logic. Pertanto, diversamente da altri linguaggi che non adottano la programmazione reattiva, la comprensione del codice non risulta essere naturale.

4.3 Compilazione ed esecuzione

4.3.1 Criteri per la classificazione dei linguaggi di programmazione

I linguaggi di programmazione possono essere classificati secondo diversi criteri, come i vari paradigmi che supportano o in base al livello a cui operano (si parla di linguaggio di *alto livello* o di *basso livello*). Un'altra classificazione può essere fatta suddividendo i linguaggi in *linguaggi compilati* o *linguaggi interpretati*.

Il sorgente di un programma scritto in un **linguaggio compilato** viene *tradotto* in un programma equivalente, scritto solitamente in un linguaggio di basso livello. Questo linguaggio di basso livello è comprensibile ad una certa *macchina* (macchina astratta o hardware), ovvero, all'ambiente in cui verrà eseguito il programma compilato. Oltre alle librerie e ad altri componenti esterni necessari per il funzionamento, un programma compilato non ha la necessità di utilizzare un secondo programma (*l'interprete*) per poter essere eseguito. Infatti, l'assenza di un interprete per questa tipologia di linguaggi

implica un minor utilizzo della memoria centrale e un minor tempo di esecuzione del codice. Tuttavia, questo fattore non è l'unico a determinare un minor tempo di esecuzione, in quanto dipende anche dalle caratteristiche della macchina su cui il programma compilato viene eseguito. Aspetto che lo differenzia dai linguaggi interpretati è l'assenza della presenza simultanea sia del compilatore che del sorgente, al momento dell'esecuzione del codice. Nei linguaggi compilati la fase di *debugging* risulta essere più efficace, in quanto a tempo di compilazione possono essere scoperti degli errori sui tipi delle variabili. D'altro canto, questa restrizione può portare al negare la compilazione di programmi il cui codice è lecito durante l'esecuzione, ma che non soddisfa i controlli statici eseguiti a tempo di compilazione. I linguaggi compilati, grazie a tutti i vantaggi che offrono, vengono ampiamente utilizzati in ambiti in cui la velocità di esecuzione del codice è di fondamentale importanza, dove le risorse hardware devono essere sfruttate appieno o tali risorse sono presenti limitatamente (principalmente per sistemi e dispositivi IoT ed applicazioni real-time). In queste situazioni si preferisce avere un maggior controllo delle risorse per sfruttarle nel modo più efficace ed efficiente possibile, a scapito di implementare con facilità il codice che dovrà essere eseguito.

Nei **linguaggi interpretati** si utilizza l'*interprete*, un programma necessario per l'esecuzione del codice. L'interprete rappresenta il luogo del controllo dell'esecuzione del codice, in quanto esso riceve in input il programma da eseguire e i dati da utilizzare e passo passo traduce in linguaggio macchina ogni singola istruzione sul momento. Questo funzionamento implica che il codice da eseguire e l'interprete debbano essere necessariamente presenti al momento dell'esecuzione del programma. La presenza simultanea dell'interprete e del programma comporta ad una minore efficienza a tempo di esecuzione: un programma interpretato richiede più memoria centrale ed è meno veloce a causa dell'*overhead* introdotto dall'interprete stesso. I linguaggi interpretati risultano essere più flessibili rispetto ai linguaggi compilati, in quanto tutti i controlli sui tipi (ed altri controlli) vengono effettuati al momento dell'esecuzione del codice. La fase di debugging risulta essere meno efficace in quanto è necessario eseguire il codice per scoprire eventuali bug presenti nel programma. Un notevole vantaggio che i linguaggi interpretati offrono è l'alta portabilità dei programmi, aspetto che non è possibile implementare per i linguaggi compilati in quanto il *codice oggetto*, generato dal compilatore, dipende strettamente dalla macchina in cui il programma è stato compilato.

I **linguaggi semi-interpretati o multipiattaforma** sono dei linguaggi che cercano di unire i vantaggi sia dei linguaggi compilati che dei linguaggi interpretati. Il linguaggio che rappresenta maggiormente questa categoria è Java. I programmi scritti in Java vengono compilati in una prima fase per generare

il *bytecode*, un programma equivalente a quello sorgente, scritto in istruzioni macchina per una particolare *macchina virtuale*: la *Java Virtual Machine*. Il bytecode viene interpretato dalla JVM, la quale esegue il programma sulla macchina fisica in cui si trova. Questo livello di astrazione permette agli sviluppatori di realizzare applicazioni che siano totalmente indipendenti dall'hardware su cui verranno eseguite, ovvero, permette di avere la *portabilità* delle applicazioni. Per ogni sistema operativo viene realizzata una JVM e tutte quelle devono comportarsi in modo uguale. Così facendo ogni programma scritto in Java è eseguibile su ogni sistema operativo che ha una JVM. Lo slogan che esprime al meglio il concetto che sta alla base di Java è: " *Write Once, Run Everywhere*" . In termini di prestazioni, i programmi Java sono meno efficienti dei programmi scritti in un linguaggio compilato. Questo è dovuto al fatto che vi è una fase di compilazione del programma Java e una fase di interpretazione del bytecode. Tuttavia, la portabilità dei programmi è una caratteristica peculiare del linguaggio e risulta essere molto utile nello sviluppo delle applicazioni.

4.3.2 Tipizzazione

I linguaggi di programmazione possono essere classificati secondo un altro criterio, ovvero, in base alla loro *tipizzazione*. I linguaggi possono implementare una:

1. **Tipizzazione statica:** a tempo di compilazione, vengono eseguiti i controlli sull'uso corretto dei valori rispetto al loro tipo;
2. **Tipizzazione dinamica:** i controlli sull'uso corretto dei tipi vengono fatti a tempo di esecuzione del codice.

Non vi è una suddivisione netta tra i due insiemi: i linguaggi che implementano una tipizzazione statica eseguono comunque dei controlli a tempo di esecuzione del codice. Ad esempio, il controllo sulla dimensione di un vettore è un controllo che deve essere necessariamente fatto a tempo di esecuzione perché tale controllo non può essere effettuato in alcun modo a tempo di compilazione.

Il vantaggio principale di una tipizzazione statica è la possibilità di rilevare gli errori a tempo di compilazione, prima ancora che il codice venga eseguito. Tuttavia questi linguaggi risultano essere meno flessibili rispetto ai linguaggi che adottano una tipizzazione dinamica. D'altronde, con la tipizzazione dinamica è necessario eseguire il programma per trovare eventuali errori di tipo.

Dart è considerato un linguaggio ***type safe*** [20], ovvero, garantisce l'esecuzione di controlli esaustivi sull'uso corretto dei valori rispetto al loro tipo,

non solo a tempo di compilazione. La combinazione di queste due tipizzazioni fornisce una maggiore solidità al codice. Avendo una tipizzazione sia statica che dinamica, il linguaggio permette due tipologie di compilazione: **AOT** (**Ahead Of Time**) e **JIT** (**Just In Time**).

4.3.3 Compilazione ed esecuzione del codice

Ahead Of Time (AOT)

La **compilazione anticipata** consiste nel compilare un programma scritto in un linguaggio di alto livello in un codice macchina nativo, in modo che il file binario risultante possa essere eseguito nativamente. Di conseguenza, il risultato della compilazione è dipendente dal sistema su cui è stata effettuata la compilazione [18]. Eseguendo una compilazione anticipata, è possibile produrre del codice ottimizzato per la macchina, analogamente ad un normale compilatore nativo. La differenza è che la compilazione anticipata trasforma il *bytecode* (codice intermedio [19]) di una macchina virtuale esistente in codice macchina. In particolare, la compilazione anticipata permette di compilare il codice intermedio nel codice macchina prima che il codice stesso venga eseguito. In questo modo è possibile limitare l'ambiente di *runtime*, risparmiando memoria, spazio su disco e durata della batteria. Per questo motivo, può essere utile per realizzare applicazioni per dispositivi mobili o per dispositivi che hanno uno spazio di memoria limitato e devono mantenere dei bassi consumi energetici.

In generale, i linguaggi statici sono gli unici linguaggi a poter supportare questa tipologia di compilazione. Il motivo è che i linguaggi macchina in genere devono conoscere il tipo delle variabili, pertanto nei linguaggi dinamici, dove il tipo non viene determinato in anticipo, questo controllo risulta essere inapplicabile. I linguaggi dinamici infatti sono in genere interpretati o compilati *Just-In-Time*.

Utilizzare la compilazione anticipata durante la realizzazione del software causa dei *cicli di sviluppo* più lenti. Per ciclo di sviluppo si intende il tempo che intercorre dal momento in cui viene apportata una modifica al codice di un programma e la possibilità di eseguire il programma per poter vedere il risultato, dopo tale modifica. Tuttavia questo approccio alla compilazione comporta che i programmi possono essere eseguiti in modo più prevedibile e senza la necessità di dover impiegare del tempo in fase di esecuzione del programma per l'analisi e la compilazione. Di conseguenza, grazie alla compilazione, i programmi possono essere avviati più rapidamente nella fase iniziale.

Just In Time (JIT)

Quando la fase di compilazione viene svolta a tempo di esecuzione del codice, si sta parlando di compilazione **Just-In-Time** [21]. Nella maggior parte dei casi, si tratta di dover compilare a *runtime* del codice sorgente o del bytecode in codice macchina, che viene quindi eseguito direttamente.

La compilazione JIT è una combinazione dell'approccio AOT e dell'approccio dell'interpretazione. Si cerca di combinare la velocità del codice compilato con la flessibilità dell'interpretazione. Tuttavia questo comporta l'introduzione sia dell'*overhead* di un interprete e dell'overhead della compilazione. La compilazione JIT è una forma di compilazione dinamica e consente l'attuazione di un'ottimizzazione adattiva come la *ricompilazione* dinamica. La compilazione JIT è particolarmente adatta per linguaggi di programmazione dinamici, in quanto il sistema di runtime può ritardare la gestione dei tipi di dato.

Contrariamente alla compilazione anticipata, la compilazione JIT fornisce dei cicli di sviluppo molto più veloci, ma può comportare ad un'esecuzione più lenta. In particolare, i compilatori JIT hanno tempi di avvio più lenti, perché quando il programma viene mandato in esecuzione, è necessario effettuare la fase di compilazione del programma prima che il codice possa essere eseguito.

4.3.4 Compilatori ed interpreti

Potendo utilizzare Dart per la realizzazione di applicazioni che potranno essere eseguite su dispositivi di natura diversa (mobile, desktop, Web), è necessario che il linguaggio sia fornito di un compilatore sufficientemente flessibile [22]. Infatti, i compilatori disponibili sono:

1. **Dart Native**: per eseguire applicazioni su device come smartphone, desktop e per applicazioni lato server. Dart Native include sia una macchina virtuale (Dart VM) con compilazione JIT (just-in-time), sia un compilatore AOT per la produzione di codice macchina;
2. **Dart Web**: per eseguire il codice Dart su piattaforme Web basate su JavaScript. Dart Web include sia un compilatore dei tempi di sviluppo (`dartdevc`) che un compilatore dei tempi di produzione (`dart2js`). Con Dart Web, il codice Dart viene compilato in codice JavaScript, che a sua volta viene eseguito in un browser. In particolare:
 - (a) **dartdevc**: è un compilatore Dart-to-JavaScript ottimizzato. Invece di utilizzare direttamente `dartdevc`, lo si utilizza con `webdev`, uno strumento che supporta le attività principali dello sviluppatore come l'esecuzione e il debug [23].

- (b) **dart2js**: è un tool che compila il codice Dart in JavaScript velocemente e in maniera compatta. Utilizza delle tecniche per l'eliminazione del codice che non viene mai chiamato durante l'esecuzione (*dead-code*).

4.3.5 DartVM

La **Dart Virtual Machine** non è propriamente una macchina virtuale come la *JVM*. La DartVM fornisce un ambiente di esecuzione e un insieme di componenti per l'esecuzione nativa di un linguaggio di alto livello, in questo caso Dart. Questo non implica che il codice Dart possa essere *sempre* interpretato o compilato *Just-In-Time* quando viene eseguito dalla DartVM. Infatti, DartVM offre diversi modi per interpretare il codice, in quanto la macchina virtuale può eseguire il codice utilizzando JIT o delle istantanee AOT. La scelta dipende principalmente da come e quando la macchina virtuale converte il codice sorgente Dart in codice eseguibile. Indipendentemente dall'opzione scelta, l'ambiente di runtime che facilita l'esecuzione rimane lo stesso.

Qualsiasi codice Dart, all'interno della macchina virtuale, è in esecuzione all'interno di un **Isolate**, ovvero uno spazio con una propria memoria e con un proprio thread di controllo. Possono esserci molti *Isolate* che eseguono del codice Dart contemporaneamente, ma non possono condividere direttamente nessuno stato. Vari *Isolate* possono comunicare solo tramite messaggi che passano attraverso determinate porte.

4.4 Principali caratteristiche

In questa sezione si va ad introdurre ed elencare alcuni dei costrutti fondamentali del linguaggio, che verranno successivamente utilizzati per la realizzazione del progetto.

4.4.1 Isolate

Nonostante Dart sia un linguaggio a singolo thread, offre il supporto a **Future**, **Stream**, lavoro in background ed altri meccanismi che permettono di scrivere codice in modo moderno, asincrono e reattivo [24].

Un **Isolate** è il luogo in cui viene eseguito tutto il codice Dart. È come un piccolo spazio sulla macchina con la sua zona di memoria privata ed un singolo thread che esegue un loop di eventi. Nella propria zona di memoria, nessun **Isolate** è autorizzato ad accedervi, nemmeno l'**Isolate** principale (che è il genitore). Da qui il nome **Isolate** per enfatizzare che questi spazi di memoria

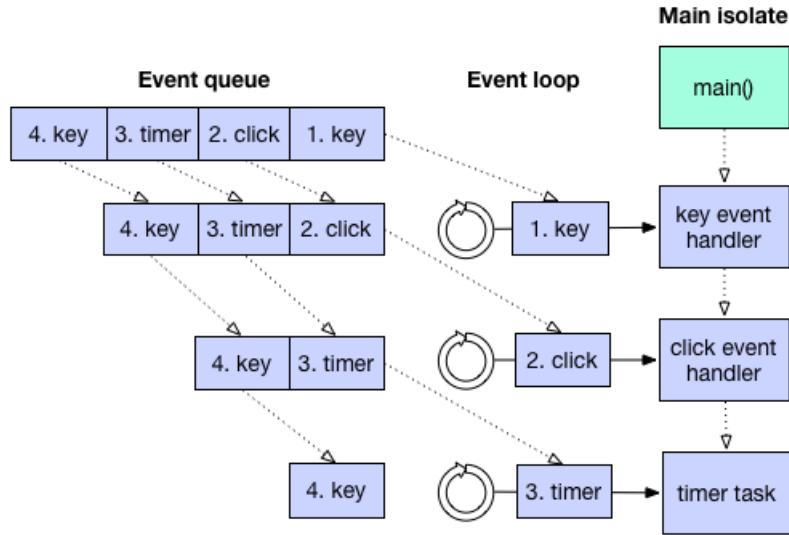


Figura 4.2: Illustrazione del funzionamento dell’*event loop* [26].

sono completamente isolati l’uno dall’altro. L’unica modalità di comunicazione concessa tra **Isolate** è tramite lo *scambio di messaggi* tra l’uno e l’altro.

Soltanamente, le applicazioni realizzate in Dart eseguono il loro codice in un singolo **Isolate**, ovvero, quello principale. Se necessario, è possibile creare altri **Isolate**: in particolare, quando devono essere eseguite delle elaborazioni pesanti e complesse che possono far degradare le prestazioni della UI, è possibile creare degli **Isolate** separati tramite il metodo `Isolate.spawn()` (nel caso di Flutter si potrà utilizzare il metodo `compute()`). Così facendo, si lascia l’**Isolate** principale libero e la parte più laboriosa del codice avviene in un **Isolate** secondario.

4.4.2 Event Loop

Un **event loop** [25] è una architettura asincrona che permette di inserire gli eventi generati in una coda e di gestirli uno alla volta estraendoli dalla coda ed eseguendo il rispettivo *handler*. In particolare, viene preso l’evento più vecchio presente nella coda, viene eseguito il suo codice e torna ad estrarre un altro evento dalla coda, fino a quando la coda non è vuota. Questa architettura è stata pensata in quanto dal momento in cui viene avviata un’applicazione fino al momento in cui viene deallocata dalla memoria, l’applicazione non può prevedere quando dovranno essere elaborate delle operazioni di I/O, gestire il download di un file, gestire la UI ed altro ancora. Per di più, tutte queste

operazioni devono essere gestite in un singolo thread che non si deve mai bloccare. Per questo motivo viene impiegato l'event loop. Ogni qualvolta si verifica un'interruzione, che può essere invocata ad esempio dall'interazione con l'utente o dalla risposta di una socket, l'event loop continua ad elaborare gli eventi fino a quando la coda non termina. In quel caso, il sistema attende la notifica di nuovi eventi.

Tutte le API di alto livello e le funzionalità del linguaggio Dart per la programmazione asincrona, `Future`, `Stream`, `async` e `await` sono tutte basate sull'*event loop*. In particolare, le API sono solo dei modi per notificare all'*event loop* che è pronto del codice da eseguire. Gli `Stream`, i `Future` e `async` e `await` rappresentano le colonne portanti della programmazione asincrona in Dart.

4.4.3 Future

La problematica che può verificarsi durante l'esecuzione del codice nell'utilizzare un modello a singolo thread, è la possibilità di elaborare del *codice bloccante*, il quale causerebbe il blocco il meccanismo dell'event loop. La programmazione asincrona viene utilizzata per risolvere questa problematica. Il tipo di dato `Future<T>` è un'operazione asincrona che produce un risultato di tipo T [27]. Quando viene chiamato un metodo che restituisce un `Future` succede che:

1. Il metodo accoda le operazioni da eseguire;
2. Al termine delle operazioni, il `Future` viene completato con il risultato o con un errore.

Per poter utilizzare il risultato ottenuto da un metodo che restituisce un `Future<T>`, vengono utilizzate delle **callback**.

Se il metodo non deve restituire alcun tipo di risultato allora il tipo di ritorno della funzione sarà `Future<void>`.

4.4.4 Async e await

Nei linguaggi che supportano la programmazione asincrona, come ad esempio JavaScript, nella sintassi del linguaggio sono presenti le keyword `async` e `await`. `async` viene utilizzato nella *firma* di un metodo, per indicare che verrà restituito un `Future`, o in un metodo in cui all'interno del suo corpo viene chiamato un metodo che restituisce un `Future`. Mentre `await` viene utilizzato per indicare di attendere che il `Future` possa restituire il risultato. L'uso di `await` permette di avere del codice non bloccante, permettendo all'event loop di proseguire e di non rimanere in attesa che il metodo termini. Quando le

operazioni sono terminate, l'event loop riprenderà l'esecuzione dell'evento del `Future` che ha prodotto il risultato.

4.4.5 Stream

Gli `Stream` sono sequenze di eventi asincroni, che restituiscono un evento quando questo è terminato. Ad uno stato iniziale, lo `Stream` è semplicemente un oggetto che contiene degli eventi al suo interno. Uno `Stream` viene elaborato mediante l'uso di `await` o mediante la callback `listen()`. Quest'ultima, consente ad un *Listener* di ascoltare lo `Stream` e di estrarre gli eventi contenuti al suo interno.

In base alle esigenze, gli `Stream` si suddividono in:

1. **Single subscription stream:** gli eventi all'interno allo `Stream` vengono restituiti tutti e in ordine. Questo flusso però può essere ascoltato da un solo *Listener* un'unica volta. Il motivo è che un ulteriore ascolto può comportare la perdita degli eventi iniziali;
2. **Broadcast stream:** questa modalità è destinata ai singoli messaggi che possono essere gestiti uno alla volta, come ad esempio gli eventi generati dal mouse. Più *Listener* possono mettersi in ascolto sul medesimo `Stream` di questa categoria e ci possono essere ascolti successivi senza che questi comportino la perdita di eventi.

Ad uno `Stream` possono essere collegati più *Listener*. Quando è disponibile un nuovo evento, questo verrà notificato immediatamente a tutti gli ascoltatori collegati.

4.4.6 dynamic

Come è già stato spiegato precedentemente, Dart è un linguaggio fortemente tipizzato e supporta l'**inferenza** sui tipi. Quando però non viene dichiarato esplicitamente alcun tipo, viene implicitamente utilizzato il tipo **dynamic**. In Dart, ogni cosa è un oggetto che deriva dalla classe `Object`. L'uso di `dynamic` può significare che:

1. È necessario rappresentare un tipo di dato non rientrante tra quelli consentiti;
2. È necessario rappresentare un tipo di dato al di fuori dell'insieme dei tipi statici;

3. Si dichiara esplicitamente il dinamismo per quel dato a tempo di esecuzione.

Dal punto di vista dell'implementazione, questo significa che Dart non applica il controllo sul dato quando viene utilizzato *dynamic*. Ad esempio, se si dichiara un dato di tipo **A** e si prova a chiamare su quel dato un metodo non presente in **A**, Dart avvisa dell'errore. Se invece il tipo è stato dichiarato con **dynamic**, è possibile chiamare qualsiasi metodo, anche se questo genererà un errore a tempo di esecuzione, in quanto Dart smette di applicare a tale dato il meccanismo di controllo statico.

4.4.7 Gestore dei pacchetti

Pub è uno strumento per la gestione dei pacchetti per Dart [12]. È una raccolta di plugin *open source* che possono essere poi inclusi ed utilizzati nelle varie applicazioni. Sono presenti pacchetti sia per Flutter ma anche per Dart Web, Linux, macOS e Windows.

4.5 Familiarità della sintassi

La sintassi del linguaggio Dart è molto simile alla sintassi dei linguaggi Java, JavaScript e di altri linguaggi. A supporto di quanto appena affermato, si illustrano alcuni pezzi di codice scritti in diversi linguaggi che implementano il medesimo codice [13]:

1. Dart

```
1 class Segment {  
2     int links = 4;  
3     toString() => "I\u00b9have\u202a$links\u202alinks";  
4 }
```

2. Kotlin

```
1 class Segment {  
2     var links: Int = 4  
3     override fun toString() =>  
4         "I\u00b9have\u202a$links\u202alinks"  
5 }
```

3. Swift

```

1 class Segment: CustomStringConvertible {
2     var links: Int = 4
3     public var description: String { return
4         "I have \(links) links"
5     }
6 }
```

4. TypeScript

```

1 class Segment {
2     links: number = 4
3     public toString() = () : string => { return
4         'I have ${this.links} links'
5     }
6 }
```

4.6 Confronto con JavaScript

4.6.1 Introduzione a JavaScript

JavaScript è un linguaggio di *scripting* interpretato che permette di realizzare applicazioni sia *lato client* che *lato server*. È il linguaggio di scripting che viene maggiormente usato nel contesto del Web. JavaScript è un linguaggio *prototype-based* ed *event-driven*.

Contrariamente a quanto suggerisce il nome, JavaScript non ha alcuna correlazione con il linguaggio di programmazione *Java*, se non per la sintassi. Il nome è stato scelto per ragioni di marketing piuttosto che per la vicinanza dei due linguaggi.

JavaScript non produce delle applicazioni completamente *stand-alone* ma necessitano di essere eseguite in un determinato ambiente. Nel caso di applicazioni lato client, queste vengono eseguiti nel *browser*. I programmi lato server, come ad esempio quelli realizzati in *Node.js*, vengono eseguiti grazie ad uno specifico *engine*. Sia Google Chrome che Node.js utilizzano lo stesso engine: il *V8* sviluppato da Google.

L'uso principale di JavaScript è in ambito Web per la realizzazione di potenti applicazioni Web dinamiche.

4.6.2 Confronto

Class-based e prototype-based

Come prima differenza possiamo notare che Dart è un linguaggio *class-based* e non *prototype-based* come **JavaScript**. Quest'ultimo è un linguaggio che si basa sugli *oggetti* e non prevede l'utilizzo delle *classi*. Le classi sono dei costrutti utilizzati come modelli per la creazione degli oggetti. Ogni modello può contenere degli attributi e dei metodi. **Dart**, a differenza di JavaScript, essendo un linguaggio class-based, integra il concetto di *interfaccia*, di classe e di oggetti che vengono istanziati a partire da esse.

Curva di apprendimento

Supponendo di conoscere i concetti di base della programmazione, l'apprendimento di **JavaScript** risulta essere semplice. C'è un'ampia letteratura online ed offline riguardo questo linguaggio e tutte le sue peculiarità.

Sulla base delle medesime considerazioni iniziali, la curva di apprendimento di **Dart** risulta essere leggermente più alta rispetto a quella di JavaScript. La causa è dovuta alla scarsa diffusione del linguaggio e alla poca letteratura che è stata prodotta a riguardo. La documentazione più prolissa è quella fornita da Google. Le similitudini a livello sintattico con JavaScript e Java aiutano molto l'apprendimento a quei sviluppatori che provengono appunto da esperienze con tali linguaggi.

Semplicità d'uso del linguaggio

JavaScript è nato nel 1995 ed è conosciuto da molti programmati. È uno dei linguaggi più diffusi ed utilizzati nel mondo. È un linguaggio maturo, stabile ed è molto facile da utilizzare. Vi sono numerosi framework e librerie scritti in questo linguaggio.

Dart è un linguaggio relativamente recente per la maggior parte degli sviluppatori. Sebbene Google abbia fatto molti sforzi per documentare il linguaggio, è ancora difficile per gli sviluppatori trovare delle soluzioni a dei specifici problemi.

Compilazione ed interpretazione

Dart è un linguaggio compilato e questo permette di trovare la gran parte degli errori di programmazione a tempo di compilazione. Essendo inoltre *type safe*, è più sicuro dal punto di vista della tipizzazione rispetto a JavaScript, rendendo più semplice il debugging delle applicazioni. **JavaScript** è un linguaggio

debolmente tipizzato ed interpretato, caratteristica strutturale che degrada le prestazioni dell'applicazione. Inoltre Dart, può essere compilato AOT o JIT in base alle esigenze e ai contesti.

Web e mobile

JavaScript è sicuramente il linguaggio che viene per la maggior parte utilizzato durante lo sviluppo di applicazioni mobili e Web grazie a diversi framework. JavaScript è alla base di molti framework che permettono di realizzare Web app, PWA, applicazioni ibride e cross-platform.

Analogamente a JavaScript, **Dart** può essere utilizzato sia per lo sviluppo mobile che Web. Tuttavia il successo del linguaggio è dovuta al framework *Flutter*. Quindi, in questo periodo, Dart viene identificato più come un linguaggio adatto per costruire applicazioni multipiattaforma che per sviluppare applicazioni per il Web.

Capitolo 5

Flutter

5.1 Introduzione

Flutter è un framework sviluppato da Google che permette di realizzare delle applicazioni compilate nativamente per dispositivi mobili, Web e desktop a partire da un *singolo codice sorgente*, scritto in *Dart* [28]. L’obiettivo è consentire agli sviluppatori di offrire app ad alte prestazioni naturali su piattaforme diverse.

5.2 Struttura del framework

Il framework è strutturato a *strati* con l’obiettivo di poter realizzare di più, scrivendo meno codice. È stato pensato per facilitare la stesura e la leggibilità del codice.

Flutter si divide in:

1. **Framework:** è lo strato superficiale realizzato in Dart. Vengono messe a disposizione librerie ed astrazioni necessarie per lo sviluppo dell’applicazione. In particolare è possibile notare il livello dedicato ai **Widget**, uno dei più importanti durante lo sviluppo. Sulla base di questo *layer* è possibile definire altri due livelli: *Material* e *Cupertino*. Questi due strati implementano rispettivamente i componenti grafici secondo lo stile Android e iOS. È lo strato che viene per la maggior parte utilizzato dai programmatori durante lo sviluppo dell’applicazione. Gli sviluppatori possono creare dei nuovi componenti sulla base di quelli già forniti da Flutter o creare dei nuovi componenti ad-hoc;
2. **Engine:** realizzato in C++, permette di rendere le applicazioni prestanti ed efficienti. L’*engine* include molti componenti di basso livello, fonda-

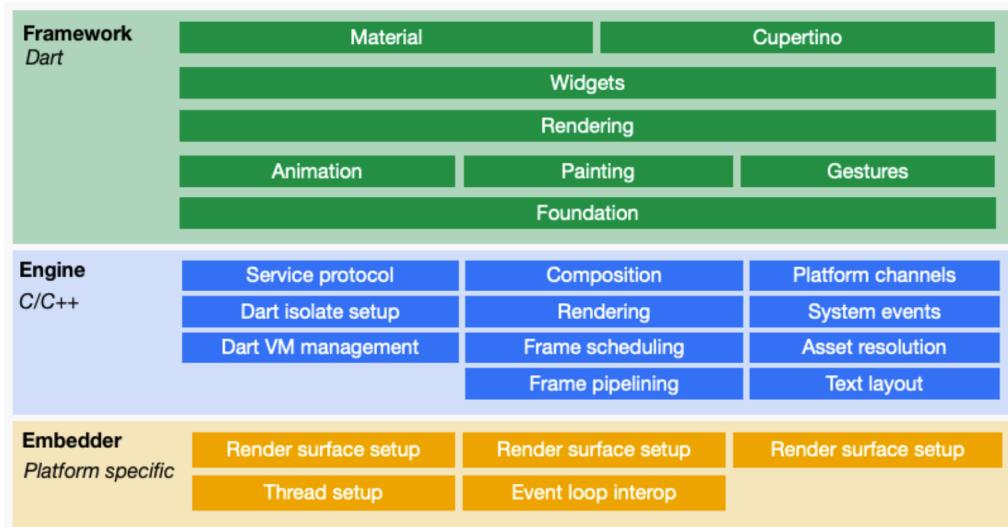


Figura 5.1: Architettura del framework [29].

mentali per il funzionamento del framework e per le sue operazioni di base. Tra questi troviamo il motore grafico *Skia*;

3. **Embedder:** è il livello più basso dell'architettura e rappresenta il cuore dell'engine di Flutter. In questo strato vengono definiti gli *embedder* specifici per le piattaforme, che hanno il compito di legare tra loro il rendering ai componenti della schermata nativa, alla gestione degli eventi di input e molto altro. Per implementare queste funzionalità, gli embedder interagiscono con l'engine (lo strato superiore) tramite delle API scritte in C/C++ di basso livello. Queste API non sono accessibili al programmatore, ma possono essere usate soltanto internamente da Flutter. Un altro componente importante di questo strato è la **shell** che ospita la *macchina virtuale* di Dart. La shell è specifica per ogni piattaforma e offre un accesso alle API native della piattaforma in questione. Le shell implementano del codice specifico per permettere la gestione degli eventi del ciclo di vita dell'app, in base al sistema operativo di interesse.

5.2.1 Skia

Skia [30] è un software che permette di effettuare il *rendering* dell'interfaccia grafica. Questa libreria è scritta in C++ e fornisce delle API comuni che funzionano su diverse piattaforme hardware e software (è utilizzata come motore grafico anche per *Google Chrome*).

5.3 Widget

"Everything is a widget" è il motto che più rappresenta Flutter [29]. I **Widget** sono i mattoni fondamentali dell’interfaccia utente dell’applicazione. I Widget sono tutti quegli oggetti che vengono visualizzati a schermo, con cui l’utente può interagire. I Widget, una volta dichiarati, sono *immutabili*. Altri framework separano i concetti di viste, controller di visualizzazione, layout ed altre proprietà. Flutter invece raggruppa tutti questi concetti in un’unica rappresentazione, infatti, un Widget può definire:

1. Una caratteristica del *layout* (ad esempio, il *padding*);
2. Un elemento *stilistico* (ad esempio, il *colore*);
3. Un elemento che va a costituire la *struttura* dell’interfaccia grafica (ad esempio, un *bottone*).

Tutti i Widget che vengono utilizzati per la realizzazione della UI costituiscono una gerarchia basata sulla **composizione**. Ogni Widget eredita le proprietà dell’oggetto genitore, non esiste un Widget che sia separato da questa gerarchia. Essendo i Widget immutabili, vi è uno specifico meccanismo per gestire gli eventi. Quando l’interfaccia grafica deve essere aggiornata a seguito di un’interazione con l’applicazione, il framework viene informato di *sostituire* un Widget della gerarchia (**Widget tree**) con un nuovo Widget. Il nuovo Widget può essere lo stesso di quello precedente ma con un nuovo contenuto oppure può essere un Widget differente. Quindi il framework opera un confronto tra il vecchio ed il nuovo Widget: se nota delle differenze tra i due, allora imposta il nuovo Widget, altrimenti lascia il Widget così com’è. Così facendo, si prevengono eventuali *reload* inutili della UI, aggiornando la UI soltanto quando è necessario.

La composizione dei Widget è molto **flessibile**: c’è molta libertà nel comporre i vari Widget per creare degli elementi grafici *personalizzati* e *riusabili* in parti differenti dell’applicazione.

È rilevante precisare che il team di sviluppo di Flutter ha voluto realizzare un software differente dagli altri framework cross-platform presenti nel mercato: con Flutter è possibile personalizzare il **singolo pixel** dell’applicazione. Ai programmatore e ai designer è concessa la piena libertà nella personalizzazione della UI. Tale caratteristica permette a questo framework di differenziarsi da tutti gli altri. In particolare, diversamente da Flutter, *React Native* (il principale competitor sviluppato da *Facebook*) utilizza gli elementi grafici nativi che offre il sistema operativo. Ovvero, il team di React Native ha sviluppato delle API per interagire con tali componenti. Invece, nel caso di Flutter, tutti i componenti grafici sono stati completamente ridisegnati, appunto intervenendo su

ogni singolo pixel. Il vantaggio di React Native è che se il sistema operativo apporta una modifica ad un componente grafico, il framework non deve essere aggiornato. Nel caso di Flutter, il team di sviluppo invece deve ridisegnare il componente e pubblicare un aggiornamento del framework. Tuttavia, Flutter lascia un'ampia libertà e flessibilità nella realizzazione degli elementi grafici, mentre React Native risulta essere più limitato da questo punto di vista, oltre ad avere la necessità di un *bridge* per interagire con i componenti grafici, causando l'introduzione di *overhead*.

5.3.1 Classificazione dei Widget

I Widget possono essere classificati in due categorie: possono essere **Stateful** o **Stateless**. Questa suddivisione nasce per motivi di efficienza. Come è stato accennato precedentemente, sulla base di un evento, il framework valuta se sostituire o meno un Widget. Tuttavia nell'interfaccia grafica è possibile trovare degli elementi che non vengono mai modificati durante tutto il ciclo di vita dell'applicazione: questi elementi possono essere bottoni, icone, testi ed altro ancora. Pertanto, questi Widget possono essere dichiarati **Stateless**, ovvero, non necessitano di possedere uno **stato** che può cambiare nel tempo. Quindi queste tipologie di Widget, una volta *renderizzate* dal framework, non vengono più aggiornate, ottenendo così delle performance migliori. I Widget vengono dichiarati **Stateful** quando l'utente può interagire con loro, cambiando il loro *stato*. Quando lo stato cambia, il framework si occupa di apportare le dovute modifiche, che possono essere, ad esempio, l'aggiornamento di un componente o l'aggiornamento di una variabile interna alla classe. Lo stato di un Widget è memorizzato in un oggetto di tipo **State<T>**, in modo da mantenere separato lo stato del Widget dalla sua rappresentazione grafica. Quando lo stato del Widget cambia, l'oggetto **State<T>** chiama il metodo **setState()**, eseguendo il suo contenuto. Tuttavia, l'utilizzo del metodo **setState()** comporta ad introdurre della business logic nella parte dedicata alla realizzazione dell'interfaccia grafica. Pertanto, per ovviare a questa problematica, viene sfruttata la *programmazione reattiva*, presentata nel capitolo relativo all'introduzione al linguaggio Dart.

Di seguito, si illustra qual è la struttura di base per creare un **Stateful** Widget e un **Stateless** Widget:

1. **Stateful**:

```
1 class MyButton extends StatefulWidget {  
2     @override  
3     _MyButtonState createState() => _MyButtonState();  
4 }
```

```
5
6 class _MyButtonState extends State<MyButton> {
7     @override
8     Widget build() {
9         // Build your own Widget
10        // by composing other Widgets
11        return ...;
12    }
13 }
```

2. Stateless:

```
1 class MyIcon extends StatelessWidget {
2     @override
3     Widget build() {
4         // Build your own Widget
5         // by composing other Widgets
6         return ...;
7     }
8 }
```

5.3.2 Costruzione dei Widget

Per costruire il Widget, il framework chiama il metodo `build()`, presente sia nella classe `Stateful` che nella classe `Stateless`. Questo metodo restituisce un albero di Widget (*Widget tree*). Se ci poniamo nella radice di questo albero, verrà restituito tutto l'albero dei Widget dell'applicazione; se ci poniamo in un nodo interno di questo albero, verrà restituito un sottoalbero del Widget tree. L'albero rappresenta l'interfaccia utente, realizzata per composizione con i vari Widget. A partire dalla radice, il framework chiama **ricorsivamente** il metodo `build()` per ciascun Widget, fino a quando non si arriva alle foglie dell'albero, ovvero, si arriva a costruire dei Widget elementari, che non sono più costituiti da altri Widget. Un esempio semplificato di un Widget tree è illustrato nella Figura 5.2.

La creazione di un Widget dovrebbe essere *priva di effetti collaterali*. A seguito di un cambiamento dello stato dell'applicazione, il framework confronta la struttura costruita precedentemente con la struttura costruita in quel preciso istante, per determinare quali modifiche devono essere apportate all'interfaccia utente. Questo confronto automatizzato risulta essere efficace, consentendo la realizzazione di applicazioni interattive e prestanti.

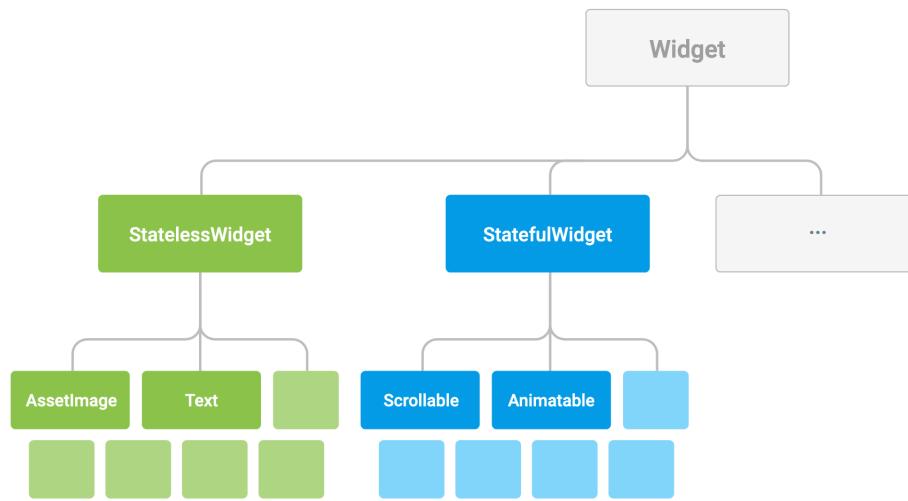


Figura 5.2: Un esempio semplificato di un *Widget tree* [29].

L'utilizzo del metodo `build()` permette di focalizzarsi sul come costruire un Widget per composizione, tralasciando la complessità dell'aggiornamento della UI da uno stato ad un altro.

5.4 State Management

L'utilizzo della programmazione reattiva porta a gestire in maniera differente i dati che devono essere comunicati tra i vari componenti. Nella comunità di Flutter, questo è un tema sempre molto discusso, da cui fuoriescono diverse soluzioni e possibili architetture per gestire in modo efficiente la *gestione dello stato* (*state management*). Una qualsiasi modifica, come la ricezione di un messaggio o l'interazione dell'utente con l'app, può essere intesa come una variazione dello stato dell'applicazione. Questo aspetto è di fondamentale importanza in quanto è una caratteristica intrinseca del framework: pertanto, questo tema deve essere affrontato ampiamente, andando ad analizzare la documentazione fornita dalla comunità di Flutter. Se lo *state management* non viene gestito correttamente durante lo sviluppo dell'applicazione, non solo si possono ottenere delle inefficienze dal punto di vista delle prestazioni, ma anche delle vere e proprie problematiche nella comunicazione dei cambiamenti di stato, che possono causare, per esempio, la mancata ricezione di alcuni dati. Nel contesto d'uso in cui l'applicazione deve far parte, è necessario porre molta

attenzione nella ricerca e nello sviluppo di un'architettura che riesca a gestire correttamente lo *state management*.

Di seguito vengono elencate le diverse soluzioni ed architetture proposte fino ad ora, sia dal team di Flutter che dalla comunità.

5.4.1 setState

Questo è l'approccio di gestione dello stato più semplice presente in Flutter. Tramite la funzione `setState()` è possibile indicare nel suo corpo tutte le operazioni che modificano il valore di un qualche oggetto dell'applicazione. Questo oggetto è condiviso da più Widget e pertanto la modifica di tale deve avvenire all'interno del metodo `setState()` in modo da forzare il framework ad aggiornare i rispettivi Widget.

I vantaggi sono la facilità di utilizzo e la facilità di comprensione del meccanismo. Tuttavia, quando l'applicazione cresce di complessità, questo approccio risulta essere molto limitante. Gli altri svantaggi sono:

1. Non è possibile gestire degli stati che devono persistere tra le sessioni;
2. L'utilizzo di questo approccio rende il processo di manutenzione molto più laborioso e complicato in quanto lo stato viene sparso in varie zone dell'applicazione;
3. Si inserisce nel codice dedicato all'interfaccia grafica, mescolando UI e business logic;

Questo approccio è consigliato per la realizzazione di piccole applicazioni e che in futuro non verranno ampliate.

5.4.2 InheritedWidget

`InheritedWidget` è un Widget che non mostra nulla a livello grafico ma permette di salvare dei dati al suo interno e di propagarli nel Widget tree a lui sottostante. I Widget dell'albero possono accedere ai dati del `InheritedWidget` tramite il metodo `InheritedWidget.of(context)`. Questo approccio è alla base del pattern Provider e viene ampiamente utilizzato anche per la creazione di molti componenti di default di Flutter.

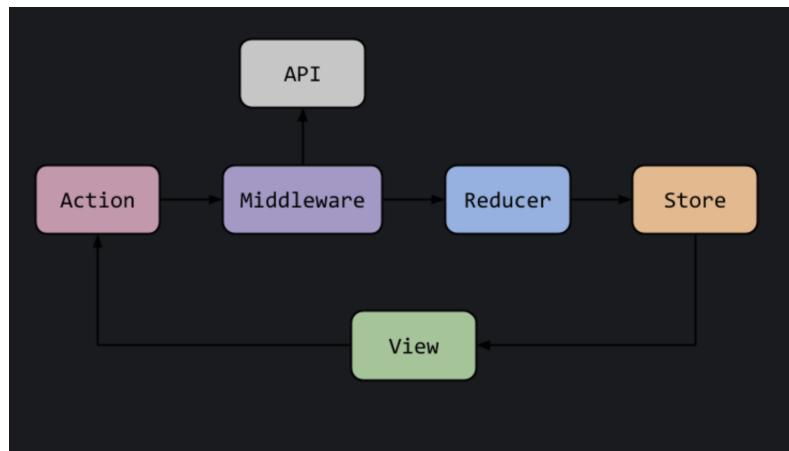


Figura 5.3: Il pattern architetturale Redux.

5.4.3 Scoped Model

Lo **Scoped Model** è un approccio basato su `InheritedWidget` e offre un modo, leggermente migliore, per accedere, aggiornare e mutare lo stato. Permette di passare facilmente un `Model` di dati da un Widget genitore ai suoi discendenti. Inoltre, ricostruisce anche tutti i *child* che utilizzano tale modello, nel momento in cui questo viene aggiornato. Così facendo, potrebbero sorgere delle problematiche relative alle prestazioni, a seconda di quanti `ScopedModelDescendants` ha un modello. I `ScopedModelDescendants` vengono ricostruiti ogni volta è presente un nuovo aggiornamento dello stato. Questo problema può essere risolto decomponendo lo `ScopedModel` in più modelli, in modo da ottenere delle dipendenze più precise.

Specificando il flag `rebuildOnChange` su `false`, viene risolto il problema, in quanto si va a specificare quale Widget necessita di essere ricostruito all'aggiornamento dello stato.

5.4.4 Redux

Redux è un'architettura nella quale è presente un unico flusso unidirezionale dei dati che semplifica lo sviluppo, la manutenzione e il test delle applicazioni [36].

È una nota libreria JavaScript ampiamente utilizzata in React. Grazie al suo successo, è stato effettuato un *porting* della libreria per Flutter.

5.4.5 Provider e BLoC

Questi due pattern architetturali verranno descritti ampiamente nel capitolo successivo, in quanto sono i due principali metodi utilizzati per la realizzazione dell'architettura complessiva dell'applicazione.

5.5 Hot Reload

Questa rappresenta una delle funzionalità più apprezzate dagli sviluppatori Flutter. Google tentò più volte di implementare *Instant Run* per lo sviluppo di applicazioni per Android, ma con scarsi successi.

Nella fase di sviluppo di un'applicazione, Flutter utilizza il compilatore *JIT*. L'*hot reload* permette di ricaricare e di continuare l'esecuzione del codice, in pochi decimi di secondo. Lo stato dell'applicazione rimane inalterato tra un *reload* e l'altro quando possibile: se avviene un aggiornamento dal punto di vista grafico, allora lo stato rimane inalterato; se invece l'aggiornamento apportato riguarda una modifica fatta a livello di business logic, allora è possibile che lo stato venga alterato. In quel caso sarà necessario ricompilare l'applicazione per ripristinare uno stato consistente.

Questa funzionalità accelera i tempi di sviluppo dell'interfaccia grafica in quanto vengono eliminati tutti i tempi di attesa, dal momento in cui viene avviata la nuova compilazione fino al suo termine. Pur sembrando irrisoni, la somma di tutti questi tempi morti comporta a risparmiare molto tempo durante l'intero processo di sviluppo e anche durante la fase di *debugging* [31].

5.6 Fuchsia

Lo sviluppo di questo framework fa parte di un contesto molto più grande di quello attuale. Negli ultimi anni Google ha iniziato a sviluppare un nuovo sistema operativo denominato **Fuchsia** [32] [33]. A differenza dei precedenti sistemi operativi, come Android e Chrome OS, che si basano su kernel *Linux*, Fuchsia si basa su un nuovo *microkernel* denominato **Zircon**, derivato da *Little Kernel (LK)*, progettato per funzionare su qualsiasi dispositivo. In particolare è stato progettato per funzionare su smartphone e computer moderni, con processori veloci, quantità molto alte di memoria RAM e con periferiche arbitrarie per il calcolo computazionale.

Flutter diventerà il principale framework di sviluppo per le applicazioni di Fuchsia. Per essere più precisi, l'SDK di Flutter sarà nativamente supportato dal futuro sistema operativo (si veda Figura 5.3).

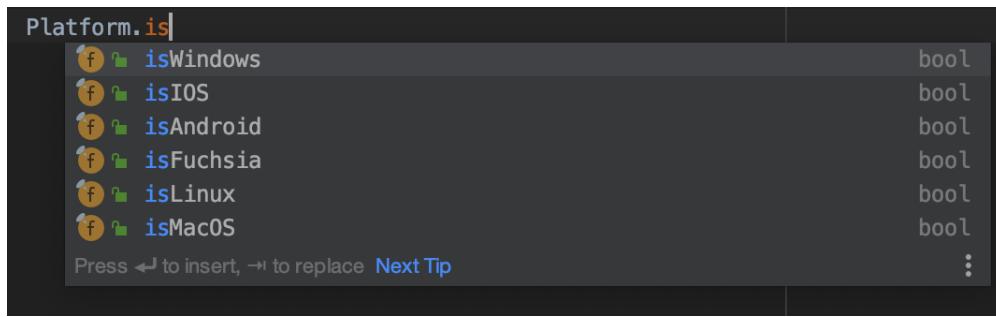


Figura 5.4: Già da diverse versioni dell’SDK di Flutter, sono presenti alcuni suggerimenti per il futuro supporto di Fuchsia (la versione di Flutter in questione è la `1.12.13+hotfix.9` del canale stabile e la versione di Dart è la `2.7.2`).

Una delle proprietà più rilevanti di Fuchsia che è possibile notare è la capacità di adattare l’interfaccia grafica a seconda del dispositivo in cui viene eseguito, tutto in maniera estremamente semplice. L’obiettivo di Google è un obiettivo molto ambizioso, intrapreso recentemente da diverse aziende tecnologiche come Microsoft, Samsung ed Apple. In particolare:

1. Microsoft ha tentato l’unificazione con la *Universal Windows Platform* (UWP), progetto naufragato a causa della rimozione del supporto a *Windows Mobile*. L’obiettivo era quello di realizzare applicazioni universali che potessero essere eseguite su Windows 10, Windows 10 Mobile e Xbox One;
2. Samsung ha avviato *Samsun DeX*: collegando il proprio smartphone ad un monitor e ad una tastiera è possibile utilizzarlo tramite un’interfaccia che si adatta alle dimensioni di un monitor;
3. Apple ha presentato *Apple Catalyst*, un software che permette di creare applicazioni che possono essere eseguite sia su Mac che iPad.

Osservando tutte queste iniziative avviate dalle più grandi aziende tecnologiche al mondo, è facile intuire che il futuro sarà l’unificazione dei dispositivi mobili, tablet e desktop dal punto di vista delle applicazioni.

Nell’ultimo periodo, Fuchsia è stato testato su un insieme di dispositivi molto differenti tra loro: è stato testato su uno smartphone Huawei, su un Google Nest Hub [34], Chromebook ed altri.

5.7 Ciclo di vita delle applicazioni

Il *ciclo di vita* di un'applicazione Flutter è leggermente differente sia dal ciclo di vita di un'app Android che iOS. Il motivo è che essendo un framework cross-platform, deve riuscire ad interagire con entrambi i sistemi operativi. Pertanto, risulta necessario avere un ciclo di vita comune a tutti i sistemi operativi. Gli eventi nel ciclo di vita di un'app in Flutter sono:

1. **inactive**: l'applicazione è inattiva e non riceve input dall'utente. Questo evento funziona solo per iOS in quanto non esiste un evento corrispondente in Android;
2. **paused**: l'applicazione non è momentaneamente visibile all'utente, non risponde ai suoi input ed è eseguita in background (corrisponde all'evento `onPause()` di Android);
3. **resumed**: l'applicazione è visibile e risponde all'input dell'utente (corrisponde all'evento `onPostResume()` di Android);
4. **suspending**: l'applicazione è momentaneamente sospesa. Equivale all'evento `onStop()` di Android. Non esiste un evento corrispondente su iOS.

Flutter non mette a disposizione molti metodi per il controllo del ciclo di vita. Il motivo è che ci sono poche argomentazioni valide per osservare il ciclo di vita dal punto di vista di Flutter e quindi tali metodi vengono nascosti allo sviluppatore. Questi metodi vengono eseguiti internamente dal framework. Nel caso fosse strettamente necessario dover osservare il ciclo di vita per poter acquisire o rilasciare delle risorse, è preferibile farlo nativamente.

5.8 Pacchetti

In precedenza è stato introdotto il *Pub* di Dart, ovvero, una raccolta di pacchetti open source disponibili anche per Flutter. In un'applicazione Flutter, questi pacchetti devono essere annotati dal programmatore su un particolare file: **pubspec.yaml**. Questo file è presente in qualsiasi progetto Flutter, fin dal momento della sua creazione. In questo file sono contenute:

1. Tutte le dipendenze dell'applicazione;
2. I percorsi degli *asset* (ad esempio: icone, immagini, file JSON, font);

Per importare nel progetto un pacchetto, è sufficiente scrivere sotto la voce `dependencies` il nome e la versione del pacchetto (ad esempio, `url_launcher: ^5.4.7[35]`). Successivamente sarà necessario lanciare il comando `flutter pub get` per poter scaricare il pacchetto desiderato. Non è necessario effettuare il download manuale, è sufficiente indicare il nome e la versione del plugin e il sistema provvederà a scaricarlo e ad integrarlo automaticamente nel progetto.

5.9 Vantaggi e svantaggi di Flutter

In conclusione ai capitoli relativi a Dart e Flutter, si elencano alcuni vantaggi e svantaggi dell'utilizzo congiunto di questo linguaggio e di questo framework, sulla base anche dell'esperienza ottenuta durante lo sviluppo del progetto.

I vantaggi di utilizzare Flutter come framework di sviluppo sono:

1. **Hot Reload:** funzionalità che consente di visualizzare le modifiche apportate al codice quasi in tempo reale, senza necessità di riavviare l'applicazione. Le modifiche apportate vengono inserite nell'applicazione in esecuzione e Flutter ricostruisce automaticamente il Widget tree in modo che tali modifiche vengano visualizzate in tempo reale;
2. **Prestazioni:** Flutter non utilizza alcun *bridge* in JavaScript per visualizzare in modo reattivo l'applicazione, come nel caso di React Native. L'applicazione quindi risulterà più prestante e veloce;
3. **Programmazione reattiva:** permette di non fare uso del metodo `setState()`, migliorando la *riusabilità del codice*, in quanto rende possibile la separazione della business logic dall'interfaccia grafica. Inoltre, migliora le prestazioni dell'applicazione, poiché evita di ridisegnare un Widget e l'intero Widget tree ad ogni modifica apportata. Le modifiche vengono propagate mediante gli `Stream`.

Di seguito, si elencano gli svantaggi dell'utilizzo di Flutter e, più in generale per alcuni punti, dei framework cross-paltform:

1. **Dart:** diversamente da altri framework che utilizzano dei linguaggi già affermati nel mondo dello sviluppo software (JavaScript nel caso di React Native e C# nel caso di Xamarin), imparare un nuovo linguaggio comporta comunque il dover superare una certa *curva di apprendimento*, seppur bassa. La curva di apprendimento è bassa ed è dovuta alla vicinanza di Dart a linguaggi come JavaScript e Java, che semplificano l'apprendimento di tale linguaggio.

2. **Multipiattaforma:** con un'accezione negativa, non è possibile sviluppare delle applicazioni che devono implementare delle funzionalità che sfruttano dei particolari servizi offerti dal sistema operativo;
3. **Dimensioni dell'applicazione:** non possono essere create delle applicazioni di dimensioni inferiori ai 4 MB. Il motivo riguarda la natura stessa di Flutter: nel codice dell'applicazione devono essere inclusi tutti i Widget necessari, in quanto il framework non utilizza i componenti grafici offerti dal sistema operativo;
4. **Programmazione reattiva:** non sempre è possibile utilizzare la programmazione reattiva, spesso rende difficoltosa la leggibilità del codice, specialmente se la struttura dei Widget implementati è complessa.

5.10 Confronto con React Native

5.10.1 Introduzione a React Native

React Native è un framework JavaScript per lo sviluppo di applicazioni mobili che permette di effettuare il rendering nativo sia per iOS che per Android. Questo framework si basa su *React*, la libreria JavaScript di Facebook per la creazione di interfacce utente Web. Il motivo di questa scelta risulta essere molto astuta: Facebook può coinvolgere gli sviluppatori, che già conoscono la versione Web, verso lo sviluppo di applicazioni multipiattaforma, senza dover far imparare agli sviluppatori delle nuove logiche ed una nuova architettura per realizzare applicazioni mobile. Gli sviluppatori dovranno fare un minimo sforzo per capire in che modo il framework interagisce con i dispositivi.

Analogamente a *React.js* (la libreria Web), le applicazioni React Native sono scritte usando una combinazione di JavaScript e XML, noto come *JSX*. Quindi, il **bridge** nativo di React invoca le API di rendering native in Objective-C (per iOS) e in Java (per Android). Pertanto, l'applicazione eseguirà il rendering utilizzando i componenti grafici già presenti nel sistema operativo, senza effettuare visualizzazioni Web. React Native espone anche le interfacce JavaScript per le API della piattaforma, quindi le applicazioni React Native possono accedere a funzionalità del dispositivo come la videocamera del telefono o la posizione dell'utente.

5.10.2 Confronto

Linguaggio di programmazione utilizzato

React Native utilizza JavaScript per creare le applicazioni. JavaScript è un linguaggio molto popolare nello sviluppo di applicazioni Web. Con React Native, gli sviluppatori Web possono creare delle applicazioni mobili, aggiungendo poche conoscenze basilari.

Flutter utilizza *Dart* come linguaggio per scrivere le applicazioni. È stato introdotto da Google e il suo uso è limitato a questo framework. La sintassi Dart è di facile comprensione per gli sviluppatori JavaScript o Java, in quanto supporta la maggior parte dei concetti orientati agli oggetti. Inoltre il linguaggio è provvisto di una buona documentazione.

JavaScript è ampiamente utilizzato dalla maggior parte degli sviluppatori Web: di conseguenza è facile adottare il framework React Native. Nonostante Dart abbia delle ottima funzionalità e caratteristiche, è usato raramente ed è meno conosciuto nella comunità degli sviluppatori.

Architettura

L'architettura di **React Native** si basa fortemente sull'architettura dell'ambiente di runtime Javascript, ovvero, il **bridge**. React Native lo utilizza per comunicare con i moduli nativi del dispositivo. Questo intermediario tra l'applicazione e il dispositivo comporta ad un degrado delle prestazioni e ad una perdita di efficienza.

In **Flutter**, diversamente, tutti i componenti sono integrati nell'applicazione: questo comporta una maggiore dimensione dell'applicazione una volta che questa deve essere installata sul dispositivo. Tuttavia, permette di interagire in modo diretto con i moduli nativi, senza aver la necessità di un *bridge*. In questo modo, le prestazioni di un'app realizzata con questo framework si avvicinano a quelle di un'applicazione nativa.

Installazione

Il framework **React Native** può essere installato utilizzando **Node Package Manager (NPM)**. Per gli sviluppatori che già conoscono JavaScript, l'installazione di React Native risulta essere semplice e familiare. Questo gestore dei pacchetti può installare i pacchetti localmente o globalmente. Sta allo sviluppatore capire dove si trovano esattamente i binari.

Flutter può essere installato scaricando il binario, per una piattaforma specifica, da *Github*. L'installazione di Flutter richiede diversi passaggi da svolgere per poterlo integrare nel sistema. Il meccanismo usato da React Native

risulta essere molto più intuitivo, semplice e diretto, senza la necessità di dover scaricare manualmente i binari.

Tuttavia entrambi mancano di un'installazione *one-liner* con gestori di pacchetti nativi per un sistema operativo specifico.

Setup e configurazione del progetto

La guida relativa a **React Native** assume come base di partenza che lo sviluppatore abbia già tutte le conoscenze necessarie per lo sviluppo per iOS ed Android. Vengono fornite poche informazioni per il setup corretto dell'IDE. La documentazione passa direttamente alla fase di creazione del progetto.

La guida introduttiva per **Flutter**, invece, contiene informazioni dettagliate riguardo la configurazione dell'IDE e l'introduzione alle piattaforme iOS ed Android. Inoltre, Flutter ha uno strumento da linea di comando chiamato **flutter doctor**, che può guidare gli sviluppatori attraverso l'installazione. Questo tool controlla quali pacchetti sono installati sul computer locale e quali strumenti devono essere configurati.

Componenti grafici e API di sviluppo

React Native di base fornisce soltanto il rendering dell'interfaccia utente e le API per accedere ai moduli del dispositivo. Per accedere alla maggior parte dei moduli, React Native deve fare affidamento su librerie di terze parti. Purtroppo, React Native dipende troppo dalle librerie di terze parti.

Flutter fornisce molti componenti di rendering dell'interfaccia utente, l'accesso all'API del dispositivo, una gestione dello stato (*state management*) e molte librerie. Questo ricco set di componenti elimina la necessità di utilizzare librerie di terze parti. Tutto il necessario per sviluppare un'applicazione in Flutter viene fornito direttamente dal framework, una volta installato. In Flutter sono stati implementati manualmente, dal team di sviluppo, i Widget per Material e Cupertino, che consentono agli sviluppatori di eseguire facilmente il rendering dell'interfaccia sia su Android che iOS.

Supporto della community

React Native è stato rilasciato nel 2015 e da allora ha ottenuto molto successo. Negli anni molti sviluppatori si sono avvicinati a questo framework ed ora React Native può vantare un solido seguito di programmatore.

Flutter ha attirato molta attenzione dal momento in cui Google lo ha promosso nella conferenza *I/O* nel 2017. La comunità Flutter sta crescendo rapidamente in questo ultimo periodo; tuttavia, gli sviluppatori non dispongono ancora di risorse sufficienti per risolvere problemi specifici.

Testing

Essendo **React Native** un framework JavaScript, Internet pullula di framework per effettuare test a livello di unità, come ad esempio *Jest*. Tuttavia, quando si tratta di test a livello di integrazione o di interfaccia utente, React Native non offre alcun supporto ufficiale.

Flutter offre un ricco set di funzionalità per testare le applicazioni a livello di unità e di integrazione. Flutter ha un'ottima documentazione sul testing delle applicazioni, oltre ad avere un'apprezzata funzionalità per testare i Widget: è possibile eseguire dei test per l'interfaccia utente ed eseguirli alla velocità dei test unitari.

Capitolo 6

Sviluppo dell'applicazione

In questo capitolo verranno trattati gli argomenti relativi alla progettazione e allo sviluppo concreto dell'applicazione. Si procederà con l'introduzione dei *requisiti* che l'applicazione deve soddisfare, per poi esporre le scelte progettuali e le motivazioni che hanno portato alla realizzazione dell'architettura finale. Le varie architetture verranno introdotte prima da un punto di vista teorico, per poi affrontare una spiegazione più tecnica, strettamente legata all'effettiva implementazione.

6.1 Requisiti

L'applicazione deve essere in grado di interrogare un server tramite una *socket*, la quale invierà come risposta il set di dati, rilevati dai sensori installati sulla barca. L'applicazione deve svolgere un *polling* della socket, periodicamente per un tempo prefissato. Questi dati dovranno essere organizzati e visualizzati su più schermate, in modo da facilitare l'utente nel monitorare i dati di cui necessita.

Rispetto all'applicazione Android sviluppata precedentemente, si vuole ridisegnare l'interfaccia grafica in modo da rendere la visualizzazione dei dati il più efficace possibile.

Si vuole inoltre disporre di alcune funzionalità come la possibilità che l'utente possa decidere ogni quanti secondi interrogare il server e che questo non sia un parametro fisso integrato nel codice.

In conclusione, si vuole rendere l'applicazione disponibile sia per gli utenti con dispositivi Android che iOS. Pertanto la scelta di un framework cross-platform risulta essere adatta nel rispettare questa particolare richiesta.

6.2 Architetture utilizzate

6.2.1 Introduzione

L'architettura rappresenta l'elemento fondante di tutta l'applicazione. Essa permette di definire le logiche e i comportamenti che avrà l'applicazione al termine dello sviluppo e definisce anche in che modo deve avvenire la cooperazione tra i vari componenti dell'applicazione. Un altro fattore importante da tenere in considerazione è il contesto in cui l'applicazione verrà utilizzata. Quindi, è necessario quindi scegliere attentamente l'architettura corretta sulla base dei requisiti e del risultato finale che si vuole ottenere. Il processo di scelta dell'architettura richiede una certa esperienza nella programmazione, nel conoscere il framework con il quale si creerà l'applicazione ed una conoscenza delle nozioni di *ingegneria del software*.

Nella scelta dell'architettura influiscono anche proprietà come le *prestazioni* che si vogliono ottenere e la *flessibilità* dell'architettura stessa, necessaria per poter supportare l'implementazione di nuove funzionalità in futuro.

In conclusione, la scelta dell'architettura è una fase particolarmente delicata ed importante in quanto determinerà il successo nel raggiungere i requisiti richiesti o il mancato raggiungimento di essi.

6.2.2 Scelta dell'architettura

L'architettura scelta farà un largo uso della *programmazione reattiva*, caratteristica peculiare del linguaggio del framework.

Un principale svantaggio dell'utilizzo di Flutter come framework di sviluppo è il suo recente rilascio nel mercato. Essendo un framework relativamente nuovo rispetto agli altri, la comunità di Flutter ha proposto diverse architetture e pattern per implementare le applicazioni. Tuttavia, Google non ha mai fornito un supporto alla realizzazione di un qualche pattern ideale nelle sue guide relative al framework. Nella sua documentazione, il team di Flutter ha soltanto racchiuso in una pagina una serie di link alle risorse che illustrano alcuni pattern, spiegati da chi li hanno ideati. Pertanto, la ricerca e l'analisi di dell'architettura adatta è un processo più laborioso rispetto ad altri framework già affermati nel mercato.

6.2.3 Provider

L'introduzione di questo pattern è di fondamentale importanza in quanto rappresenta il fulcro di tutto il pattern architetturale finale dell'applicazione.

Una delle possibili definizioni per il pattern **Provider** può essere la seguente: *"Il pattern Provider condivide i valori tra Widget tree e i Widget facenti parte dell'albero vengono ricostruiti in base ai cambiamenti di stato avvenuti nell'applicazione"*. Esaminando questa definizione è possibile affermare che:

1. Il Provider è un Widget;
2. Le dipendenze di un Provider sono dei Widget tree;
3. Il Provider si occupa di aggiornare i Widget nel momento in cui rileva un cambiamento di stato;

Da queste considerazioni, si può capire che l'aggiornamento debba avvenire in modo *asincrono* e debba essere *reattivo*. In particolare, i Widget che non cambiano con la generazione di un nuovo stato, non necessitano di dover rimanere in ascolto di un nuovo stato; i Widget che cambiano nel momento in cui viene rilevata una variazione dello stato dell'applicazione, devono rimanere in ascolto non appena un nuovo stato viene creato. Pertanto, in quest'ultima casistica, verranno utilizzati **Future** e **Stream**.

Quando lo stato cambia, questo viene fornito al Provider piuttosto che a ciascun discendente (Widget). Per fare ciò il Provider utilizza internamente un **InheritedWidget**, un oggetto che permette di propagare, in modo efficiente, i cambiamenti lungo il Widget tree. La gestione dell'aggiornamento dei Widget tree rappresenta uno dei principali problemi dello sviluppo dell'applicazione, in quanto questo fattore può essere uno dei principali responsabili del degrado delle prestazioni.

Il Provider rappresenta, sostanzialmente, un mezzo per condividere un valore per i Widget che ne necessitano. Con le successive spiegazioni, si andrà ad apprezzare la *flessibilità* di questo pattern.

Strutturazione del Provider

Ricordando che questo pattern è un Widget e pertanto si inserirà nel Widget tree complessivo dell'applicazione, ogni Provider ha un *figlio* (il parametro **child**) che rappresenta la radice di un certo Widget tree (sottoalbero del Widget tree dell'app). Il Provider rende il valore disponibile per tutti i Widget a loro sottostanti. Ogni Widget decide individualmente se utilizzare o meno il valore ricevuto.

Non ci sono restrizioni su ciò che può essere un *valore*. Può essere un *servizio* (ad esempio un API), un *modello* o uno *stato*. Un modello potrebbe essere un oggetto che rappresenta una certa entità, come ad esempio un messaggio o un utente. Uno stato potrebbe essere il valore di un campo, la quantità di dati finora caricati o lo stato di riproduzione di un flusso di dati audiovisivo.

La scelta di non prendere i nuovi valori da delle variabili globali visibili a tutti i Widget che ne necessitano, è dovuta alla rigida gerarchia dei Widget tree, la quale impedisce che si verifichino delle *dipendenze circolari*. Un'altra ragione, riguarda il passaggio di un valore attraverso più costruttori per farlo raggiungere ad uno specifico Widget dell'albero. Questa tecnica rende il codice complesso e verboso, pertanto vengono utilizzate le tecniche di *dependency injection*.

Il Provider viene creato prima di istanziare tutti i Widget dipendenti che esso contiene (Figura 6.2). Costruire per primo il Provider, permette di avere già a disposizione dei valori per i Widget discendenti che **non sono ancora stati costruiti**. Quando i Widget dipendenti vengono successivamente creati, essi hanno accesso a tali valori e possono effettuare le loro modifiche grafiche in base ai valori ricevuti.

Recuperare ed utilizzare i valori dal Provider

I Widget possono riferirsi ai valori contenuti nel Provider con il metodo statico `Provider.of<T>(context)`. Sulla base di questo metodo è possibile individuare un principale svantaggio di questo pattern: per poter recuperare i valori all'interno del Provider, è necessario che il metodo venga chiamato all'interno di un Widget (solitamente all'interno del metodo `build()`) o che comunque gli venga passato un `BuildContext context`. Questo svantaggio rappresenta una restrizione in quanto non è possibile utilizzare il Provider in classi dove non vi sia un *contesto*. Ad esempio, il Provider non può essere utilizzato nelle classi che vanno a costituire la business logic.

Quando si va a definire e ad utilizzare un Provider, è necessario specificare il tipo di dato che il Provider deve rendere disponibile ai suoi Widget discendenti. Questo significa che il Provider è una classe *parametrica*: questo gli garantisce una certa flessibilità d'uso. Pertanto un Provider può rendere disponibile un valore di un qualsiasi tipo di dato.

Di seguito si introducono i passi che vengono svolti per poter istanziare il pattern:

1. Come prima cosa Flutter si occupa della creazione del Provider. La costruzione del Provider implica la costruzione del figlio (`child`) e la costruzione del figlio implica a sua volta la costruzione di tutti i Widget

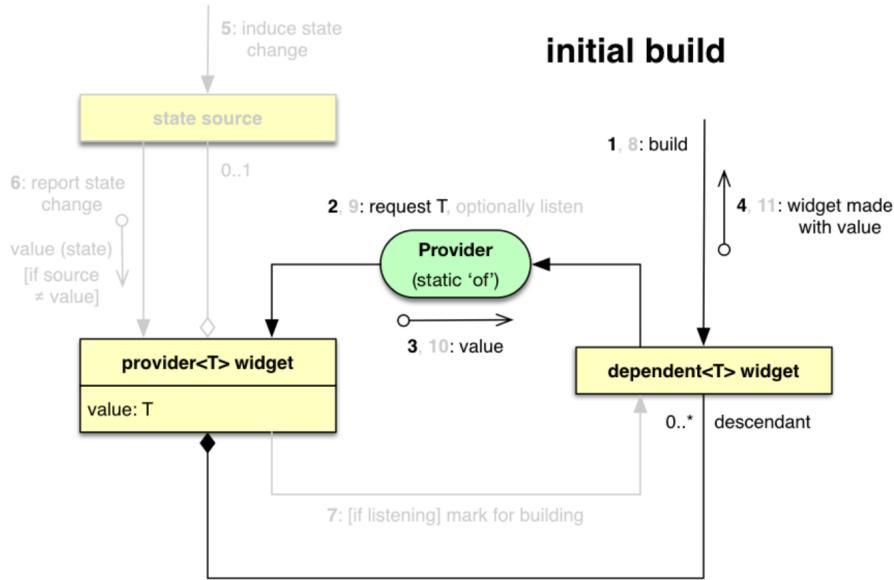


Figura 6.1: Inizializzazione del Provider [37].

in esso contenuti. Il framework chiama il metodo `build()` sul figlio per costruire il Widget che deve essere renderizzato. Questo metodo viene chiamato per tutti i Widget discendenti;

2. Durante la compilazione (in particolare alla prima invocazione del metodo `build()`), uno o più Widget discendenti, effettuano una chiamata al metodo `Provider.of <T>()` per richiedere un valore. Questa chiamata viene fatta immediatamente dai Widget sottostanti, in quanto non possono essere creati senza un valore iniziale dello stato;
3. Dopo che l'albero dei Widget è stato creato per la prima volta sulla base di uno stato iniziale, il Provider è in grado di ricevere i cambiamenti di stato. Al momento della ricostruzione dei Widget, il Provider crea un nuovo `InheritedWidget` dalla stessa istanza del figlio utilizzata inizialmente. `InheritedWidget` identifica i *child* che stanno ascoltando i cambiamenti di stato e contrassegna soltanto i Widget che devono essere ricostruiti (Figura 6.2).

Nel momento in cui il Provider riceve un nuovo dato e lo deve rendere disponibile ai suoi discendenti, esso causerebbe il rendering di tutti i Widget sottostanti, anche di quelli che non sono influenzati dal cambiamento di stato.

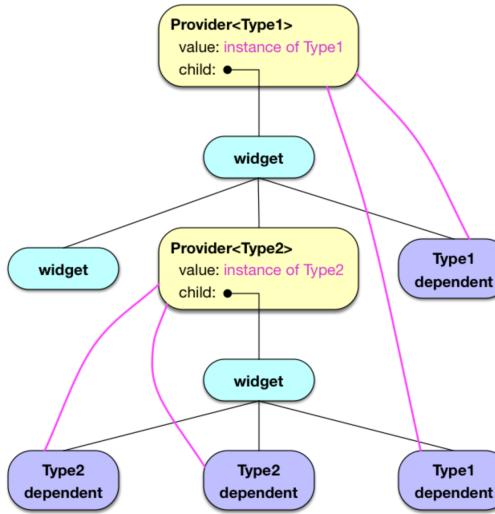


Figura 6.2: Aggiornamento dei Widget a fronte di un cambiamento di stato [37]. Si può notare che non tutti i Widget vengono aggiornati ma soltanto quelli che hanno come "genitore" `Provider.of < T > (context)`.

Pertanto, per ovviare a questa problematica, si può specificare esplicitamente quali Widget devono rimanere in ascolto per poter essere aggiornati di un nuovo cambiamento di stato (Figura 6.3). Ci sono vari modi per specificarlo:

1. Chiamando il metodo `Provider.of<T>(context);`
2. Utilizzando l'oggetto `Consumer`

```

Consumer<T>(
    builder: ( context , user , child ) {
        // Return a Widget tree
    }
)
  
```

L'oggetto `Consumer<T>` specifica che, al cambiamento di stato, deve essere aggiornato il Widget tree da quel punto in poi, evitando così di ricaricare i alcuni Widget immutabili. Queste tecniche permettono di non aggiornare eventuali Widget che si trovano a dei livelli superiori del Widget tree totale dell'applicazione, infatti, posizionare l'oggetto `Consumer` o chiamare il metodo `Provider.of<T>(context)` vicino alla radice del Widget tree complessivo, potrebbero provocare dei rallentamenti dell'applicazione. La problematica che

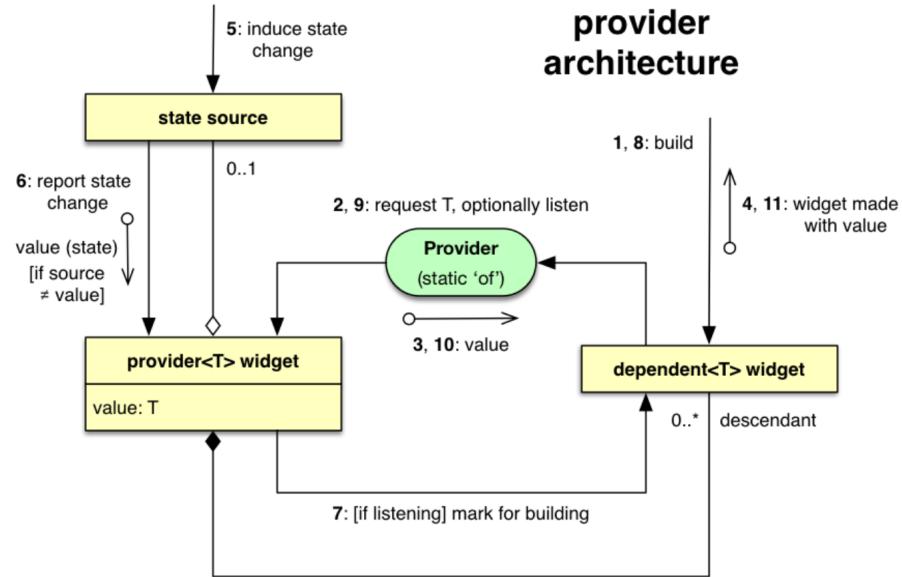


Figura 6.3: Architettura complessiva del pattern Provider[37].

nessuno di questi due metodi risolve è l'eventuale presenza di uno o più Widget che non devono essere aggiornati, ma che fanno comunque parte del sotto albero di `Consumer` o di `Provider.of<T>(context)`. È possibile ovviare a questo problema, specificando il valore del parametro `listen` nel seguente modo: `Provider.of<T>(context, listen: false)`. Così facendo, il Widget comunica al framework che è soltanto interessato a ricevere i dati dal Provider e che non necessita di essere ricostruito dal motore di rendering.

Aggiornamento dello stato

L'aggiornamento dello stato sfrutta i benefici che offre la programmazione reattiva. Possono venire utilizzati gli `Stream`, i `Future`, i `ChangeNotifier` e i `ValueNotifier`. Tutti questi oggetti permettono di comunicare istantaneamente i cambiamenti che rilevano. Così facendo, permettono di avere un flusso di dati che vengono resi disponibili al livello di presentazione dell'applicazione in tempo reale. L'*aggiornamento dello stato* verrà descritto con maggior dettaglio nella spiegazione dedicata al pattern finale dell'applicazione.

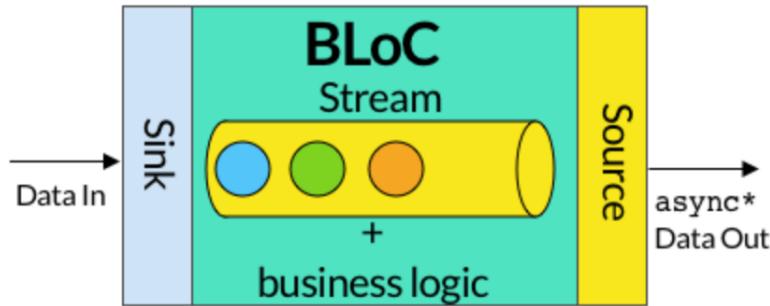


Figura 6.4: Architettura complessiva del pattern BLoC.

6.2.4 BLoC

Strutturazione del BLoC

Il **BLoC** è un sistema di *state management*, consigliato dagli stessi sviluppatori di Google. Aiuta a gestire lo stato e ad accedere ai dati da una posizione centrale nel progetto. Questo pattern è stato presentato per la prima volta alla conferenza *DartConf* del 2018 da parte da Google [38]. Tuttavia, Google non lo considera un pattern ufficiale per la realizzazione di applicazioni in Flutter, ma vuole semplicemente proporre un modello che può essere integrato con altre architetture o utilizzato singolarmente per lo sviluppo di app.

L'intento principale del pattern è quello di mantenere separata la **business logic** dall'**interfaccia grafica**. Non a caso, l'acronimo di BLoC è **Business Logic Components**. È uno dei pattern che vengono maggiormente utilizzati nella realizzazione di applicazioni in Flutter, in quanto permette di avere una struttura **scalabile** e **flessibile**. Anche questo pattern, come i precedenti, sfrutta la programmazione reattiva, sia per rilevare immediatamente un'interazione dell'utente, sia per propagare un nuovo stato ai vari Widget dell'applicazione.

Questo pattern può essere considerato come una *pila*: l'entrata della pila si chiama **sink** e l'uscita si chiama **stream**. In entrata vengono accettati gli eventi generati dall'applicazione, come ad esempio un tap dell'utente, e in uscita viene ritornato uno stato che può essere utilizzato per propagare il cambiamento nell'albero dei Widget. Sulla base di questi due termini, si può intuire che l'oggetto principale che permette di rendere disponibile il flusso di dati è lo **Stream**. La Figura 6.5 rappresenta graficamente le direzioni dei vari flussi all'interno del pattern.

I componenti principali di questo pattern sono due:

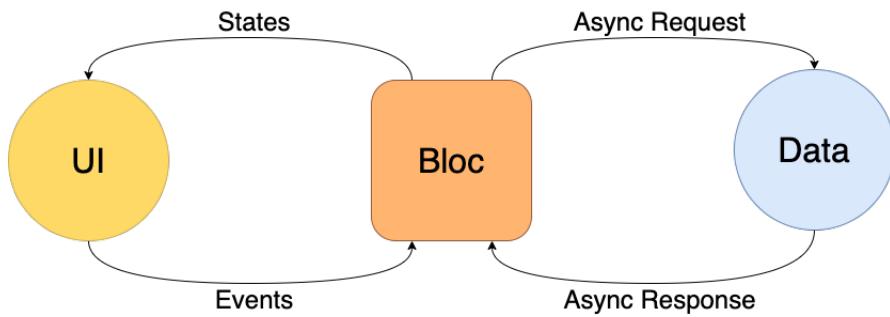


Figura 6.5: Rappresentazione del flusso di dati e di eventi.

1. UI: l'interfaccia grafica con cui l'utente può interagire;
2. BLoC: la logica dell'applicazione che va a gestire gli eventi della UI e si collega a dei servizi esterni o a delle risorse interne (ad esempio, una API, cache ed altro ancora).

Al verificarsi di un evento nell'interfaccia grafica, questo viene comunicato al BLoC (*sink*), il quale elaborerà l'evento ed eventualmente apporterà un cambiamento di stato. Il BLoC, oltre a gestire gli eventi della UI, fornisce i dati prelevati da dei servizi esterni e li invia all'interfaccia grafica. L'uso degli Stream permette di sfruttare i vantaggi della programmazione reattiva, avendo così un'applicazione performante e reattiva.

Lo Stream viene istanziato dal BLoC (tramite l'oggetto `StreamController`) e viene poi utilizzato dall'interfaccia grafica grazie al Widget `StreamBuilder`. Nel `child` dello `StreamBuilder` viene associato tutto l'albero dei Widget che verrà aggiornato ogni volta che un nuovo dato sarà disponibile.

Integrazione del pattern Provider

Il BLoC è un componente che deve essere inserito nel Widget tree per poter essere usufruito. Pertanto, viene utilizzato il pattern *Provider* a supporto del BLoC, il quale appunto permette di inserirlo all'interno dell'albero dei Widget. In particolare, il BLoC dovrà essere inserito prima dello `StreamBuilder`, in quanto quest'ultimo dipende strettamente dallo Stream dichiarato nel BLoC.

L'architettura così descritta definisce una struttura sufficientemente rigida da poter vincolare lo sviluppatore a seguire il flusso *unidirezionale* dei dati. In questo modo, risulta più semplice focalizzarsi sulla realizzazione di un'interfaccia grafica reattiva, senza preoccuparsi della logica che deve essere implementata. Dualmente, nella business logic non è necessario conoscere come verrà

realizzata l'interfaccia grafica: ci si concentra invece nel ricevere gli eventi dalla UI e in base ad essi, svolgere le azioni corrispondenti.

Un vincolo che è bene rispettare, è l'implementazione del metodo `dispose()`. Questo metodo viene chiamato quando il BLoC deve essere deallocated e pertanto è necessario chiudere correttamente lo `Stream`, per evitare di causare degli errori nell'applicazione. L'ambiente di sviluppo stesso (che sia Android Studio, IntelliJ, Visual Studio Code o altri) genera un *warning* se nella classe del BLoC non viene mai chiuso lo `Stream`.

Oltre alla rigidità della struttura, è concessa anche una certa flessibilità nella realizzazione dei componenti: un Widget complesso può essere scomposto in Widget più semplici, in modo che siano più facili da gestire e da riusare. Ogni Widget scomposto possiede il proprio BLoC. Quindi è possibile realizzare un componente, con il rispettivo BLoC, ed inserirlo in differenti schermate dell'applicazioni con molta facilità.

Questa architettura impone il rispetto di una regola fondamentale: **non è possibile condividere** lo stesso BLoC per Widget differenti. Questo dubbio può sorgere allo sviluppatore in situazioni in cui si verifica la necessità di condividere gli stessi dati a due o più Widget differenti contemporaneamente. Risulta naturale, in tale situazione, collegare tutti i Widget sotto un unico BLoC. Tuttavia questa *non è una buona pratica* di programmazione. La condivisione di un BLoC per più schermate può generare diverse problematiche, ad esempio:

1. L'aggiornamento di una schermata provoca l'aggiornamento anche delle altre schermate, anche quando non vi è bisogno;
2. Nel momento in cui viene chiamato il metodo `dispose()`, il BLoC viene deallocated per tutte le schermate con cui è stato condiviso. Quindi navigando tra le schermate, dopo che il BLoC è stato deallocated, si possono verificare degli errori e dei *crash* dell'applicazione.

Un vantaggio dell'integrazione del Provider nel pattern BLoC, è la possibilità di poter istanziare soltanto una volta il BLoC. Una volta istanziato, questo può essere utilizzato in qualsiasi punto dell'applicazione, purché vi sia un `context`. Così facendo, è possibile avere un'ottima flessibilità dell'architettura, evitando di istanziare più volte lo stesso oggetto durante il ciclo di vita dell'applicazione e potendolo utilizzare con una certa facilità, quando lo si necessita.

6.2.5 Motivazioni della scelta

È stato scelto di utilizzare il pattern *BLoC* in quanto l'ho confrontato con le altre architetture disponibili e sono giunto alla conclusione che fosse

il pattern ideale per la realizzazione di questa applicazione. Viene sfruttata appieno la programmazione reattiva, caratteristica peculiare del linguaggio, e fornisce una struttura rigida. La rigidità dell'architettura permette di creare delle applicazioni chiare dal punto di vista strutturale. Inoltre permette di avere una certa flessibilità per l'implementazione di funzionalità future.

6.3 Architettura proposta

In questo paragrafo vengono spiegate le problematiche dell'architettura BLoC e le motivazioni che hanno portato a dover attuare delle opportune modifiche per soddisfare i requisiti dell'applicazione.

6.3.1 Problematiche del pattern BLoC

L'applicazione che deve essere realizzata richiede di accedere a dei dati che devono essere forniti da un server, tramite una socket. Questi dati devono essere condivisi tra più schermate: in particolare, si avrà una schermata generale (una *dashboard*), per avere un quadro generale di tutti i dati, e poi vi saranno delle schermate specifiche, che illustreranno soltanto un piccolo set di dati. Tutte queste schermate attingono da un'unica fonte di dati: dalla socket.

Come è stato discusso precedentemente, non è possibile condividere un BLoC per più schermate contemporaneamente. Pertanto, si è dovuti procedere a modificare l'architettura proposta dalla comunità di Flutter, per poterla adattare alle esigenze richieste.

6.3.2 Introduzione del Repository pattern

Per poter risolvere le limitazioni del BLoC, viene integrato il **Repository** pattern [39]. Una spiegazione del Repository è stata data da Martin Fowler nel libro *Patterns of Enterprise Application Architecture*. L'autore spiega che un "*Repository incapsula l'insieme di oggetti persistenti in un archivio dati e le operazioni eseguite su di essi, fornendo una vista più orientata agli oggetti del livello di persistenza.*" Il Repository permette anche di ottenere una netta separazione e una dipendenza unidirezionale tra il dominio e i livelli di mappatura dei dati. Questa concezione è nata in un periodo in cui le applicazioni *enterprise* necessitavano ridurre la duplicazione della logica delle query.

Tuttavia, nel tempo, le esigenze delle applicazioni sono cambiate, in particolare per le applicazioni mobili. In questo contesto, il concetto di Repository viene inteso più come un componente che si interfaccia con un archivio esterno (in generale, qualsiasi altro servizio esterno all'applicazione). In questo caso,

il Repository ha il compito di nascondere l'implementazione della logica che permette di interfacciarsi con i servizi esterni. Così facendo, la struttura dell'applicazione risulta essere più flessibile per le modifiche future dei requisiti e il conseguente *refactoring*.

Si possono notare delle differenze sostanziali tra le due definizioni. Per la realizzazione dell'applicazione è stata presa in considerazione l'ultima definizione descritta, le cui motivazioni verranno esposte nella sezione successiva.

6.3.3 Architettura modificata

L'architettura proposta per l'applicazione è costituita dall'integrazione del pattern **BLoC** con il **Repository** pattern. Come è stato illustrato precedentemente, nella versione base, il BLoC si occupa di interfacciarsi con la UI da un lato e con i servizi esterni dall'altro. Per sopperire alla problematica della condivisione dei dati ricevuti dalla socket tra le diverse schermate, viene introdotto un Repository tra il BLoC e i servizi esterni. Così facendo si possono ottenere i seguenti vantaggi:

1. In questo modo, è possibile utilizzare il Repository come punto di riferimento per la ricezione dei dati per i BLoC. Ovvero, i vari BLoC possono collegarsi ad un Repository comune a tutti e possono ricevere i dati filtrati dal Repository stesso. Infatti, il Repository permette di astrarre dalla sorgente di dati esterna: i BLoC devono soltanto interfacciarsi con il Repository per ottenere i dati, senza preoccuparsi se questi provengano da una socket, da un'API, dalla memoria interna del dispositivo o da qualsiasi altra fonte di dati. Il Repository si preoccupa di interfacciarsi con tali servizi e di confezionare in un certo formato i dati ricevuti, per renderli disponibili e comprensibili ai vari BLoC che vi si collegano;
2. Grazie a questo nuovo livello di astrazione, tutta la business logic viene alleggerita dal carico di lavoro sostenuto per connettersi ai servizi esterni. Infatti, nel caso in cui in futuro si dovrà cambiare il servizio che permette di ottenere i dati navali, bisognerebbe andare a modificare il codice di ogni singolo BLoC che è stato creato. L'utilizzo invece del Repository, toglie questa responsabilità ai BLoC ed esso diventa l'unico "servizio" a cui i BLoC si devono collegare.

Uno svantaggio dell'integrazione di queste due architetture è appunto l'introduzione di un nuovo livello di astrazione, ovvero, l'introduzione di *overhead*. Per ridurre al minimo questa possibile problematica, viene in aiuto la programmazione reattiva con l'utilizzo degli *flussi* (**Stream** o **ValueNotifier**). Altro fattore che permette di minimizzare questo problema è quello di ridurre al

minimo il lavoro che deve svolgere il Repository. I compiti che deve svolgere sono:

1. Connetersi con la socket;
2. Ricevere i dati;
3. Confezionare i dati ricevuti in un oggetto comprensibile da parte di tutti i BLoC;
4. Trasmettere i dati in uno Stream.

Il lavoro di selezione dei dati viene effettuato dai singoli BLoC, rendendo più veloce il passaggio dei dati dalla sorgente fino ai vari componenti della business logic.

6.4 Architettura implementata

Nei paragrafi precedenti, si è andati a descrivere nel particolare le varie architetture, illustrando le motivazioni per cui sono state scelte. Sono state esposte le varie problematiche ed è stata proposta una nuova versione dell’architettura. In questo paragrafo verrà mostrata l’architettura complessiva implementata nell’applicazione.

6.4.1 Descrizione

Sulla base di tutte le nozioni che sono state fornite precedentemente, si è giunti ad una struttura finale dell’architettura dell’applicazione. In particolare, gli strati dell’architettura sono (Figura 6.6 [40]):

1. **Interfaccia grafica (View)**: tutti quei componenti grafici con cui l’utente può interagire e visualizzare le informazioni richieste;
2. **BLoC**: lo strato in cui è contenuta la logica dell’applicazione. Lo logica totale dell’app è data dall’insieme di tutti i BLoC delle singole schermate e dei singoli Widget;
3. **Repository**: un livello che si occupa di astrarre il link tra i BLoC e i servizi esterni;
4. **Servizi esterni**: in questo caso, sono rappresentati dalla socket. In futuro, altri servizi che potrebbero essere inseriti sono, ad esempio, un’API o anche i dati GPS provenienti dal sensore installato sul dispositivo mobile.

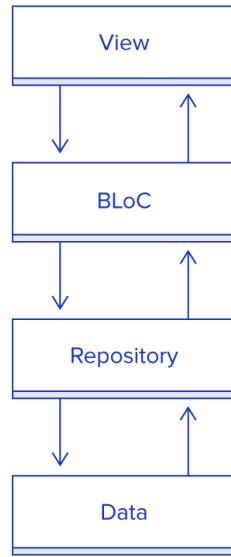


Figura 6.6: Illustrazione degli strati dell’architettura implementata.

6.4.2 Integrazione del Repository

Nell’architettura di base del BLoC, le classi che implementavano la business logic si occupavano di andare a richiedere i dati ai vari servizi esterni. Ora, invece, i vari BLoC si collegano al Repository, che fa da intermediario tra i servizi esterni e i BLoC stessi. L’aggiunta di uno strato ulteriore implica la modifica della modalità di comunicazione tra i BLoC e i servizi. Con il Repository, la comunicazione avviene nel seguente modo;

1. Il Repository si collega ai vari servizi esterni, effettua una richiesta sulla base di un particolare evento ricevuto, riceve la risposta e la inoltra in un `ValueNotifier<T>`. Un `ValueNotifier<T>` può essere usato per contenere un singolo valore e avvisare i suoi ascoltatori quando questo cambia;
2. Al `ValueNotifier<T>` del Repository, si collegano tutti i BLoC di cui necessitano i dati provenienti da questo strato;
3. Nei BLoC viene inizializzato un *ascoltatore*, che alla ricezione di un nuovo dato nel `ValueNotifier<T>`, chiama una funzione, passandogli come parametro il dato appena ricevuto;

4. Una volta ricevuto il dato, la funzione del BLoC specifico elimina i dati di cui non necessita (il Repository fornisce tutti i dati provenienti dalla socket) e li propaga nell'interfaccia grafica. La comunicazione tra la UI e i BLoC avviene tramite **Stream**: la comunicazione tra questi due strati non è cambiata dalla versione base del pattern.

L'implementazione dell'architettura così descritta, permette di **astrarre** dal tipo di servizio che viene utilizzato e permette anche di avere un **punto di riferimento per la distruzione dei dati** ai vari BLoC. Tutte queste caratteristiche rendono l'architettura adatta per l'applicazione che deve essere implementata.

Per quanto riguarda il passaggio dei dati tra il Repository e i BLoC, si è deciso di non inviare i dati ai BLoC già convertiti e pronti per essere utilizzati. Il motivo è quello di non *sovrafficare* il Repository, in quanto causerebbe il degrado delle prestazioni. Così facendo, il carico di lavoro viene distribuito tra il Repository e i BLoC.

L'integrazione del Repository pattern nell'architettura avviene grazie all'utilizzo del *Provider*. È possibile fare un discorso analogo a quello che è stato fatto per l'integrazione del Provider nel BLoC. Essendo un componente di *fondamentale* importanza, il Repository viene posto vicino alla radice dell'albero dell'applicazione: venendo utilizzato da diverse schermate durante il ciclo di vita dell'applicazione, nel momento in cui avviene la chiusura dell'app, il Repository sarà uno degli ultimi oggetti che verranno deallocati. In questo modo, le schermate non saranno soggette ad errori, in quanto queste verranno deallocate prima del Repository.

6.5 Descrizione dettagliata dell'architettura implementata

In questa sezione si procede a descrivere dettagliatamente l'architettura implementata, a partire dai componenti più interni, per procedere poi a trattare le parti che gestiscono l'interazione con l'utente.

6.5.1 Organizzazione delle classi e dei package

In questa breve sezione, viene illustrata l'organizzazione dei package e delle varie classi dell'applicazione realizzata.

L'applicazione è stata strutturata seguendo gli strati dell'architettura, ovvero, è stato creato un package per ogni livello dell'architettura.

L'applicazione è stata organizzata nel seguente modo (in ordine alfabetico):

1. blocs

- (a) dashboard
 - i. *dashboard_widget_bloc.dart*
- (b) gps
 - i. *gps_bloc.dart*
- (c) location
 - i. *location_bloc.dart*
- (d) navigation
 - i. *navigation_bloc.dart*
- (e) wind
 - i. *wind_bloc.dart*
- (f) *bloc.dart*

2. models

- (a) repository
 - i. *repository_data_model.dart*
 - ii. *timestamp.dart*
- (b) settings
 - i. cache
 - A. *cache.dart*
 - ii. server
 - A. *server_parameters.dart*
 - B. *server_settings.dart*
 - C. *server_settings_file.dart*
- (c) ui
 - i. *base_model.dart*
 - ii. *dashboard.dart*
 - iii. *gps.dart*
 - iv. *location.dart*
 - v. *navigation.dart*
 - vi. *wind.dart*
- (d) *acronyms.dart*

3. repositories(a) *repository_socket.dart***4. services**(a) *socket_service.dart***5. ui**

(a) common_widgets

i. app_bar

A. *page_app_bar.dart*B. *sensor_data_app_bar.dart*

ii. grid_box

A. *grid_boxwidget.dart*

iii. location

A. *location_widget.dart*

(b) pages

i. dashboard

A. *widgets.dashboard_widget.dart*B. *dashboard_page.dart*

ii. gps

A. *gps_page.dart*

iii. navigation

A. *navigation_page.dart*

iv. settings

A. *cache.cache_page.dart*B. *server.server_page.dart*C. *theme.theme_page.dart*D. *settings_page.dart*

v. wind

A. *wind_page.dart***6. utils**

(a) ui

i. themes

- A. *custom_theme.dart*
- B. *default_theme.dart*
- C. *high_contrast_theme.dart*
- D. *theme_handler.dart*
- E. *themes.dart*
- ii. *colors_palette.dart*

7. *main.dart*

6.5.2 Modelli

In questa sezione vengono descritti tutti i modelli utilizzati dai vari strati dell'applicazione.

Acronyms

Questa è una semplice classe contenente delle mappe *acronimo - nome del dato* che vengono utilizzate nelle schermate dell'applicazione. I dati forniti dalla socket sono espressi sotto forma di *acronimi*. Tuttavia in alcune sezioni dell'applicazione è preferibile avere il nome completo del dato che si sta osservando.

La mappa che contiene le associazioni tra acronimi e nomi dei dati dei sensori è la seguente:

```

1  static final Map<String, String> _acronyms = {
2      "aws": "Apparent\u2022Wind\u2022Speed",
3      "awa": "Apparent\u2022Wind\u2022Angle",
4      "sog": "Speed\u2022Over\u2022Ground",
5      "cog": "Curse\u2022Over\u2022Ground",
6      "mh": "Magnetic\u2022Heading",
7      "sow": "Speed\u2022Over\u2022Water",
8      "tws": "True\u2022Wind\u2022Speed",
9      "twa": "True\u2022Wind\u2022Angle",
10     "ih": "imu_heading",
11     "cang": "calypso_ang",
12     "camp": "calypso_amp",
13     "nwang": "nwang",
14 };

```

Tramite il seguente metodo, dato un acronimo, è possibile ottenere il nome completo del dato corrispondente:

```
1 static String getSentence(String acronym) {  
2     return _acronyms[acronym];  
3 }
```

RepositoryDataModel

Questo oggetto viene utilizzato per incapsulare i dati ricevuti dalla socket. Grazie a questo oggetto è possibile poi trasmetterlo al Repository e ai vari BLoC.

Si può notare dal commento, che l'introduzione di un oggetto `Timestamp` sarà necessario per dei sviluppi futuri, ad esempio, la realizzazione di grafici per i vari sensori. Il timestamp è già presente ogni volta che viene fatta una richiesta alla socket e vengono ricevuti dei dati. Inoltre il timestamp è già presente nella *cache*: è sufficiente utilizzarlo in quanto l'implementazione effettiva è già stata predisposta.

L'oggetto `Timestamp` contiene delle informazioni riguardanti l'ora, il minuto e il secondo in cui i dati sono stati ricevuti dalla socket.

```
1 class RepositoryDataModel {  
2     final String jsonData;  
3     final Timestamp timestamp;  
4     // I need this to know what time I requested the data  
5     // from the socket, so in this way I can build a chart  
6  
7     RepositoryDataModel({@required this.jsonData,  
8         @required this.timestamp});  
9  
10    @override  
11    String toString() {  
12        return jsonData + "\n" + timestamp.toString();  
13    }  
14 }
```

BaseModel

Questa *classe astratta* rappresenta un riferimento comune per tutti i modelli delle specifiche schermate. Per ogni schermata viene associato un particolare modello, nei quali vengono salvati soltanto un piccolo set dei dati forniti dal Repository.

L'unico metodo che dovrà essere implementato dai vari modelli che estenderanno questa classe astratta, è il metodo `toMap()`, che permette alla classe di restituire i dati che essa contiene, sotto forma di una mappa.

```

1 abstract class BaseModel {
2     Map<String, dynamic> toMap();
3 }
```

Dashboard

Il modello che verrà illustrato è il modello che contiene il maggior numero di campi, in quanto la schermata *Dashboard* ha il compito di mostrare all'utente un quadro generale della situazione, presentando tutti i dati ricevuti dal Repository.

Il metodo `Dashboard.fromJson(Map<String, dynamic> json)` estrapola tutti i dati di cui necessita, da un particolare oggetto che li contiene. Questo metodo viene chiamato nel momento in cui vengono ricevuti i dati dal Repository, che sono salvati in formato JSON.

I modelli `Navigation`, `GPS` e `Wind` sono molto simili al modello che è stato presentato. Variano soltanto per numero e tipologia di dati memorizzati al loro interno.

Il modello `Location` è molto simile al modello `Dashboard`, con l'unica differenza che non viene implementato il metodo `fromJson`. Il motivo è che i dati vengono già decodificati dal JSON e vengono poi passati direttamente a tale modello.

```

1 class Dashboard extends BaseModel {
2     final String apparentWindSpeed;
3     final String apparentWindAngle;
4     final String speedOverGround;
5     final String curseOverGround;
6     final String magneticHeading;
7     final String speedOverWater;
8     final String trueWindSpeed;
9     final String trueWindAngle;
10    final String imu_heading;
11    final String calypso_ang;
12    final String calypso_amp;
13    final String nwang;
14
15    // Constructor
16    Dashboard({
```

```
17     this.apparentWindSpeed = "0",
18     this.apparentWindAngle = "0",
19     this.speedOverGround = "0",
20     this.curseOverGround = "0",
21     this.magneticHeading = "0",
22     this.speedOverWater = "0",
23     this.trueWindSpeed = "0",
24     this.trueWindAngle = "0",
25     this.imu_heading = "0",
26     this.calypso_ang = "0",
27     this.calypso_amp = "0",
28     this.nwang = "0",
29   );
30
31 factory Dashboard.fromJson(Map<String, dynamic> json) {
32   return Dashboard(
33     apparentWindSpeed: json['aws'],
34     speedOverGround: json['sog'],
35     curseOverGround: json['cog'],
36     magneticHeading: json['mh'],
37     speedOverWater: json['sow'],
38     trueWindSpeed: json['tws'],
39     trueWindAngle: json['twa'],
40     imu_heading: json['imu_heading'],
41     calypso_ang: json['calypso_ang'],
42     calypso_amp: json['calypso_amp'],
43     nwang: json['nwang'],
44   );
45 }
46
47 @override
48 Map<String, dynamic> toMap() {
49   ...
50 }
```

Cache

Questo modello permette di salvare i dati che vengono ricevuti dalla socket. I dati vengono salvati nella *cache* dal Repository. Le varie schermate utilizzano la cache nella loro fase di inizializzazione, ovvero, recuperano l'ultimo dato salvato per poter istanziare correttamente tutti i componenti della schermata.

La **Cache** è una classe parametrica, quindi è possibile utilizzarla in qualsiasi contesto e con qualsiasi tipo si desideri. I vari dati vengono memorizzati in una lista. I metodi che supporta questa classe sono:

1. `save(T data)`: questo metodo si interfaccia con la lista interna e permette di salvare un nuovo dato in tale lista;
2. `void emptyCache()`: questo metodo permette di svuotare la cache;
3. `bool isEmpty()`: permette di verificare se la lista interna è vuota o meno;
4. `T getLastDataCached()`: è un metodo interno alla classe e permette di ottenere l'ultimo dato che è stato salvato nella lista;
5. `get lastData`: è un metodo pubblico che utilizza `_getLastDataCached()`. Non è stato utilizzato direttamente il metodo privato in quanto si voleva rendere il nome del metodo più intuitivo;
6. `List<T> _getDataCached ()`: questo metodo privato permette di ottenere tutto il contenuto della cache;
7. `get dataCached`: è un metodo pubblico che chiama `_getDataCached ()`. Le motivazioni della creazione di questo metodo sono analoghe a quelle del metodo `get lastData`;
8. `get size`: questo metodo permette di ottenere quanti dati sono stati salvati nella cache, ovvero, nella lista.

ServerParameters

Questa classe permette di salvare tutti i dati necessari per effettuare la connessione al server (indirizzo IP e porta) e per implementare il *polling* (un numero intero che rappresenta l'intervallo di tempo, espresso in secondi). La classe in questione, viene utilizzata dalla schermata *Server*, nella quale è possibile aggiornare i parametri di connessione e del polling. Pertanto, vi sono diversi metodi che vanno ad analizzare i diversi input, in formato stringa, per controllare che rispettino i tipi e i formati richiesti.

ServerSettingsFile

Questa classe ha un ruolo di interfaccia tra l'applicazione ed un file di configurazione in JSON. In questo file vengono memorizzati tutti i dati relativi alla connessione e al polling. Così facendo, al riavvio dell'applicazione, se sono

state effettuate delle modifiche a tali parametri, questi non vengono persi e possono essere ricaricati. Le operazioni supportate da questa classe sono:

1. `static Future<String> readSettings()`: operazione per leggere dal file JSON;
2. `static Future<File> writeSettings(String serverSettings)`: metodo che permette di scrivere i dati sul file JSON.

Vengono utilizzati degli oggetti `ServerSettings` per incapsulare tutti i dati che devono essere memorizzati nel file.

ServerSettings

Questa classe contiene i medesimi parametri della classe `ServerParameters` e viene utilizzata dalla classe `ServerSettingsFile`. L'oggetto ottenuto per istanziazione di questa classe, possiede un metodo per *codificare* i dati in JSON (`Map<String, dynamic> toJson()`), in modo da poterli memorizzare nel file, e un metodo per *decodificare* i dati dal file JSON (`factory ServerSettings.fromJson(Map<String, dynamic> json)`), in modo da renderli disponibili ai livelli superiori dell'applicazione.

6.5.3 Socket

La *socket* che viene fornita dal server *Argos* tramite la porta 4545, rappresenta il servizio esterno da cui l'applicazione prende i dati necessari. Nel caso in cui in futuro, i medesimi dati vengano resi disponibili tramite un altro software o un'altra metodologia, sarà sufficiente creare una nuova classe o un nuovo modulo che implementi la connessione al servizio e i metodi per ottenere i dati da tale servizio.

Metodo `getData()`

Questo metodo permette di connettersi ad una socket e di ottenere i dati da essa. Secondo il protocollo che è stato realizzato nel server *Argos*, una volta che il client effettua una richiesta al server, il server invia la risposta e chiude immediatamente la socket. Pertanto, una volta ricevuti i dati (da riga 10 a riga 15), la socket deve essere chiusa correttamente anche dal lato del client (riga 17).

Il metodo appena descritto è un metodo *asincrono*, ovvero, l'applicazione non viene bloccata fino a quando non vengono ricevuti i dati, ma è possibile svolgere altre azioni. Nel caso ci fossero delle problematiche con la connessione

ad Internet, l'implementazione che è stata attuata permette di non bloccare l'applicazione.

Il tipo di ritorno è un `Future<RepositoryDataModel>`: `Future` perchè è un metodo asincrono, e `RepositoryDataModel` perchè i dati ricevuti dalla socket vengono incapsulati in un oggetto comune che possa essere comprensibile ai vari BLoC che si collegheranno al Repository. Incapsulare i dati già in questo strato dell'applicazione permette di alleggerire il carico di lavoro del Repository.

```

1  Future<RepositoryDataModel> getData() async {
2      Completer<RepositoryDataModel> repositoryData =
3          Completer<RepositoryDataModel>();
4
5      try {
6          Socket socket = await Socket.connect(serverIPAddress,
7              serverPort, timeout: Duration(seconds: 2));
8          log("Socket\u25a1connected", name: "SailingSocket");
9
10         socket.listen((data) async {
11             repositoryData.complete(RepositoryDataModel(
12                 jsonData: String.fromCharCodes(data).trim(),
13                 timestamp: Timestamp(dateTime: DateTime.now()),
14             ));
15         });
16
17         socket.close();
18         log("Socket\u25a1closed", name: "SailingSocket");
19     } on SocketException catch (e) {
20         rethrow;
21     } on TimeoutException catch (e) {
22         rethrow;
23     }
24     return repositoryData.future;
25 }
```

6.5.4 Repository

Come è già stato descritto precedentemente, il Repository è uno strato che si interpone tra i BLoC e i vari servizi esterni. Il compito del Repository è quello di nascondere come avviene la connessione con i servizi esterni. Questi possono essere dei *Web services*, una socket o qualsiasi altra fonte di dati. In questo

modo, gli altri componenti utilizzano un'interfaccia unica per connettersi al Repository, indipendentemente dalla tipologia di servizio che viene usufruito.

La scelta di introdurre un ulteriore strato all'architettura originale è dovuta anche ad una caratteristica intrinseca del progetto. Un requisito dell'applicazione consiste nell'interrogare periodicamente (eseguire un *polling*) la socket, in modo da avere i dati sempre aggiornati. Inoltre, tutte le schermate dell'applicazione attingono i dati dalla medesima fonte. Non potendo condividere i BLoC per più schermate, la soluzione del Repository risulta essere la scelta più ragionevole. Quindi, nel Repository viene svolto il *polling*, ovvero, periodicamente vengono effettuate delle richieste alla socket sulla base di un parametro che indica quanti secondi devono essere attesi tra una richiesta e quella successiva.

Da un punto di vista più approfondito, nel Repository è presente:

1. `ValueNotifier<RepositoryDataModel>`: permette di sfruttare la programmazione reattiva, comunicando, a tutti gli oggetti connessi al Repository, i dati trasmessi attraverso questo oggetto;
2. `start()`: questo metodo viene invocato nella schermata *Dashboard* e permette di inizializzare il *polling* del servizio;
3. `stop()`: questo metodo permette di fermare il polling. Viene invocato dall'utente grazie all'interazione con la schermata *Dashboard*;
4. `_getDataFromSocket()`: è un metodo privato (in Dart vengono identificati con il carattere di *underscore*, '_') che va ad effettuare esplicitamente la chiamata al servizio esterno. Una volta ricevuto il dato, questo viene trasmesso nel `ValueNotifier<RepositoryDataModel>`;
5. `dispose()`: è un metodo che viene invocato nel momento in cui viene deallocated il Repository. È buona pratica chiudere correttamente gli `Stream` e i `ValueNotifier<T>` per evitare che il canale di comunicazione rimanga aperto.

Metodo `start()`

Di seguito verrà illustrato il codice del metodo `start()`. Questo metodo permette di inizializzare l'oggetto `Timer`, tramite il quale sarà possibile implementare il *polling*. I parametri di `Timer.periodic()` (riga 16) sono una funzione da eseguire periodicamente ed il numero di secondi che indica ogni quanto effettuare una richiesta. La funzione `_getDataFromSocket()` viene chiamata per poter ottenere i dati dalla socket.

Si può notare che prima di eseguire `Timer.periodic()`, viene effettuata una chiamata alla funzione `_getDataFromSocket()` (riga 11). Questo viene fatto in quanto, quando viene chiamato `Timer.periodic()`, non viene immediatamente fatta una connessione alla socket, ma si attende un tempo pari a `pollingInterval`. Pertanto, l'utente, dopo aver premuto il bottone di avvio, potrebbe pensare che l'applicazione si sia bloccata o abbia subito un rallentamento. Effettuando una chiamata a `_getDataFromSocket()` è possibile ottenere immediatamente i dati, senza dover attendere.

Le variabili booleane private `_internetConnection` e `_timeoutSocket` sono necessarie per la gestione degli errori di connessione con la socket.

Il tipo di ritorno di questo metodo è un `Future<void>`, in quanto il metodo `_getDataFromSocket()` è anch'esso un `Future<void>`. Inoltre, la parola riservata `await` indica che bisogna attendere il termine della funzione prima di poter continuare ad eseguire il codice del metodo chiamante.

```
1 Future<void> start() async {
2     if (!isTimerActive) {
3         isTimerActive = true;
4         log("Start", name: "Repository");
5
6         _internetConnection = true;
7         _timeoutSocket = true;
8
9         // Get data immediately at startup stage of the app
10        // Don't wait 'pollingInterval' seconds to get the data
11        await _getDataFromSocket();
12
13        if (_internetConnection && _timeoutSocket) {
14
15            // Start the timer
16            _timer = Timer.periodic(
17                Duration(seconds: pollingInterval),
18                (Timer timer) => _getDataFromSocket(),
19            );
20        }
21    }
22}
```

Metodo `_getDataFromSocket()`

Il metodo che verrà illustrato rappresenta l'effettivo utilizzo del servizio esterno, in questo caso la socket. Il metodo si interfaccia con le funzionalità messe a disposizione dalla classe `SocketService` (riga 6). Una volta ricevuti i dati, li salva nella `cache` (riga 7) e li trasmette ai vari BLoC attraverso il `ValueNotifier<RepositoryDataModel>` (riga 8). In questo modo i BLoC possono elaborare i dati ricevuti e presentarli all'utente.

```
1 Future<void> _getDataFromSocket() async {
2     RepositoryDataModel repositoryData;
3
4     if (_internetConnection && _timeoutSocket) {
5         try {
6             repositoryData = await socketService.getData();
7             cache.save(repositoryData);
8             socketData.value = repositoryData;
9
10            _internetConnection = true;
11            _timeoutSocket = true;
12        } on SocketException catch (e) {
13            _internetConnection = false;
14            print("No\u2022internet\u2022connection");
15        } on TimeoutException catch (e) {
16            _timeoutSocket = false;
17            print("Timeout");
18        }
19    }
20 }
```

Metodo `stop()`

Questo metodo va a fermare il timer su richiesta esplicita dell'utente, tramite la schermata *Dashboard*. Vengono effettuati i dovuti controlli per evitare di generare errori. L'istruzione a riga 7, permette di fermare il timer.

```
1 void stop() {
2     if (_isTimerActive) {
3         _isTimerActive = false;
4         log("Stop", name: "Repository");
5
6         if (_timer != null) {
```

```

7         _timer.cancel();
8     }
9 }
10 }
```

Metodo dispose()

Questo metodo viene utilizzato nel momento in cui deve essere deallocated il Repository. Viene chiamato il metodo `stop()`, in modo da terminare l'esecuzione del `Timer.periodic()`, per non causare eventuali errori nella chiusura dell'applicazione.

```

1 void dispose() {
2     this.stop();
3 }
```

6.5.5 BLoC

In questa sezione viene descritto il funzionamento del BLoC, a livello di codice. Durante lo sviluppo, si è notato che l'implementazione dei differenti BLoC per ogni schermata (o componente), risultava essere un'attività ripetitiva. Infatti, tutti i BLoC sono accumunati tra loro da una struttura molto simile. Pertanto, si è realizzata un'*astrazione di base* per poter implementare più facilmente tutti i BLoC necessari. Così facendo, per i futuri sviluppi, sarà più semplice andare ad aggiungere dei nuovi BLoC o modificare i BLoC esistenti.

Classe astratta

La classe `bloc.dart` va a realizzare un'*astrazione di base* per creare i BLoC necessari ai vari componenti. A partire dal costruttore (dalla riga 2 alla riga 4), si può notare il collegamento diretto con il Repository (`this.repository`) e il collegamento diretto con il flusso di dati proveniente dal Repository (`this.socketData`). In questo canale vengono trasmessi dei dati encapsulati in oggetti `RepositoryDataModel`. Nel corpo del costruttore viene inizializzato un *ascoltatore*: nel momento in cui il Repository invia dei dati nel canale di comunicazione (`ValueNotifier<RepositoryDataModel>`), il BLoC rileverà i nuovi dati ed invocherà il metodo `onReceived`.

Il metodo `onReceived` (dalla riga 13 alla riga 15) si occupa di decodificare, dal formato JSON, il contenuto dell'oggetto ricevuto. Una volta decodificato,

i dati vengono trasmessi nello `Stream`, tramite l'oggetto `StreamController`, un oggetto che si occupa di gestire gli stream.

Questa classe astratta è una classe *parametrica* `Bloc<T extends BaseModel>`, che accetta come parametro T, soltanto quelle classi che estendono la classe `BaseModel`. Il motivo di questa realizzazione è dovuto ad un aspetto implementativo dell'astrazione. Il metodo `getDataFromJSON` accetta come parametro un oggetto `RepositoryDataModel` e restituisce un risultato di tipo T. La classe del tipo di ritorno deve poter estendere la classe `BaseModel`, in quanto gli oggetti che le varie schermate gestiscono, estendono tale classe. Tutti questi aspetti hanno contribuito a realizzare una buona astrazione del concetto di BLoC.

Il metodo `dispose()` (dalla riga 17 alla riga 21) si occupa di:

1. Rimuove l'ascoltatore;
2. Chiudere il canale di comunicazione del
`ValueNotifier<RepositoryDataModel>;`
3. Chiudere lo `StreamController`, ovvero, chiudere il flusso di dati con l'interfaccia grafica.

Di seguito, il codice che implementa le funzionalità basilari del BLoC.

```
1 abstract class Bloc<T extends BaseModel> {  
2     Bloc({@required this.repository, @required this.socketData}) {  
3         socketData.addListener(onReceived);  
4     }  
5  
6     final RepositorySocket repository;  
7     final ValueNotifier<RepositoryDataModel> socketData;  
8     final StreamController _streamController =  
9     StreamController<T>();  
10  
11    Stream<T> get stream => _streamController.stream;  
12  
13    void onReceived() {  
14        _streamController.add(getDataFromJSON(socketData.value));  
15    }  
16  
17    void dispose() {  
18        socketData.removeListener(onReceived);  
19        socketData.dispose();  
20        _streamController.close();  
21    }  
}
```

```

22     T getDataFromJSON(RepositoryDataModel repositoryData);
23 }
24 }
```

NavigationBloc

Questo BLoC si interpone tra la schermata `NavigationPage` ed il Repository. La struttura di questo BLoC ha un'impostazione standard che è presente anche nei BLoC *GPS* e *Wind*. L'unico metodo che implementa la classe è il metodo `getDataFromJSON`, necessario per trasformare i dati dal formato JSON all'oggetto `Navigation` (in questo caso).

```

1 class NavigationBloc extends Bloc<Navigation> {
2     NavigationBloc({@required this.repository,
3                     @required socketData})
4         : super(repository: repository, socketData: socketData);
5     final RepositorySocket repository;
6
7     @override
8     Navigation getDataFromJSON(
9         RepositoryDataModel repositoryModel) {
10        Map<String, dynamic> map = jsonDecode(
11            repositoryModel.jsonData);
12        return Navigation.fromJson(map);
13    }
14 }
```

DashboardWidgetBloc

Questa classe è molto simile al BLoC `NavigationBloc`. Le uniche differenze sono l'implementazione dei due metodi `start()` (dalla riga 15 alla riga 17) e `stop()` (dalla riga 19 alla riga 21). Questi metodi sono necessari in quanto tramite la schermata *Dashboard*, l'utente può *avviare* o *fermare* le richieste periodiche alla socket (*polling*). I metodi descritti interagiscono direttamente con il Repository.

```

1 class DashboardWidgetBloc extends Bloc<Dashboard> {
2     DashboardWidgetBloc({@required this.repository,
3                     @required socketData})
4         : super(repository: repository, socketData: socketData);
5     final RepositorySocket repository;
6 }
```

```
7  @override
8  Dashboard getDataFromJSON(
9   RepositoryDataModel repositoryModel) {
10    Map<String, dynamic> map = jsonDecode(
11      repositoryModel.jsonData);
12    return Dashboard.fromJson(map);
13  }
14
15  void start() {
16    repository.start();
17  }
18
19  void stop() {
20    repository.stop();
21  }
22 }
```

6.5.6 View

In questa sezione vengono presentate tutte le implementazioni delle varie schermate dell'applicazione, illustrando anche le motivazioni delle scelte grafiche attuate.

Navigation

In questa schermata vengono visualizzati i dati relativi alla *navigazione*, in particolare è possibile monitorare i seguenti dati:

1. Apparent Wind Speed
2. Apparent Wind Angle
3. Magnetic Heading
4. Speed Over Water

La struttura della classe che implementa questa schermata (`NavigationPage`) risulta essere molto compatta e semplice, grazie all'astrazione che è stata fatta sui BLoC e all'astrazione di alcuni componenti grafici comuni. Di seguito viene illustrato il codice della schermata in questione.

```
1 class NavigationPage extends StatefulWidget {
2     NavigationPage(
3         {Key key, @required this.navigationBloc,
4          @required this.themeHandler})
5         : super(key: key);
6     final NavigationBloc navigationBloc;
7     final ThemeHandler themeHandler;
8
9     static Widget create(BuildContext context) {
10        final RepositorySocket repository =
11            Provider.of<RepositorySocket>(context, listen: false);
12
13        final ThemeHandler themeHandler =
14            Provider.of<ThemeHandler>(context, listen: false);
15
16        return Provider<NavigationBloc>(
17            create: (_) => NavigationBloc(
18                repository: repository,
19                socketData: repository.socketData),
20                dispose: (context, bloc) => bloc.dispose(),
21                child: Consumer<NavigationBloc>(
22                    builder: (context, bloc, _) =>
23                        NavigationPage(navigationBloc: bloc,
24                            themeHandler: themeHandler)),
25                );
26        }
27
28        @override
29        State<StatefulWidget> createState() => _NavigationPageState();
30    }
31
32    class _NavigationPageState extends State<NavigationPage>
33        with AutomaticKeepAliveClientMixin<NavigationPage> {
34        @override
35        bool get wantKeepAlive => true;
36
37        @override
38        Widget build(BuildContext context) {
39            super.build(context);
40            return GridBox(
```

```
41     bloc: widget.navigationBloc,
42     themeHandler: widget.themeHandler,
43     initialData: Navigation(),
44   );
45 }
46 }
```

`NavigationPage` è una classe che estende `StatefulWidget`, in quanto dovrà gestire eventuali cambiamenti di stato dell'applicazione. Il metodo `static Widget create(BuildContext context)` (dalla riga 9 alla riga 26) viene chiamato dal `main.dart` e permette di inizializzare correttamente sia la schermata di navigazione che il rispettivo BLoC. In particolare, è possibile notare che viene utilizzato il `Repository` grazie al `Provider`, specificando il parametro `listen:false`. In questo modo si specifica al `Provider` che all'arrivo di un nuovo dato, la grafica non deve essere aggiornata. Allo stesso modo viene utilizzato il `ThemeHandler`.

Sulla base dell'architettura proposta (sezione *Architettura implementata*), il BLoC relativo alla schermata deve essere posizionato prima dell'oggetto `NavigationPage`. Infatti, nel metodo `create` viene ritornato prima il `Provider<NavigationPage>` e poi, come figlio di quest'ultimo, `NavigationPage`. La schermata viene incapsulata nel `Consumer`: una spiegazione approfondita è stata redatta nella sezione *Recuperare ed utilizzare i valori dal Provider*.

Nella classe privata `_NavigationPageState` viene utilizzata la *mixin* `AutomaticKeepAliveClientMixin<T>` (un approfondimento è presente nell'appendice B), necessario per evitare di ricaricare i dati anche quando si cambia schermata. Il metodo `build()` (dalla riga 37 alla riga 45) ritorna un Widget comune: il `GridBox`. A questo Widget viene passato il corrispondente BLoC, la classe che gestisce i temi e i dati che devono essere visualizzati appena istanziato il Widget. In particolare, `initialData` è necessario nel momento in cui viene avviata l'applicazione e non è stata ancora effettuata alcuna richiesta alla socket. Quindi, non avendo a disposizione dei dati provenienti dal server, il Widget deve comunque illustrare dei dati iniziali. Questi dati iniziali vengono forniti appunto tramite il parametro `initialData`.

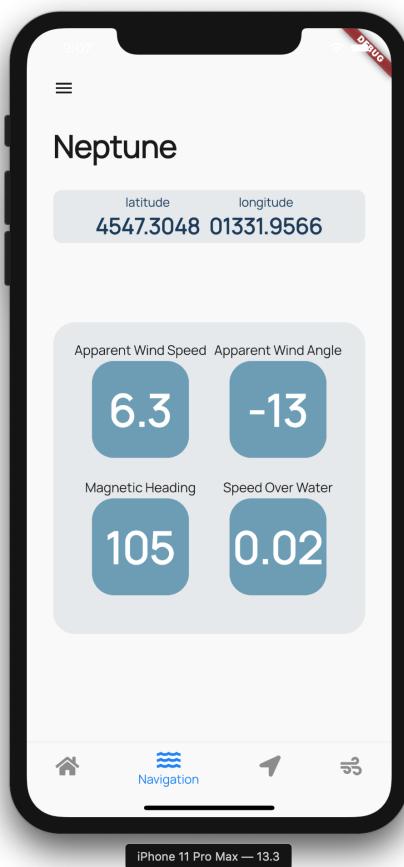


Figura 6.7: Screenshot della schermata *Navigation*.

GPS e Wind

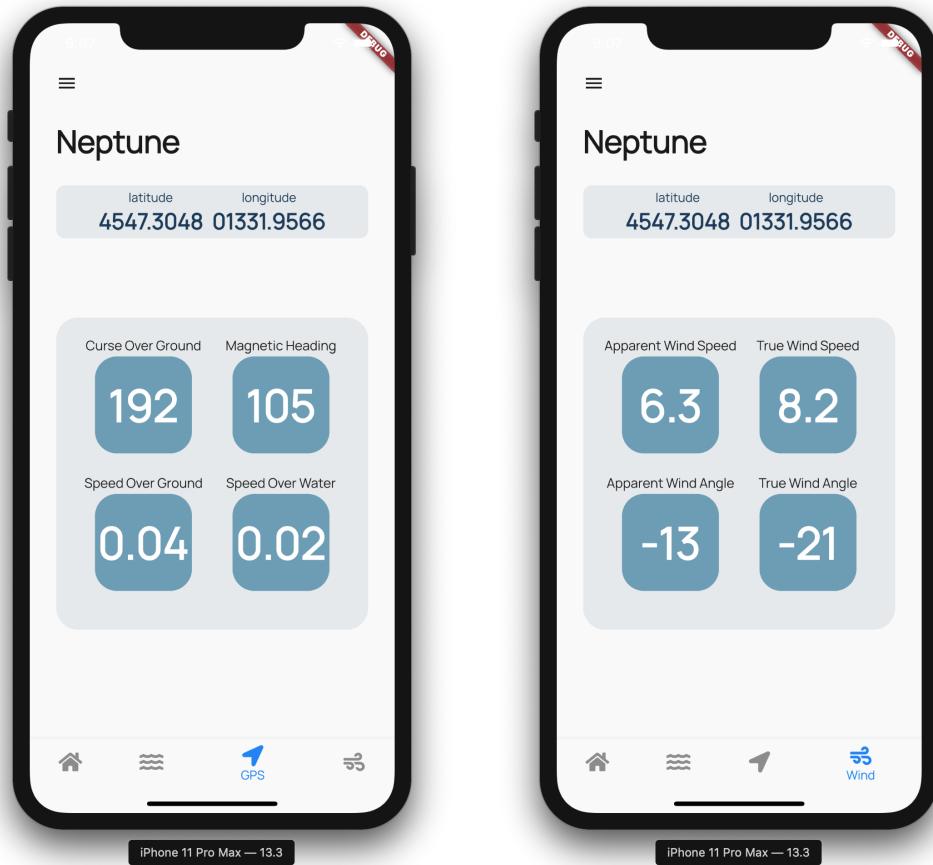


Figura 6.8: Screenshot della schermata *GPS* (a sinistra) e della schermata di *Wind* (a destra).

Nella schermata **GPS** è possibile monitorare i dati relativi alla geolocalizzazione (le coordinate vengono prese dal sensore GPS installato sul Raspberry e non dal dispositivo in cui viene utilizzata l'applicazione). Mentre, nella schermata **Wind** è possibile visualizzare i dati relativi al vento, come ad esempio la reale velocità del vento.

In particolare, nella schermata *GPS* è possibile monitorare i seguenti dati:

1. **Curse Over Ground**
2. **Magnetic Heading**
3. **Speed Over Ground**

4. Speed Over Water

Nella schermata *Wind* l'utente può controllare i seguenti dati:

1. Apparent Wind Speed
2. True Wind Angle
3. Apparent Wind Angle
4. True Wind Angle

La struttura di queste schermate segue il modello semplice e compatto della schermata *Navigation*.

Dashboard

Questa è la schermata che l'utente vede all'apertura dell'applicazione. Il principale obiettivo di questa schermata è quello di fornire all'utente un quadro generale della situazione. In questo modo può monitorare tutti i dati riguardanti la navigazione. Inoltre da questa schermata, l'utente può avviare o fermare il *polling* dei dati, ovvero, può avviare o fermare le richieste verso la socket quando lo ritiene opportuno.

Nonostante questa sia la schermata principale dell'applicazione, la spiegazione di essa viene effettuata soltanto ora in quanto possiede una struttura differente rispetto allo schema *standard* delle schermate *Navigation*, *GPS* e *Wind*. Le altre schermate seguono un approccio più semplice: ad ogni pagina è associato un BLoC. Come è possibile notare dal codice della classe *DashboardPage*, non è presente alcun riferimento ad un qualche BLoC. Questa schermata è stata strutturata in modo che la *DashboardPage* possa integrare diversi Widget per comporre una grafica complessiva. Ogni componente grafico al suo interno possiede il proprio BLoC (sia *LocationWidget* che *DashboardWidget*, riga 23 e 24).

```

1 class DashboardPage extends StatefulWidget {
2   DashboardPage({Key key}) : super(key: key);
3
4   @override
5   State<StatefulWidget> createState() => _DashboardPageState();
6 }
7
8 class _DashboardPageState extends State<DashboardPage>
9   with AutomaticKeepAliveClientMixin<DashboardPage> {
10   @override

```

```
11   bool get wantKeepAlive => true;
12
13   @override
14   Widget build(BuildContext context) {
15     super.build(context);
16     return _buildUI();
17   }
18
19   Widget _buildUI() {
20     return ListView(
21       shrinkWrap: true,
22       children: <Widget>[
23         LocationWidget.create(context),
24         DashboardWidget.create(context),
25       ],
26     );
27   }
28 }
```

Osservando il codice relativo alla classe `DashboardWidget`, è possibile notare che la struttura di questo componente è sostanzialmente uguale a quella della schermata *Navigation*. Grazie alla pagina *Dashboard*, si possono vedere i benefici e i vantaggi dell'utilizzo di un'architettura BLoC: non è necessario che ogni singola schermata possieda il proprio BLoC, ma possono essere creati dei BLoC per componenti di dimensioni più piccole. Successivamente, questi componenti possono essere assemblati in un unico Widget, come nel caso della *Dashboard* con i Widget *Location* e *DashboardWidget*.

```
1 class DashboardWidget extends StatefulWidget {
2   DashboardWidget(
3     {Key key, @required this.dashboardBloc,
4      @required this.themeHandler})
5     : super(key: key);
6   final DashboardWidgetBloc dashboardBloc;
7   final ThemeHandler themeHandler;
8
9   static Widget create(BuildContext context) {
10     final RepositorySocket repository =
11       Provider.of<RepositorySocket>(context, listen: false);
12
13     final ThemeHandler themeHandler =
14       Provider.of<ThemeHandler>(context, listen: false);
```

```
15
16     return Provider<DashboardWidgetBloc>(
17         create: (_) => DashboardWidgetBloc(
18             repository: repository,
19             socketData: repository.socketData),
20             dispose: (context, bloc) => bloc.dispose(),
21             child: Consumer<DashboardWidgetBloc>(
22                 builder: (context, bloc, _) =>
23                     DashboardWidget(dashboardBloc: bloc,
24                     themeHandler: themeHandler)),);
25     }
26
27     @override
28     State<StatefulWidget> createState() => _DashboardWidgetState();
29 }
30
31 class _DashboardWidgetState extends State<DashboardWidget>
32     with AutomaticKeepAliveClientMixin<DashboardWidget> {
33     @override
34     bool get wantKeepAlive => true;
35
36     @override
37     Widget build(BuildContext context) {
38         super.build(context);
39         return StreamBuilder<Dashboard>(
40             stream: widget.dashboardBloc.stream,
41             initialData: Dashboard(),
42             builder: (context, snapshot) {
43                 return _buildGrid(snapshot.data.toMap());
44             },
45         );
46     }
47
48     Widget _buildGrid(Map<String, dynamic> dashboardData) { ... }
49
50     Widget _buildButtons() { ... }
51
52     Widget _buildSingleBox(Map<String, dynamic> dashboardData,
53     String key) { ... }
54 }
```



Figura 6.9: Screenshot della schermata *Dashboard*.

Settings

Questa schermata presenta una lista delle varie impostazioni che possono essere modificate dall'utente. In questa versione dell'applicazione, possiamo notare tre opzioni:

1. Cache
2. Server
3. Theme

Facendo *tap* su una di queste opzioni, l'utente verrà portato sulla schermata corrispondente. Per accedere alla schermata delle impostazioni, si deve accedere al *drawer*, posto in alto a sinistra dell'applicazione.

Di seguito viene presentato il codice di questa schermata. È possibile notare la mappa **options** (dalla riga 10 alla riga 29) che va a creare delle associazioni tra il nome della schermata, in formato stringa, e l'oggetto che deve essere istanziato nel caso in cui l'utente faccia tap su una di queste opzioni.

Dal codice è possibile notare inoltre la facilità dell'utilizzo dei Widget: nel metodo **build** è presente il Widget **ListView.builder**, il quale permette di iterare su un oggetto *enumerabile*, come ad esempio delle liste. All'interno di tale Widget, viene utilizzato **ListTile**, un componente grafico che permette di realizzare le voci presenti nella schermata *Settings* (Figura 6.10, screenshot a destra). Tutto questo è possibile realizzarlo con poche righe di codice, sfruttando pochi Widget.

```
1 class SettingsPage extends StatefulWidget {  
2     final String title = "Settings";  
3  
4     @override  
5     State<StatefulWidget> createState() => _SettingsPageState();  
6 }  
7  
8 class _SettingsPageState extends State<SettingsPage>  
9     with AutomaticKeepAliveClientMixin<SettingsPage> {  
10     final Map<String, dynamic> options = {  
11         "cache": {  
12             "title": Text("Cache"),  
13             "leading": Icon(Icons.cached),  
14             "subtitle": Text("Delete the data saved from the app"),  
15             "page": CachePage(),  
16         },  
17     },  
18 }
```

```
17     "server": {
18         "title": Text("Server"),
19         "leading": Icon(Icons.perm_data_setting),
20         "subtitle": Text("Choose the server to connect to"),
21         "page": ServerPage(),
22     },
23     "theme": {
24         "title": Text("Theme"),
25         "leading": Icon(Icons.color_lens),
26         "subtitle": Text("Choose the theme of the app"),
27         "page": ThemePage(),
28     },
29 };
30
31 StatefulWidget _getSettingPage(String option) {
32     return options[option] ["page"];
33 }
34
35 @override
36 Widget build(BuildContext context) {
37     super.build(context);
38     return Scaffold(
39         appBar: PageAppBar(context: context, title: widget.title),
40         body: ListView.builder(
41             itemCount: options.length,
42             itemBuilder: (ctx, i) {
43                 return Padding(
44                     padding: const EdgeInsets.only(
45                         left: 20.0,
46                         right: 20.0,
47                         bottom: 25.0,
48                     ),
49                     child: Container(
50                         decoration: BoxDecoration(
51                             color: ColorsPalette.softGrey,
52                             borderRadius: BorderRadius.circular(10.0),
53                         ),
54                         child: ListTile(
55                             leading: options[
56                                 options.keys.toList() [i]] ["leading"],
```

```
57         title: options[  
58             options.keys.toList()[i]]["title"],  
59             subtitle: options[  
60                 options.keys.toList()[i]]["subtitle"],  
61                 trailing: Icon(CupertinoIcons.forward),  
62                 onTap: () async {  
63                     Future<Widget> buildPageAsync() async {  
64                         return Future.microtask(() {  
65                             return _getSettingPage(  
66                                 options.keys.toList()[i]);  
67                         });  
68                     }  
69  
70                     Widget page = await buildPageAsync();  
71                     MaterialPageRoute route =  
72                         MaterialPageRoute(builder: (_) => page);  
73                     Navigator.push(context, route);  
74                 },  
75             ),  
76             );  
77         },  
78         ),  
79     );  
80 );  
81 }  
82  
83 @override  
84 bool get wantKeepAlive => true;  
85 }
```

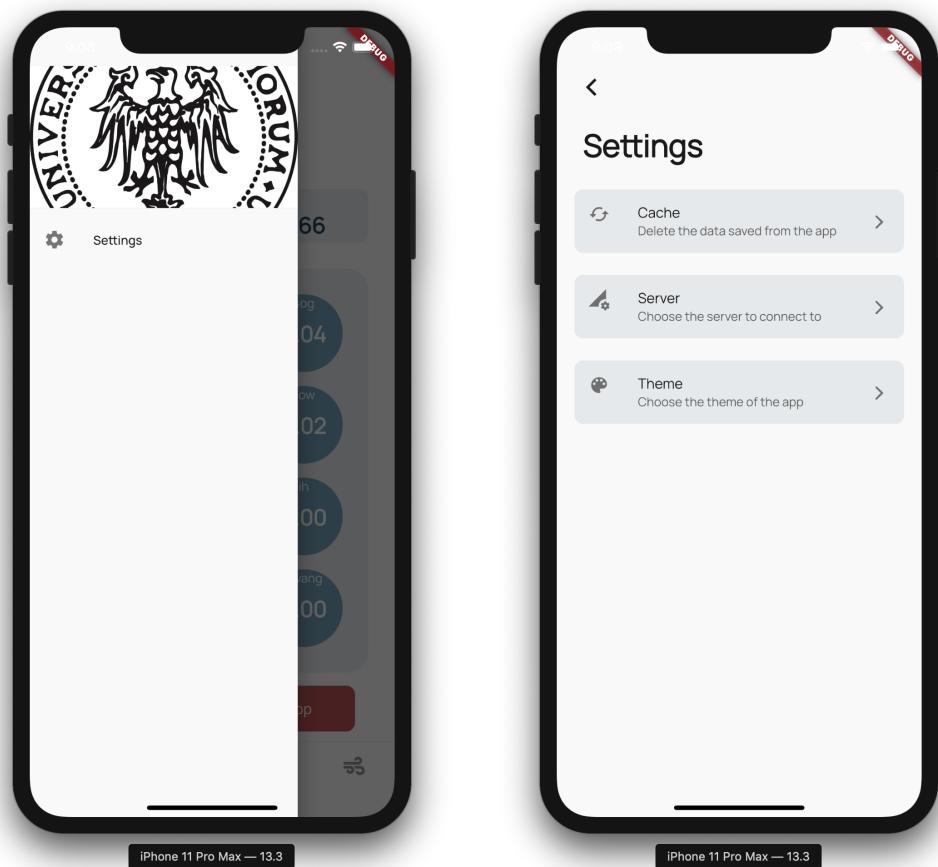


Figura 6.10: Screenshot del *drawer* dell'applicazione (a sinistra) e della schermata di *Settings* (a destra).

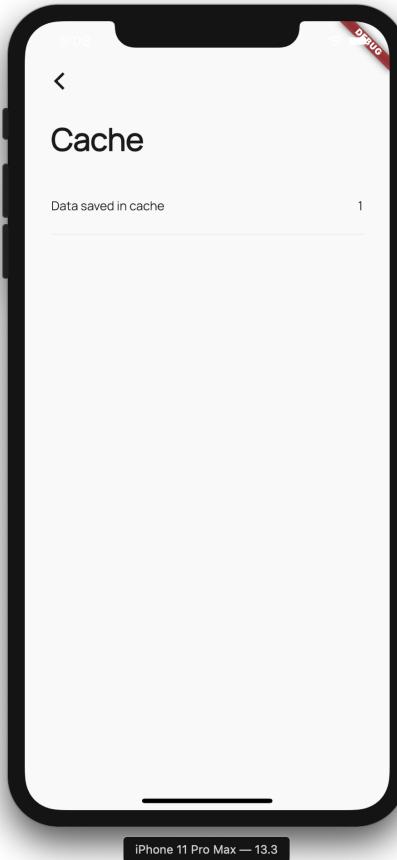


Figura 6.11: Screenshot della schermata *Cache*.

Cache

In questa schermata l'utente può eliminare i dati salvati nella *cache*. Facendo un *tap* sull'unica voce presente nella schermata (Figura 6.11), si presenterà un *Alert Dialog* per chiedere la conferma della pulizia della cache. Se la cache è vuota, anche se si volesse selezionare la voce, non comparirà alcuna Alert Dialog.

Dagli screenshot in Figura 6.12, è possibile notare che sono state riprodotte le Alert Dialog sia per i dispositivi con sistema operativo Android che iOS. Le diverse Alert Dialog sono fornite direttamente da Flutter.

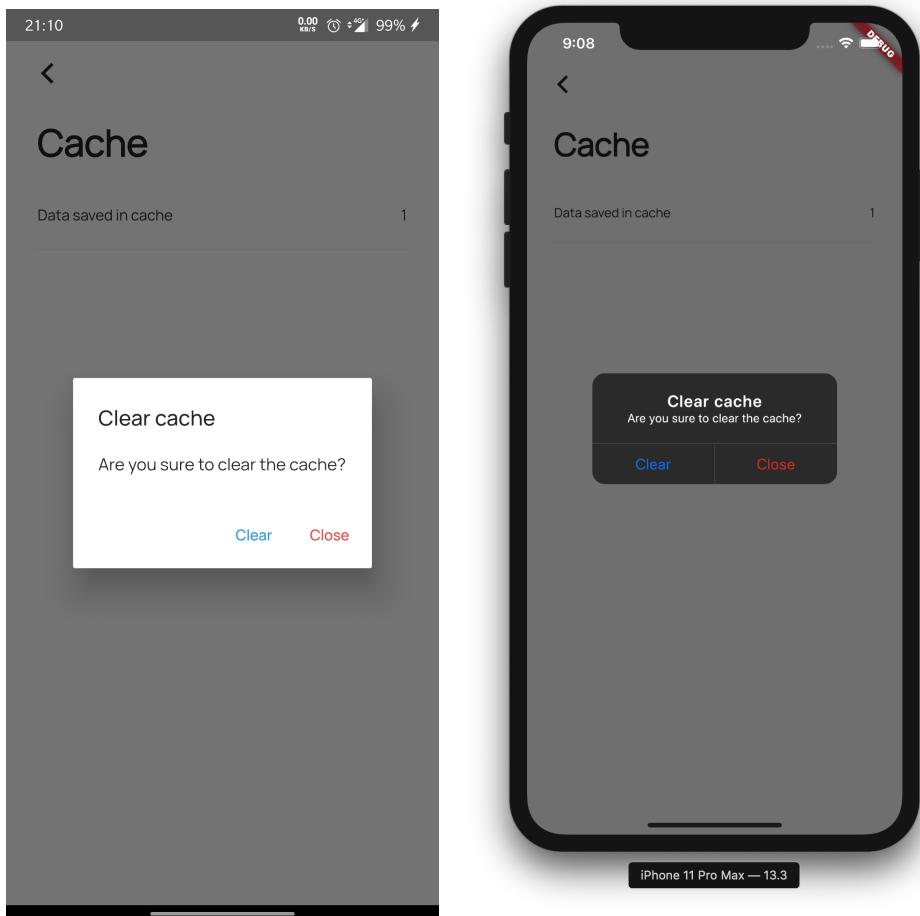


Figura 6.12: Screenshot dell'*Alert Dialog* per dispositivi Android (a sinistra) e iOS (a destra).

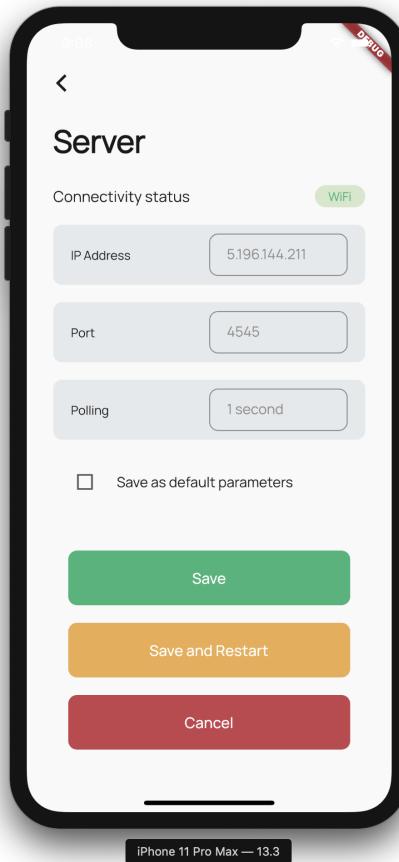


Figura 6.13: Screenshot della schermata *Server*.

Server

Questa schermata offre due principali funzionalità:

1. Controllo dello stato della *connessione*;
2. Modifica dei parametri per la connessione al server e per il *polling*.

Per quanto riguarda la prima funzionalità, quando viene rilevato un cambio di stato della connessione, ad esempio da una rete *WiFi* ad una rete *mobile*, questo viene reso noto dalla componente grafica evidenziata negli screenshot presenti nella Figura 6.14. Nel caso in cui sia assente la connessione ad Internet, viene mostrata la dicitura *"offline"*. Questa funzionalità è stata possibile implementarla grazie all'utilizzo di un particolare plugin che offre il team di

Flutter: **connectivity** (l'applicazione si basa sulla versione 0.4.8+2 del plugin) [41]. Questo pacchetto fornisce una serie di metodi che permettono di controllare lo stato della connessione.

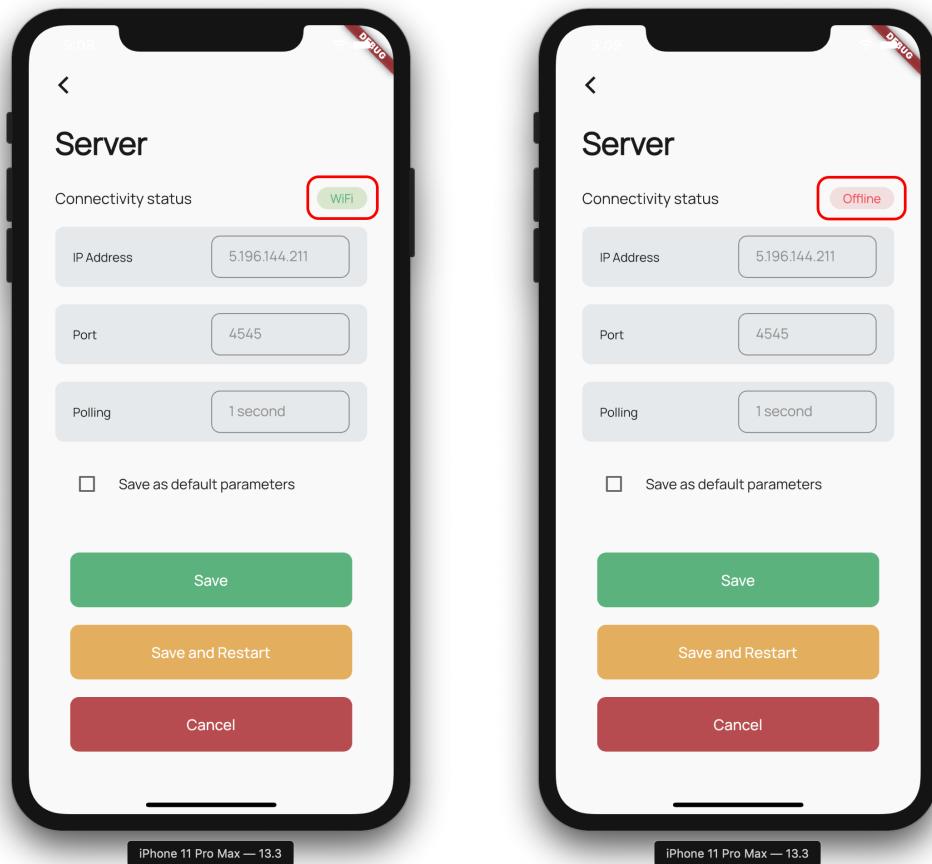


Figura 6.14: Tramite un apposito ascoltatore ed il pacchetto (*connectivity 0.4.8+2* [41]), l'utente può verificare lo stato della connessione Internet.

Dalla riga 8 fino alla riga 15, viene inizializzato un ascoltatore che appena rileva un cambio di stato della connessione, aggiorna lo stato dell'applicazione (nel metodo `setState(() {})`). In questo pezzo di codice, viene sfruttata appieno la *programmazione reattiva*: appena il contenuto della variabile privata `_connectivityStatus` cambia, questo viene propagato nell'albero dei Widget, senza dover eseguire ulteriori istruzioni.

```
1 ...
2
3 @override
4     void initState() {
5         super.initState();
6         getInitialStatus();
7
8         _subscription = Connectivity()
9             .onConnectivityChanged
10            .listen((ConnectivityResult result) {
11                setState(() {
12                    _connectivityStatus = getTextStatus(result);
13                });
14            });
15        }
16
17     void getInitialStatus() async {
18         var connectivityResult = await (
19             Connectivity().checkConnectivity());
20         setState(() {
21             _connectivityStatus = getTextStatus(connectivityResult);
22         });
23
24     ...
25 }
```

Per quanto riguarda la funzionalità di modifica dei parametri, quando i dati inseriti rispettano i formati e i vincoli richiesti, è possibile notare nella parte inferiore dell'applicazione una *SnackBar*. Lo scopo della *SnackBar* è quello di notificare l'utente che i dati sono stati inseriti correttamente e che l'aggiornamento dei nuovi parametri inseriti è avvenuto con successo (Figura 6.15). Nel caso in cui invece l'utente abbia inserito dei dati non corretti dal punto di vista del formato o dei vincoli imposti, viene visualizzata una *AlertDialog* (Figura 6.16). Se vengono generati più errori in un'unica richiesta, questi verranno mostrati tutti insieme in una sola *AlertDialog*.

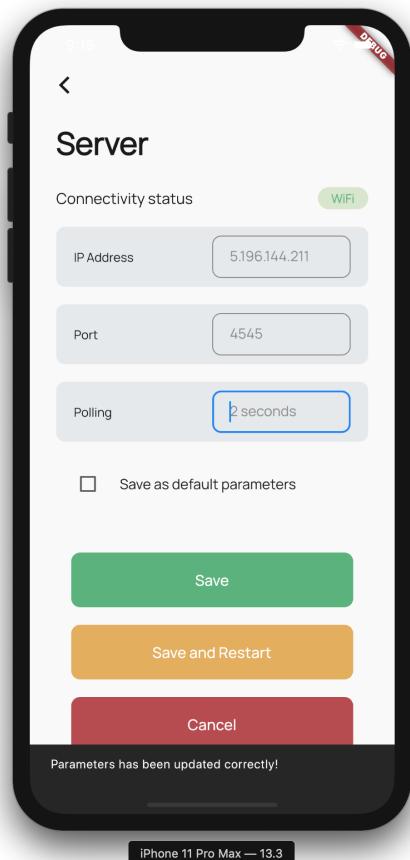


Figura 6.15: È possibile notare la *SnackBar* che notifica che i dati sono stati inseriti correttamente e che sono stati aggiornati.

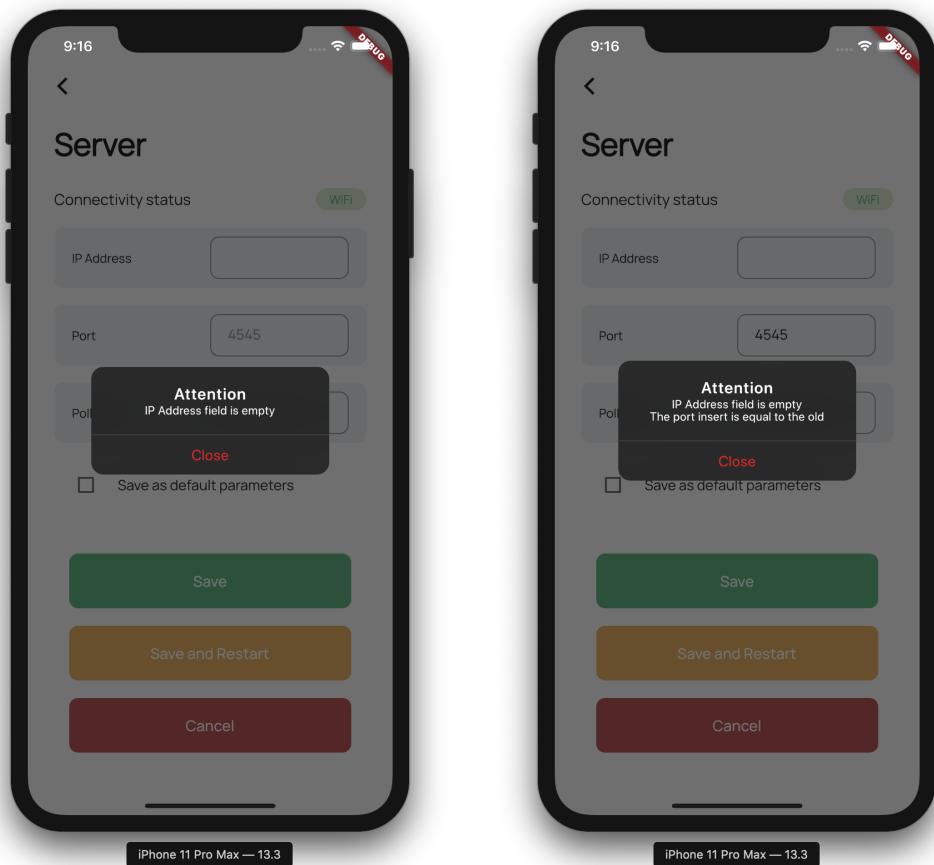


Figura 6.16: Quando vengono inseriti dei parametri in un formato non accettato, viene visualizzata una *Alert Dialog*.

Temi

In questa schermata, l'utente può selezionare uno dei temi forniti dall'applicazione. In questa versione dell'app sono presenti soltanto due temi: *default* e ad *alto contrasto*. È possibile attivare o disattivare l'attivazione del tema ad *alto contrasto* tramite l'apposito *switch* (Widget Switch). Quando il tema viene attivato o disattivato, anche il testo dell'opzione cambia. Osservando la Figura 6.17, nella schermata di sinistra il tema ad *alto contrasto* è attivato, mentre a destra è disattivato.

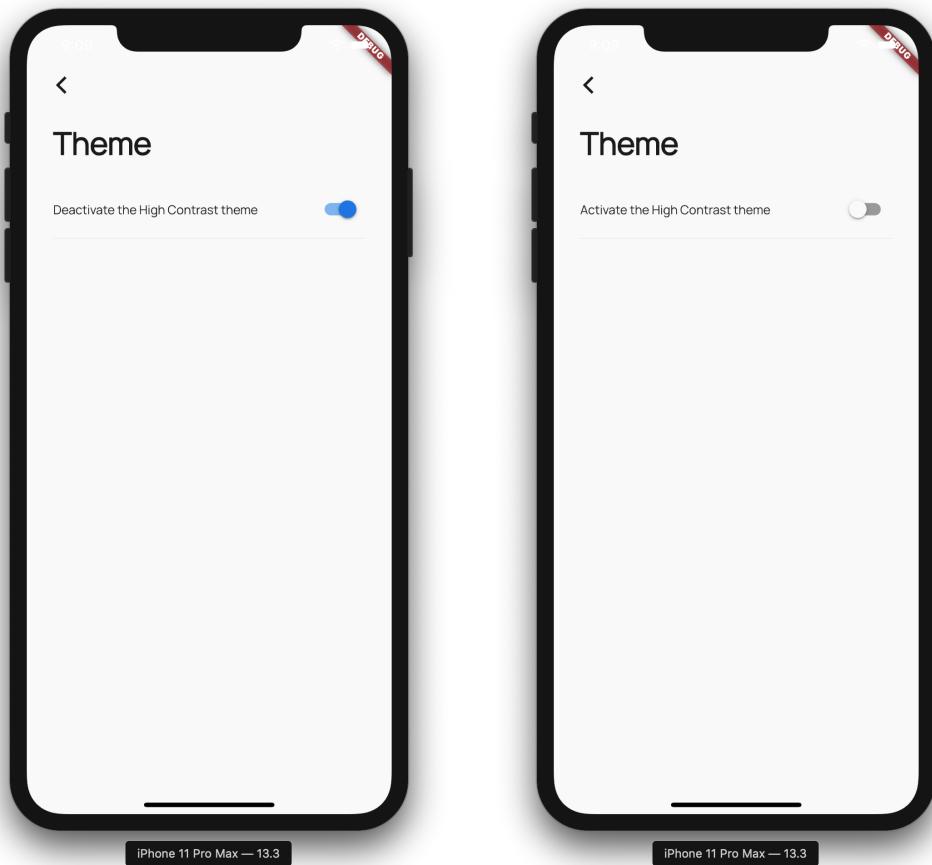


Figura 6.17: Screenshot della schermata *Theme*.

È possibile vedere dagli screenshot in Figura 6.18, come si presenta l'applicazione con il tema ad *alto contrasto* attivato. Le schermate *GPS* e *Wind* hanno il medesimo aspetto della schermata *Navigation*.

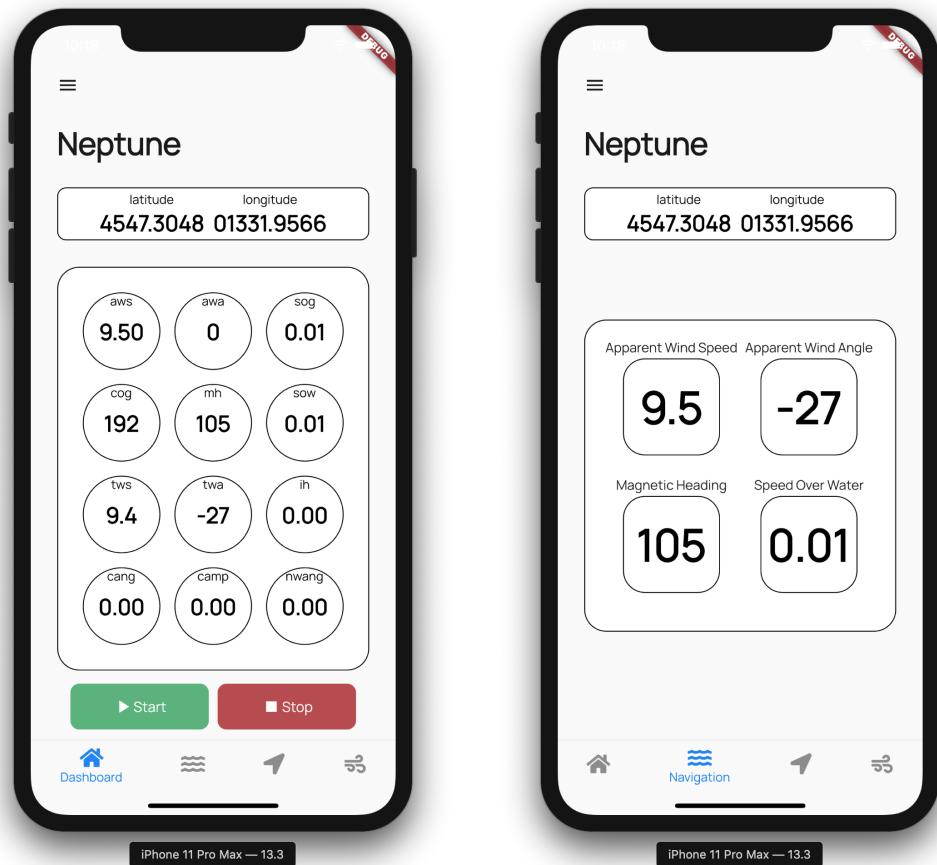


Figura 6.18: È possibile notare le schermate *Dashboard* e *Navigation* (in ordine da sinistra) con l'applicazione ad *alto contrasto*.

6.5.7 Widget comuni

L'interfaccia grafica è stata realizzata utilizzando dei Widget creati *ad hoc*. Grazie a Flutter e alla flessibilità dell'architettura implementata, questi Widget possono essere aggiunti facilmente in una schermata e risulta molto semplice integrarli, a livello di codice.

GridBox

Questo Widget viene utilizzato dalle schermate *Navigation*, *GPS* e *Wind* (Figura 6.19). È un Widget che costruisce una griglia quadrata 2×2 , in cui mostrare i dati da monitorare. Gli elementi che compongono la griglia sono stati costruiti per composizione con altri Widget elementari e la loro struttura risulta essere complessa. Tuttavia, la semplicità con cui questo Widget può essere integrato nelle varie schermate rappresenta un grande vantaggio. Tutta la complessità dell'implementazione della griglia è incapsulata all'interno della classe `GridBox`. Il codice che viene proposto, è stato preso dalla schermata *Navigation*. Dalla riga 11 alla riga 15 è possibile notare la semplicità con cui viene effettuata la chiamata al Widget `GridBox`, passando i relativi parametri, necessari per istanziare correttamente il Widget con dei valori iniziali e per gestire gli aggiornamenti futuri.

```
1 ...
2
3 class _NavigationPageState extends State<NavigationPage>
4     with AutomaticKeepAliveClientMixin<NavigationPage> {
5     @override
6     bool get wantKeepAlive => true;
7
8     @override
9     Widget build(BuildContext context) {
10         super.build(context);
11         return GridBox(
12             bloc: widget.navigationBloc,
13             themeHandler: widget.themeHandler,
14             initialData: Navigation(),
15         );
16     }
17 }
18 ...
19 ...
```



Figura 6.19: Screenshot del Widget *GridBox*. Lo stesso Widget è stato integrato nelle schermate *Navigation*, *GPS* e *Wind*.

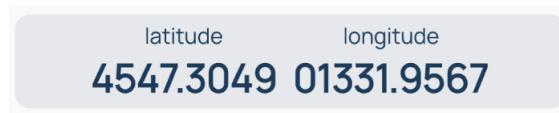


Figura 6.20: Screenshot del Widget *Location*. Lo stesso Widget è stato integrato in tutte e quattro le schermate.

Location

Questo Widget permette di visualizzare i dati relativi alla *geolocalizzazione* (Figura 6.20). Il Widget *Location* è presente in tutte le quattro schermate dell'applicazione. È posto nella parte superiore dello schermo, in modo che l'utente possa monitorare continuamente i parametri visualizzati, anche quando passa da una schermata all'altra.

Il codice che verrà illustrato, è stato preso dalla classe *DashboardPage*. Per poter utilizzare questo Widget bisogna effettuare una chiamata al metodo *create* di *Location* (riga 7). Anche in questo caso, è possibile notare la semplicità con cui i Widget, anche se personalizzati e possiedono una loro logica, possono essere integrati in varie parti dell'applicazione. Infatti, il Widget ha un suo BLoC per interagire con il Repository.

```
1 ...
2 Widget _buildUI() {
3     return ListView(
4         shrinkWrap: true,
5         children: <Widget>[
6             LocationWidget.create(context),
7             DashboardWidget.create(context),
8         ],
9     );
10 }
11 ...
```

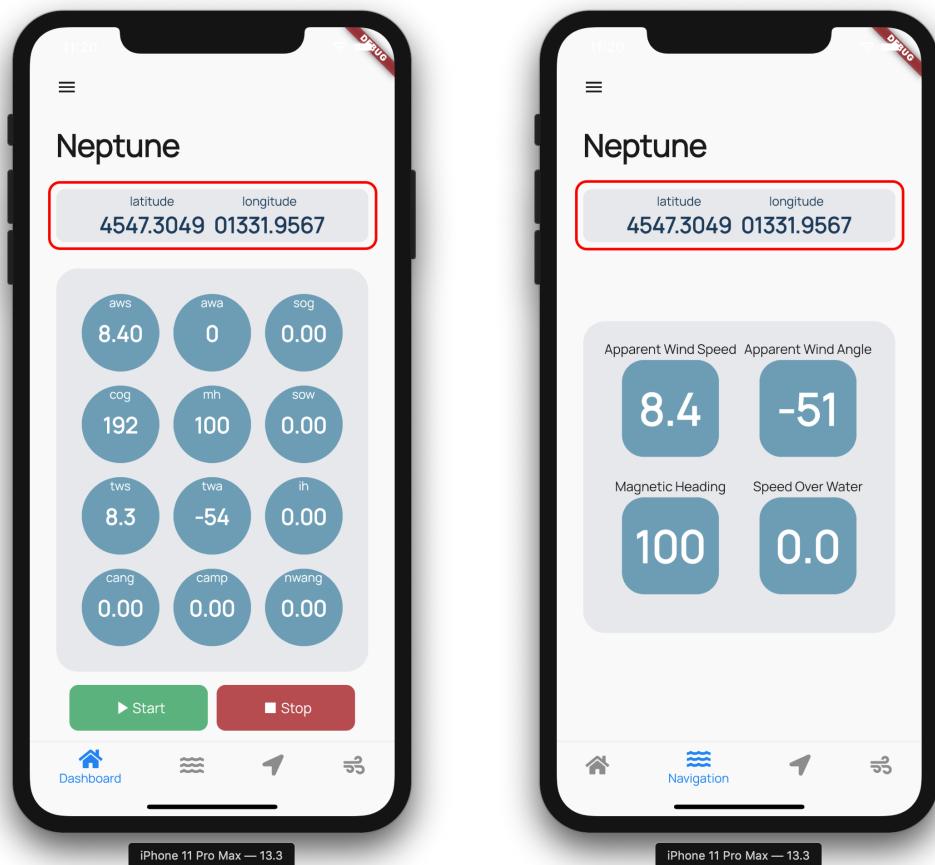


Figura 6.21: Il Widget *Location* nelle schermate *Dashboard* (a sinistra) e *Navigation* (a destra).

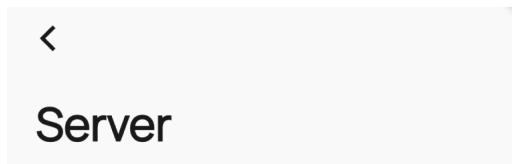


Figura 6.22: Screenshot del Widget *Page App Bar*. Lo stesso Widget è stato integrato principalmente nelle schermate relative alle impostazioni.



Figura 6.23: Screenshot del Widget *Sensor Data App Bar*. Lo stesso Widget è stato integrato in tutte le schermate che permettono di visualizzare i dettagli di un particolare dato.

Page App Bar

Questo Widget va a modificare l'*App Bar* di base fornita da Flutter. Questa *App Bar* viene utilizzata principalmente nelle schermate relative alle impostazioni, quindi nelle schermate *Settings*, *Cache*, *Server* e *Theme*. Lo screen della Figura 6.22 è stato preso dalla schermata *Server*. La flessibilità di questo Widget è data dalla possibilità di modificare il titolo dell'*App Bar*, in quanto può essere utilizzata, appunto, su molteplici schermate.

Sensor Data App Bar

Sensor Data App Bar è un Widget molto simile al precedente (Figura 6.23). Le differenze sostanziali riguardano principalmente gli aspetti grafici. Questo Widget ha la medesima flessibilità del precedente. **Sensor Data App Bar** viene utilizzato nelle schermate dedicate ai dettagli di un dato preciso. Questa funzionalità non è stata implementata, ma ne è stato predisposto l'ambiente.

6.5.8 Palette di colori

Questa classe fa parte del *package utils* e raccoglie tutti i colori personalizzati che sono stati utilizzati nell'applicazione.

Uno dei principali motivi della realizzazione di quest'applicazione era la creazione di un'interfaccia che potesse presentare i dati in maniera più efficace, rivisitando l'aspetto grafico. Pertanto, l'implementazione di questa classe ha un valore importante all'interno del progetto, nonostante non contenga una complessità paragonabile ad altri elementi dell'app.

Di seguito viene mostrato un breve esempio del contenuto di questa classe.

```
1 ...
2
3 // Red
4 static Color redAccent = Color.fromRGB(255, 87, 87, 1);
5 static Color redBackground = Color.fromRGB(198, 35, 38, 0.1);
6 static Color redButton = Color.fromRGB(178, 34, 34, 0.7);
7
8 ...
```

6.5.9 Temi

Questa classe è collocata all'interno del package **utils** e permette di gestire il tema dell'applicazione. L'applicazione ha principalmente due temi:

1. **default**: è il tema che si presenta all'apertura dell'applicazione;
2. **alto contrasto**: è un tema che utilizza soltanto il bianco ed il nero.
Questo tema mette in evidenza gli elementi dell'applicazione soprattutto quando il tempo metereologico è soleggiato.

Themes

Questa è una semplice classe *enumerazione* in cui vengono elencati i vari nomi dei temi che supporta l'applicazione. Se in futuro verranno realizzati ulteriori temi, sarà necessario aggiornare questa classe.

```
1 enum Themes {
2     DEFAULT,
3     HIGH_CONTRAST
4 }
```

CustomTheme

`CustomTheme` è un'*interfaccia* (anche se viene dichiarata come una classe astratta, si veda l'Appendice A per ulteriori spiegazioni) che fornisce dei metodi comuni per implementare tutti i temi. Di seguito viene presentato l'elenco dei metodi:

1. `Color getBackgroundColorOfSensorData()`
2. `Color getBorderColorOfSensorData()`
3. `Color getBackgroundColorOfBox()`
4. `Color getBackgroundBorderOfBox()`
5. `Color getTextOfSensorValues()`
6. `Color getTitleTextOfSensorData()`
7. `Color getTitleTextOfSensorDataInDashboard()`
8. `Color getTextColorOfLocation()`

È possibile notare che questi metodi vanno ad operare su delle specifiche componenti grafiche o delle parti di essa. Questa interfaccia può essere ampliata per aumentare la personalizzazione grafica dell'applicazione.

DefaultTheme e HighContrastTheme

Queste due classi implementano i metodi dell'interfaccia `CustomTheme`. Ciascuna classe va ad applicare dei specifici colori per realizzare un preciso tema. Il `DefaultTheme` utilizza diversi colori, mentre il `HighContrastTheme` utilizza soltanto i colori bianco e nero.

ThemeHandler

Anche questa classe implementa l'interfaccia `CustomTheme`, tuttavia non implementa alcun tema. Questa classe, invece, si occupa di applicare il tema, una volta che è stato scelto dall'utente. Infatti, tramite il metodo `applyTheme` è possibile indicare quale tema si vuole applicare. Il metodo in questione, sulla base del parametro ricevuto, istanzia un oggetto che rappresenta il tema scelto e lo applica all'app. Per poter attuare realmente il cambio di tema, questo metodo deve essere chiamato all'interno del metodo `setState(() {})`,

in modo da notificare il framework. Così facendo, il framework rileva un cambiamento di stato dell'applicazione e attua le dovute modifiche al Widget tree, visualizzando il tema scelto dall'utente.

Di seguito si mostra l'implementazione del metodo che permette di applicare il tema.

```
1 void applyTheme(Themes themeChosen) {
2     switch (themeChosen) {
3         case Themes.DEFAULT:
4             _themeChosen = Themes.DEFAULT;
5             theme = DefaultTheme();
6             break;
7         case Themes.HIGH_CONTRAST:
8             _themeChosen = Themes.HIGH_CONTRAST;
9             theme = HighContrastTheme();
10            break;
11        }
12    }
```

6.5.10 Main

Nel file `main.dart` sono contenute tutte le classi fondamentali per il funzionamento dell'applicazione. La struttura iniziale di questo file viene fornita dal framework stesso nel momento in cui viene creato un nuovo progetto in Flutter.

Con la definizione del metodo `void main() => runApp(MyApp())`, viene fatta eseguire l'applicazione, ovvero, viene avviato tutto il processo di inizializzazione delle varie task e del motore di rendering per visualizzare la grafica.

In questo file viene implementata la struttura essenziale su cui si poggia tutta l'applicazione. In particolare:

1. Un `FutureBuilder` che va leggere i dati dal file JSON, salvato nella memoria locale del dispositivo, contenente i dati relativi alla connessione al server e al polling;
2. Il Provider per la gestione dei *temi* (`Provider<ThemeHandler>`);
3. `MaterialApp` è un *core* Widget che permette di utilizzare tutti i componenti forniti dal framework. In questo Widget vengono anche definite le caratteristiche basilari dell'applicazione come:
 - (a) Il titolo;

- (b) Il colore principale;
- (c) Il nome dell'applicazione che verrà visualizzato nel menù delle app del dispositivo;
- (d) I Widget che andranno a realizzare l'applicazione nel parametro `home`.

Vengono definiti inoltre degli altri Widget importanti per la struttura essenziale dell'applicazione, come:

1. Un'App Bar personalizzata con il nome dell'applicazione;
2. Una `BottomNavigationBar` che va a definire una barra di navigazione fissa per tutte le schermate principale, posta nella parte inferiore dell'applicazione. Tramite questo Widget, è possibile navigare da una schermata all'altra tra *Dashboard*, *Navigation*, *GPS* e *Wind*;
3. Un `Drawer`, ovvero, un menù laterale che può essere visualizzato facendo uno *swipe* da sinistra a destra, in cui viene illustrata il logo dell'Università degli Studi di Udine e le opzioni;
4. Il `body` dell'applicazione in questo caso è una `PageView`. La `PageView` è un Widget che permette di organizzare e di navigare tra le varie schermate, scorrendo a destra e a sinistra. Tramite questo Widget viene offerta un ulteriore modalità per navigare tra le schermate, oltre a quella già fornita dalla `BottomNavigationBar`. Nella `PageView`, vengono effettuate tutte le chiamate alle classi delle principali schermate.

6.6 Motivazioni delle scelte grafiche implementate

6.6.1 Concetti generali

Uno degli obiettivi principali della realizzazione di questa applicazione è la completa rivisitazione dell'interfaccia grafica, cercando di sopprimere ai difetti strutturali dell'applicazione Web e dell'applicazione Android. Pertanto, il `layout` dell'applicazione doveva essere particolarmente curato, in modo da comunicare visivamente all'utente una facile comprensione dell'organizzazione degli elementi. Questo aspetto è di fondamentale importanza, in quanto essendo la *competizione* (la regata) il contesto di applicazione di questo software, è assolutamente necessario che l'utente sia in grado di visualizzare immediatamente i dati di cui necessita. Trovandosi in situazioni in cui si devono svolgere manovre

di una certa complessità, la squadra deve concentrarsi nel corretto svolgimento di tali manovre, apportando una minore attenzione all'applicazione. L'applicazione deve aiutare la squadra nel rendere l'attività di monitoraggio dei dati il più semplice e fruibile possibile. L'utente non può impiegare troppo tempo nel capire come funziona l'applicazione o nel capire dove deve andare per poter monitorare un particolare dato: il layout deve essere **intuitivo**. Se così non fosse, l'utente perderebbe troppo tempo a reperire le giuste informazioni e a capire il suo funzionamento, sprecando così del tempo utile per indirizzare correttamente la barca nel percorso prestabilito. Se invece l'applicazione è intuitiva, la squadra ne trarrà vantaggio, in quanto verrà utilizzata attivamente come supporto alle varie decisioni che vengono prese durante tutta la competizione.

6.6.2 Scelte stilistiche attuate

Per ottenere un'interfaccia grafica pulita e facile da utilizzare, nella maggior parte delle schermate si è fatto uso delle *griglie*. Questo elemento grafico va a disporre gli elementi secondo un certo ordine. Tuttavia, la griglia non va a coprire l'intero schermo del dispositivo, ma va coprire la maggior parte dello spazio dello schermo: sia la griglia che il *Location Widget* sono contenuti all'interno di un box di colore grigio. Il significato è quello di delimitare lo spazio di un determinato elemento, da tutto il resto dell'applicazione. In questo modo l'utente, se deve osservare un determinato dato, sa su quale gruppo di elementi deve andare a porre attenzione.

Inoltre, anche gli *spazi bianchi* tra un elemento e l'altro contribuiscono nell'aspetto e nella percezione del layout. Quando si va a sviluppare un layout, si cerca di ottimizzare tutto lo spazio che si ha a disposizione. Questo è un errore che porta alla realizzazione di un'interfaccia grafica molto complessa e caotica. La presenza di molti elementi che possono cambiare forma o contenuto nel tempo (come nel caso dell'applicazione realizzata), può comportare ad un senso di disorientamento e di confusione all'utente. L'utente può trovare difficoltà nel reperire le informazioni corrette con velocità. Riempire lo schermo di elementi grafici implica un **sovraffollamento cognitivo** da parte dell'utente. Questo fenomeno si verifica quando l'utente riceve troppe informazioni e non è in grado di prendere una decisione o di concentrarsi su un'informazione specifica. Se il layout realizzato presenta queste criticità, è necessario riprogettare interamente l'interfaccia, in quanto l'utilizzo di un'applicazione tale, comporterebbe ad ottenere una pessima *user experience*. Nel contesto in cui viene utilizzata l'app, questo potrebbe esporre la squadra anche a dei pericoli: se l'utente non è in grado di focalizzare la sua attenzione sui dati di cui necessita, l'utente non sarà attento né alle manovre della barca, né ai dati forniti dell'applicazione.

ne, che potrebbero avvisarlo, ad esempio, di un qualche malfunzionamento del mezzo.

In conclusione, l'organizzazione degli elementi grafici è stata pensata per la realizzazione di un layout che fosse il più pulito ed ordinato possibile, in modo che l'utente possa andare a reperire velocemente i dati, leggerli e confrontarsi con la squadra sulle decisioni da prendere.

6.7 Ottimizzazioni

In questa sezione, si andranno a descrivere le buone pratiche di programmazione in Flutter e le ottimizzazioni attuate nell'applicazione realizzata, per ottenere dei vantaggi a livello di prestazioni e per offrire all'utente un'interfaccia grafica più fluida e reattiva.

6.7.1 Provider e StreamBuilder

Per ottenere delle migliori dal punto di vista delle prestazioni, è necessario *abbassare* i Provider e gli StreamBuilder nell'albero dei Widget. Se questi Widget vengono posizionati vicini alla radice, nel momento in cui avviene il cambiamento di stato, tutti i loro sotto-alberi vengono aggiornati dal motore di rendering. Quindi, c'è molta probabilità che in una situazione simile, diversi Widget vengano aggiornati anche quando non lo necessitano, sia perché questi sono *immutabili* o perché semplicemente devono essere aggiornati solo alla ricezione di determinati dati. Risulta *efficiente* dal punto di vista delle prestazioni, posizionare nel punto più basso possibile questi Widget: così facendo, nel momento in cui vengono ricevuti dei dati, vengono aggiornati soltanto i Widget strettamente essenziali. Questa considerazione può sembrare futile, tuttavia è un tema molto importante da considerare dopo aver sviluppato una prima versione del software. Questa problematica può essere fonte di rallentamenti dell'applicazione, in particolare dell'interfaccia grafica. Di conseguenza l'esperienza utente diventa più difficoltosa e spiacevole.

6.7.2 Annidamento degli StreamBuilder

Un altro aspetto molto importante da considerare nella fase di sviluppo dell'applicazione riguarda l'*annidamento* degli StreamBuilder. Come suggerisce il nome, questi Widget sono direttamente collegati a degli **Stream** e nel momento in cui viene ricevuto un nuovo dato tramite questo flusso, il sotto-albero dello **StreamBuilder** (specificato nel parametro **child**) viene aggiornato dal

motore di rendering. Pertanto, annidare più `StreamBuilder` tra loro comporta principalmente a:

1. **Aggiornamenti a cascata:** quando uno `StreamBuilder` riceve un nuovo dato, provoca l'aggiornamento di tutti i Widget sottostanti, compresi gli `StreamBuilder` inclusi nel suo sotto-albero. Così facendo, vengono aggiornati a catena tutti gli `StreamBuilder`, provocando diverse chiamate al motore di rendering. Di conseguenza, l'interfaccia risulterà più lenta e meno reattiva alle interazioni dell'utente e farà uno uso maggiore delle risorse del dispositivo;
2. **Aggiornamenti non voluti:** un *effetto collaterale* del punto precedente, è che alcuni Widget vengono forzatamente aggiornati anche quando non lo necessitano. Questa casistica è analoga a quella descritta per i Provider.

6.7.3 Ottimizzazioni grafiche

Per ottenere delle ottimizzazioni delle prestazioni nell'interfaccia grafica, è stato utilizzato il metodo statico `Future.microtask(() { ... })`. Flutter, oltre ad avere un Event Loop (Capitolo 4) in cui inserire tutte le *task* che il framework deve elaborare, ha una seconda coda gestita con politica *FIFO*: **Microtask Queue**. Questa coda viene dedicata allo svolgimento di quelle task che possono essere elaborate e terminate velocemente. Se delle task vengono inserite in questa coda, il framework va ad eseguire prima queste e poi quelle contenute nella *Event Queue* [42].

Nell'applicazione, questo meccanismo è stato utilizzato per velocizzare e rendere più fluida l'apertura delle schermate relative ai dati dei sensori e alle impostazioni. In questo modo, quando l'utente vuole andare a modificare le impostazioni della connessione del server, per esempio, il tap dell'utente viene captato e l'evento viene inserito nella *microtask queue*. Il framework si accorge che in questa coda c'è un evento da elaborare, esegue l'elaborazione e torna a controllare se ci sono degli eventuali eventi nella medesima coda o nelle coda degli eventi. Così facendo, l'apertura delle schermate sarà più reattiva.

Il codice illustrato di seguito, è stato preso dalla classe `GridBox`. Dalla riga 7 alla riga 18 è possibile notare l'utilizzo del metodo `Future.microtask()` per l'apertura di una nuova schermata.

```
1 ...
2
3 Widget _setupBox(String title, String value) {
4     return InkWell(
5         onTap: () async {
6             Future<Widget> buildPageAsync() async {
7                 return Future.microtask(() {
8                     return Scaffold(
9                         appBar: SensorDataAppBar(context: context,
10                         title: title),
11                         body: Column(
12                             children: <Widget>[
13                                 Center(
14                                     child: _buildBox(title, value),
15                                 ),
16                             ],
17                         ),
18                     );
19                 });
20             }
21
22             Widget page = await buildPageAsync();
23             MaterialPageRoute route = MaterialPageRoute(
24                 builder: (_) => page);
25             Navigator.push(context, route);
26             },
27             child: _buildBox(title, value),
28         );
29     }
30
31 ...
```

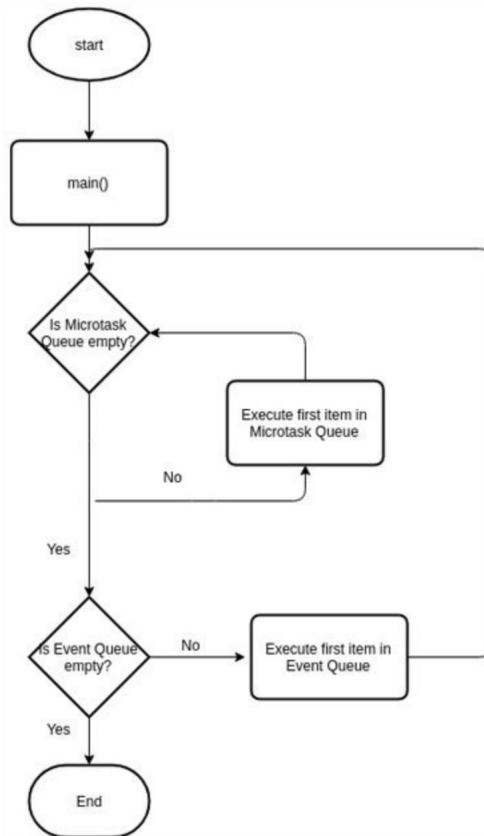


Figura 6.24: Uno schema che illustra il funzionamento del meccanismo dell'*Event Queue* e dell'*Microtask Queue*, durante l'esecuzione dell'applicazione [43].

Capitolo 7

Conclusioni

7.1 Sviluppi futuri

L'applicazione realizzata ha voluto includere le funzionalità essenziali per poter essere di supporto durante la competizione. Alcune funzionalità sono state aggiunte per poter aumentare la flessibilità dell'applicazione, in modo che possa adattarsi in base alle esigenze: ad esempio, la possibilità di determinare ogni quanti secondi effettuare una connessione al server per ricevere i dati. Inoltre è stata realizzata una grafica ordinata e pulita, in modo da poter individuare immediatamente i dati necessari. Oltre agli aspetti visivi, la gran parte del lavoro è stato fatto nel *background*, ovvero, nella strutturazione di un'architettura che fosse solida e flessibile per poter supportare in futuro delle nuove funzionalità. Pertanto in questa sezione si introducono dei possibili sviluppi futuri, di cui alcuni sono già stati impostati.

7.1.1 Notifiche push

Le *notifiche push* sono una funzionalità peculiare dei dispositivi mobili. Nel contesto d'uso, le notifiche push potrebbero essere utili nel caso in cui si volesse attirare l'attenzione dell'utente per segnalare un determinato evento. Si assuma di trovarsi nel caso in cui l'utente stia monitorando un certo set di dati da una determinata schermata. Se l'applicazione inviasse una notifica push per segnalare, ad esempio, un aumento improvviso della velocità apparente del vento, l'utente se ne accorgerebbe e potrebbe andare nell'apposita schermata per analizzare il dato e verificare se l'avviso ricevuto può essere considerato rilevante o meno.

L'utilizzo delle notifiche push potrebbe essere esteso anche per altre funzionalità in quanto è uno strumento molto flessibile e molto utile.



Figura 7.1: Facendo tap sul dato *aws*, si aprirà la schermata di destra.

7.1.2 Grafici e schermate dedicate ai dati

Una funzionalità molto importante che aiuterebbe l’equipaggio nelle varie decisioni è la realizzazione di una pagina dedicata alla descrizione del dato. Questa funzionalità è stata in parte implementata, ovvero, è già stato predisposto il meccanismo per rilevare l’evento del tocco su un determinato dato e l’apertura della corrispondente schermata. Come si può vedere dagli screenshot, nella fase finale dell’applicazione le schermate sono vuote. Questa funzionalità è già stata predisposta per le schermate *Dashboard*, *Navigation* e per il Widget *Location*. In queste schermate potrebbero essere illustrate delle descrizioni specifiche del dato ed eventualmente uno storico degli eventi riguardo a tale dato. In particolare, per *eventi* si vuole intendere il verificarsi di una forte variazione del valore di un determinato dato, in cui tale variazione risulta essere anomala e può quindi essere motivo di attenzione.

Nella medesima schermata potrebbe essere aggiunto un grafico che illustra tutte le variazioni del dato nel tempo. Il vantaggio di utilizzare questo strumento, permette all’utente di visualizzare in maniera immediata se vi sono state delle variazioni anomale. I grafici possono sfruttare la *cache* che è stata implementata nel *Repository* dell’app. In questo modo, il grafico può attingere i dati da tale fonte.

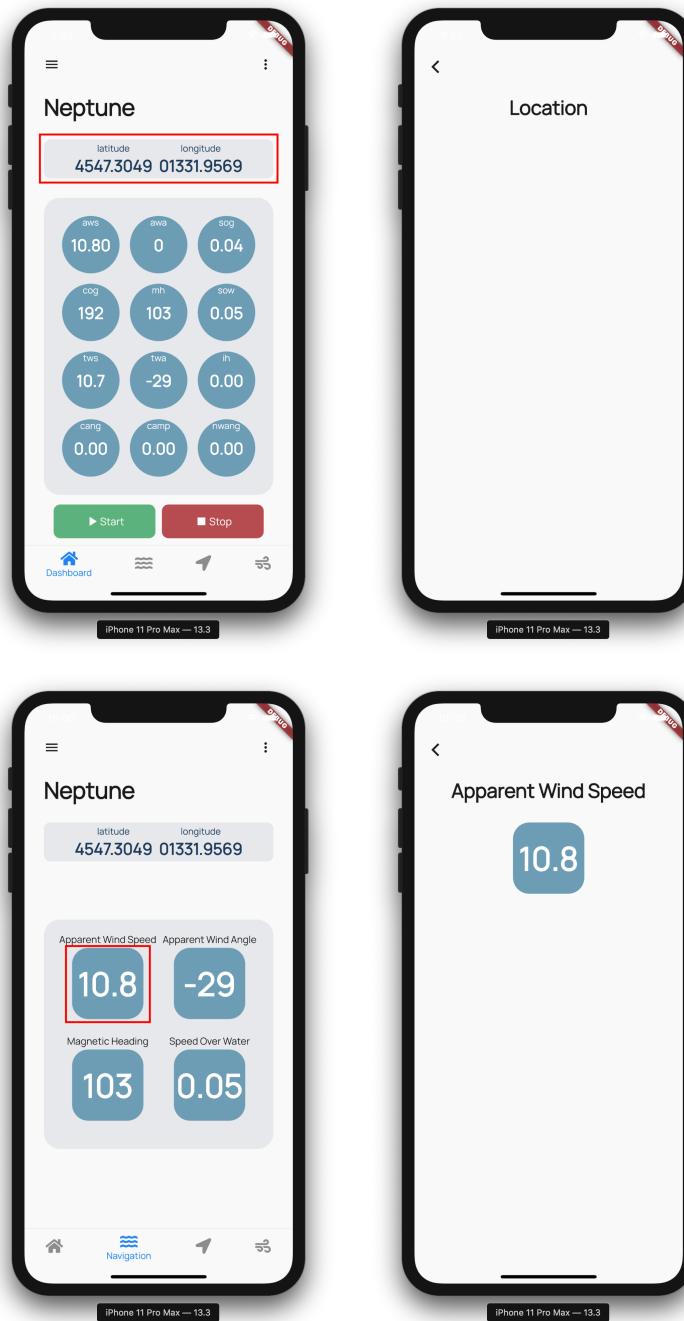


Figura 7.2: Analogamente per la schermata *Dashboard*, lo stesso avviene sia per il Widget *Location*, sia per la schermata *Navigation* (facendo tap sul dato *Apparent Wind Speed*)

7.1.3 Aggiunta di componenti grafiche

Per facilitare la fruizione delle informazioni proposte dall'applicazione, sarebbe utile l'implementazione di alcuni componenti grafici dedicati. Ad esempio, potrebbe essere realizzata una *bussola*, con l'utilizzo del giroscopio e dell'accelerometro. La bussola può essere utilizzata per rendersi conto della direzione attuale verso la quale la barca si sta dirigendo.

7.1.4 Introduzione di nuovi temi

L'applicazione così realizzata permette di avere soltanto due temi: *default* e ad *alto contrasto*. Per come è stata strutturata l'applicazione è possibile supportare soltanto due temi. Con delle piccole variazioni al codice, è possibile gestire più temi, in modo che l'utente possa scegliere opportunatamente quello che ritiene più adatto alla situazione in cui si trova.

7.1.5 Autenticazione degli utenti

Aggiungere un servizio di autenticazione degli utenti potrebbe essere utile per capire quali membri della squadra o del team tecnico accedono ai dati e a quali dati. In questo modo è possibile creare un servizio che sia più completo anche dal punto di vista della sicurezza.

7.2 Considerazioni sull'approccio cross-platform

Negli ultimi anni, sempre più aziende cominciano a considerare l'approccio cross-platform come il più conveniente secondo diversi punti di vista. Come è già stato esplicitato durante la tesi, questo approccio è un ottimo compromesso tra *performance*, in quanto questi framework interagiscono nativamente con la componentistica del dispositivo (in particolare con i sensori), e *risorse economiche*. Lo sviluppo di un'applicazione con un framework cross-platform riduce i costi in quanto non è necessario assumere sviluppatori che la realizzino per ogni singola piattaforma e permette all'azienda di affrontare meglio il *time-to-market*. Inoltre vi è un solo codice sorgente che viene gestito da un unico team di sviluppo.

Aziende molto importanti a livello mondiale si sono decise di utilizzare un framework cross-platofrom. In particolare, le aziende che verranno citate hanno sviluppato la loro applicazione utilizzando Flutter [44]:

1. **Alibaba:** uno dei più grandi e-commerce al mondo con milioni di utenti, ha realizzato la sua applicazione mobile per poter acquistare i prodotti offerti;
2. **Tencent:** questa azienda ha utilizzato Flutter su molteplici applicazioni. L'azienda fornisce principalmente servizi di intrattenimento (gaming) e di comunicazione (telefonia e messaggistica);
3. **New York Times:** ha realizzato un gioco disponibile per Android, iOS, Windows e macOS.

L'aumento dello sviluppo di applicazioni cross-platform è dato principalmente dall'evoluzione tecnologica verificatesi negli ultimi anni: prima i software multipiattaforma non garantivano delle prestazioni e dei risultati paragonabili alle corrispondenti applicazioni native.

7.3 Considerazioni su Dart

Dart è un linguaggio ben strutturato e che si è rivelato flessibile ed efficiente. Personalmente, sono convinto che il successo del linguaggio sia dato proprio dall'esperienza accumulata negli anni da Google, la quale affonda le proprie radici nello sviluppo Web. È un linguaggio che può sostituire JavaScript, sia lato client e sia lato server. A causa della scarsa diffusione, nei primi anni Dart rimase nel buio. Grazie all'avvento di Flutter, gli sviluppatori stanno scoprendo le potenzialità di questo linguaggio, aumentate durante gli anni.

Ho trovato Dart un linguaggio molto flessibile grazie ai molteplici paradigmi che supporta. È molto apprezzabile la vicinanza della sua sintassi a linguaggi come Java e JavaScript, facilitando l'apprendimento del linguaggio. Il linguaggio contiene delle caratteristiche particolari che mi hanno permesso di implementare delle soluzioni eleganti a fronte di determinate problematiche. Ho particolarmente apprezzato il doppio approccio alla compilazione *AOT* e *JIT*, che permette l'individuazione di errori già in fase di compilazione, riducendo notevolmente il tempo dedicato al *debugging*, senza però privare il linguaggio delle astrazioni che si basano sulla tipizzazione a tempo di esecuzione del codice.

Essendo un Android developer, ho potuto notare le differenze rispetto a *Kotlin*. Dart risulta essere un linguaggio molto più pulito rispetto a Kotlin anche se entrambi hanno di fatto sintassi molto simile e meccanismi comuni, come la tipizzazione *implicita* ed *esplicita*.

Sulla base della mia esperienza sostengo che Google abbia scelto Dart come linguaggio per la realizzazione di applicazioni in Flutter perchè l'azienda aveva

già sviluppato delle librerie Web per tale linguaggio, come ad esempio la libreria *Material*. Di conseguenza, risultava più facile realizzare applicazioni in Flutter mediante Dart. Se avessero deciso di utilizzare Kotlin come linguaggio per la scrittura di applicazioni in Flutter, il team di sviluppo avrebbe dovuto svolgere un lavoro ancora più oneroso.

Inoltre, sostengo che il punto a favore che aveva Dart rispetto a Kotlin era più di natura competitiva: Kotlin è un linguaggio nato nel 2011 e sviluppato da un’azienda della Repubblica Ceca, *JetBrains*. Dart invece è stato realizzato direttamente da Google. Di conseguenza, sostengo che la scelta di Dart fosse stata presa anche per non far dipendere il destino del framework da un linguaggio sviluppato da un’azienda esterna a Google.

7.4 Considerazioni su Flutter

Nonostante la giovane età, Flutter sta diventando un punto di riferimento per lo sviluppo di applicazioni cross-platform. Sempre più aziende si avvicinano a questo framework come strumento di sviluppo. LinkedIn, secondo i dati di cui dispone, mostra come Flutter comincia ad essere una *skill* molto richiesta agli sviluppatori in questo periodo e che la richiesta delle aziende a riguardo sta crescendo maggiormente rispetto a tutte le altre tecnologie mobile [45]. Un elemento da tenere in considerazione è l’enorme proliferazione di pacchetti e plugin su Pub. Questo significa che c’è un grande interesse attorno a questo framework e mostra anche come Flutter sia costantemente in evoluzione.

Dart si è rivelato essere un ottimo linguaggio per lo sviluppo di applicazioni cross-platform. Un linguaggio molto flessibile e completo in grado di risolvere le problematiche più spinose dello sviluppo mobile. Ritengo che in futuro, il mondo dello sviluppo di applicazioni mobili si polarizzerà principalmente in sostenitori di React Native e di Flutter. React Native e Flutter hanno due caratteristiche che li accomunano e che gli altri framework non possiedono: sono stati sviluppati da due aziende tecnologiche di livello mondiale (rispettivamente, Facebook e Google) e, come conseguenza di questo primo aspetto, hanno molti sviluppatori che sostengono e supportano il framework. Da quest’ultimo punto di vista Flutter è leggermente in svantaggio per il semplice fatto di essere approdato in questo mercato più tardi: sono convinto che nel giro di qualche anno riuscirà superare la metà degli sviluppatori che supporteranno React Native, in virtù anche del futuro sviluppo di Fuchsia.

Inoltre ho particolarmente apprezzato la metodologia utilizzata per realizzare le interfacce grafiche: il meccanismo della composizione dei Widget l’ho trovata affascinante dal punto di vista sia della scrittura ma anche della lettura

del codice. Progettare la UI secondo questo approccio permette di produrre del codice molto pulito.

Considero molto positivo l'obiettivo a lungo termine avviato da Google, ovvero, quello di utilizzare Flutter come framework principale per lo sviluppo di applicazioni per *qualsiasi* piattaforma, che questa sia Web, desktop, mobile o il futuro sistema operativo Fuchsia. È una visione molto ampia e di lungo termine che porterà i suoi benefici nei prossimi anni.

7.5 Considerazioni personali

7.5.1 Scelta del framework

Avendo un background come sviluppatore per applicazioni Android, ho particolarmente apprezzato questo framework soprattutto per queste caratteristiche:

1. **Cross-platform:** ho sempre desiderato implementare applicazioni anche per i dispositivi iOS, ma non conoscendo Swift e nemmeno il sistema operativo, ho sempre declinato questa possibilità. Quando scoprii che Flutter era un framework che mi permetteva di realizzare applicazioni per entrambi i sistemi operativi ho subito avuto un forte interessamento ed ho cominciato nel giro di pochi giorni a sviluppare la prima applicazione, grazie anche alla conoscenza accumulata con Java;
2. **Prestazioni:** essendo un framework multipiattaforma ho voluto accertarmi che l'applicazione prodotta rispettasse determinati parametri di efficienza e di performance. A seguito di alcuni test svolti sviluppando applicazioni che implementavano funzionalità diverse, ho avuto la possibilità di verificare tali requisiti;
3. **Forte tipizzazione:** sviluppando applicazioni per Android, volevo continuare a sviluppare app utilizzando comunque un linguaggio fortemente tipizzato come Java, per poter continuare a trarre tutti i vantaggi dalla forte tipizzazione.

Prima dell'avvento di Flutter ero già a conoscenza della presenza di React Native. Tuttavia, informandomi attraverso le esperienze di alcuni sviluppatori, non mi convinse come framework: alto consumo di memoria centrale, alto consumo di CPU, le prestazioni dell'applicazione non erano ottimali e non si avvicinavano a quelle di un'applicazione nativa. Inoltre, scrivere un'applicazione in React Native produce del codice molto verboso, a differenza di Flutter che produce del codice compatto. Altro aspetto che non mi ha convinto è la

presenza del *bridge* JavaScript per interagire con i moduli nativi del dispositivo, il quale rappresenta uno dei principali fattori che contribuisce al degrado complessivo delle prestazioni dell'applicazione.

La facilità nel realizzare delle interfacce grafiche è una ragione che mi ha convinto ancora di più ad adottare questo framework per lo sviluppo di app. Infatti è possibile realizzare UI complesse ma che siano comunque *user-friendly*.

7.5.2 Progettazione e sviluppo dell'applicazione

L'esperienza di tirocinio svolta per lo sviluppo dell'applicazione è stata molto coinvolgente e ricca di spunti per la realizzazione di nuovi progetti in futuro. Mi ha permesso di ampliare le conoscenze riguardo il mondo cross-platform ed il mondo Flutter, permettendomi anche di far tesoro dell'esperienza accumulata durante lo sviluppo di questo progetto. Sono stato particolarmente affascinato dai pattern architettonici disponibili e sostengo che la mia scelta di implementare il pattern *BLoC* sia stata una decisione corretta. È stata una scelta ben ponderata, in quanto volevo costruire una struttura sufficientemente stabile, ma allo stesso tempo, flessibile per poter supportare le funzionalità future che verranno implementate nell'applicazione.

Sono molto soddisfatto di aver fatto esperienza anche nella realizzazione dell'interfaccia grafica, un aspetto un po' carente nel mio bagaglio.

La soddisfazione più grande rimane il fatto di aver realizzato un'applicazione che può essere distribuita sia per Android che per iOS, a partire da un unico codice sorgente. È un risultato che mi rende molto felice e che mi ha particolarmente sorpreso ed entusiasmato.

Complessivamente, il processo di sviluppo mi ha invogliato molto a superare determinati limiti personali e ad andare oltre a quella *zona di comfort*, per esplorare e apprendere nuove conoscenze. Ritengo che esperienze come questa, debbano essere svolte il più possibile, in quanto permettono alla persona di crescere in modo esponenziale sotto diversi aspetti contemporaneamente: a partire dall'organizzazione del progetto, all'organizzazione dei tempi, delle risorse, all'analisi dei requisiti. Aspetti che vanno ad unire quelle che vengono definite *hard skills* e *soft skills*, entrambe di fondamentale importanza nel mondo dell'innovazione.

Appendice A

Classi ed interfacce in Dart

Una peculiarità di questo linguaggio è la definizione delle *interfacce* [46]. Quando si va a definire una normale classe, questa può essere estesa o implementata da altre classi. Prendendo l'esempio fornito dalla documentazione ufficiale [46]:

```
1 class Person {  
2     final _name;  
3  
4     // Not in the interface, since this is a constructor.  
5     Person(this._name);  
6  
7     // In the interface.  
8     String greet(String who) => 'Hello, $who. I am $_name.';  
9 }  
10  
11 class Impostor implements Person {  
12     get _name => '';  
13  
14     String greet(String who) => 'Hi $who. Do you know who I am?';  
15 }
```

Come è possibile vedere, la classe `Impostor` implementa la classe `Person` e pertanto dovrà implementare tutti i metodi pubblici. Un discorso analogo può essere fatto anche tra *classi astratte* ed interfacce. Si prenda esempio da un caso presente nell'applicazione realizzata:

```
1 abstract class BaseModel {  
2     Map<String, dynamic> toMap();  
3 }
```

```
1 class Location extends BaseModel {  
2     final String latitude;  
3     final String longitude;  
4  
5     Location({this.latitude = "0", this.longitude = "0"});  
6  
7     @override  
8     Map<String, dynamic> toMap() {  
9         return null;  
10    }  
11 }
```

Ma è possibile anche implementare la classe `BaseModel`:

```
1 class Location implements BaseModel {  
2     final String latitude;  
3     final String longitude;  
4  
5     Location({this.latitude = "0", this.longitude = "0"});  
6  
7     @override  
8     Map<String, dynamic> toMap() {  
9         return null;  
10    }  
11 }
```

Dart definisce questo concetto come una *definizione implicita* delle interfacce. Ogni classe definisce implicitamente un'interfaccia contenente tutti i membri dell'istanza della classe e di tutte le interfacce che implementa.

Appendice B

AutomaticKeepAliveClientMixin

Nei linguaggi di programmazione orientati agli oggetti, una **mixin** è una classe che contiene dei metodi che possono essere utilizzati da parte di altre classi, senza la necessità di dover *estendere* la classe. Il modo in cui le classi ottengono l'accesso ai metodi della mixin dipende dal linguaggio. Una mixin può essere vista come un'interfaccia i cui metodi sono già implementati.

Queste particolari classi vengono ampiamente utilizzate nei linguaggi moderni, sia nei linguaggi che implementano l'*ereditarietà singola*, sia nei linguaggi che supportano l'*ereditarietà multipla*. La problematica principale dell'ereditarietà singola è la mancanza di poter estendere più classi contemporaneamente: questo problema può essere parzialmente risolto con l'utilizzo delle *interfacce*, ma non sempre permettono di ottenere i risultati desiderati. Invece, nonostante l'ereditarietà multipla permetta una maggiore flessibilità nel *riuso del codice*, essa può essere fonte di problemi. In particolare, può verificarsi quello che viene chiamato **name clash**: è un problema che si verifica quando uno stesso metodo è definito in più *super-classi*. In una situazione del genere, la *sotto-classe* non è in grado di decidere quale metodo ereditare. Sulla base di queste casistiche, l'utilizzo delle mixin si rivela molto utile in quanto permette di aumentare le potenzialità dei linguaggi con ereditarietà singola e di risolvere i problemi causati dall'ereditarietà multipla.

La mixin `AutomaticKeepAliveClientMixin` [47] [48] è una particolare classe che è stata di fondamentale importanza per lo sviluppo dell'applicazione. Questa classe consente ai sotto-alberi di chiedere al framework di essere mantenuti in "vita" (*alive*). Tutta la struttura viene mantenuta in vita ogni volta che uno o più discendenti hanno inviato una `KeepAliveNotification`. Per mantenere in "vita", si intende che il framework non deve deallocare o distruggere un determinato oggetto, ma lo deve mantenere attivo, anche se in un dato istante non è strettamente necessario per il proseguimento dell'esecuzione dell'applicazione. Di seguito di introduce un esempio per comprendere meglio

l'importante compito di questa classe. Questa mixin è stata di fondamentale importanza per tutte le classi che vanno a comporre l'interfaccia grafica. Infatti, senza l'utilizzo di questa classe, ad ogni cambio di schermata, le pagine che sono state caricate precedentemente, venivano deallocate. Di conseguenza, quando si ritorna su una pagina che era già stata visualizzata precedentemente, questa doveva essere totalmente ricaricata e renderizzata. Tutto questo contribuisce a degradare le prestazioni dell'applicazione e fornisce all'utente una pessima *user experience*. Grazie a questa mixin, invece, la navigazione tra le varie schermate non comporta ad alcun *reload*. Tutte le schermate che sono già state inizializzate, perché l'utente le ha già visionate, vengono mantenute attive in *background*: nel momento in cui l'utente ritorna su una di queste schermate, il framework non fa altro che riprendere l'oggetto già istanziato e proporlo al livello di presentazione. Se l'applicazione non avesse integrato questa mixin, oltre ad avere dei problemi nell'esperienza d'uso, il problema più grave sarebbero stato quello di non fornire un servizio affidabile all'utente, ovvero, il monitoraggio dei dati. Infatti, senza questa mixin, ogni volta che l'utente ritornava su una schermata che aveva già visionato, la schermata doveva essere ricaricata. In quel frangente di tempo necessario per ricaricare e renderizzare nuovamente tutta la schermata, l'applicazione *non visualizza alcun dato*. Quindi, l'applicazione non avrebbe fornito correttamente il servizio per cui era stata progettata. Per questa ragione, durante tutta la spiegazione relativa a questa classe, essa è stata più volte definita come *fondamentale* per la corretta implementazione dell'applicazione.

A livello di codice, `AutomaticKeepAliveClientMixin` viene integrata nella classe come indicato a riga 2, ovvero, con la parola riservata `with`. Questa mixin contiene soltanto un metodo, `wantKeepAlive` (riga 4), necessario per informare il framework se mantenere attiva o meno la schermata. Inoltre, nel corpo del metodo `build` della classe `_NavigationPageState` deve essere aggiunto `super.build(context)` [47] (riga 8).

```

1 class _NavigationPageState extends State<NavigationPage>
2     with AutomaticKeepAliveClientMixin<NavigationPage> {
3
4     @override
5     bool get wantKeepAlive => true;
6
7     @override
8     Widget build(BuildContext context) {
9         super.build(context);
10        return GridBox(
11            bloc: widget.navigationBloc,
12            themeHandler: widget.themeHandler,
```

```
12     initialData: Navigation(),
13   );
14 }
15 }
```


Bibliografia

- [1] Regata costiera. [https://it.wikipedia.org/wiki/Vela_\(sport\)
#Regate_costiere](https://it.wikipedia.org/wiki/Vela_(sport)#Regate_costiere), Maggio 2020
- [2] Vela, sport. [https://it.wikipedia.org/wiki/Vela_\(sport\)](https://it.wikipedia.org/wiki/Vela_(sport)), Maggio 2020.
- [3] Barca a vela da regata William B. <https://qui.uniud.it/notizieEventi/ricerca-e-innovazione/>, articolo: *Uniud Sailing Lab: ottimi risultati nelle regate autunnali di Lignano Sabbiadoro*. Maggio 2020.
- [4] GPS. <https://www.gps.gov/systems/gps/>, Maggio 2020.
- [5] Statistiche sull'utilizzo dei sistemi operativi mobili, [https://www.statista.com/statistics/272698/
global-market-share-held-by-mobile-operating-systems-since-2009/](https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/), Maggio 2020.
- [6] Trend dei framework mobile su Google Trends. <https://trends.google.it/trends/explore?date=today%205-y&q=Flutter,React%20Native,Xamarin,Ionic,NativeScript>, Maggio 2020.
- [7] Trend dei framework mobile su StackOverflow. <https://insights.stackoverflow.com/trends?tags=flutter%2Creact-native%2Cxamarin%2Cionic%2Ccordova%2Cnativescript>, Maggio 2020.
- [8] Grafico illustrativo per le app Native, Hybrid e Web, <https://clevertap.com/blog/types-of-mobile-apps/>, Maggio 2020.
- [9] Considerazioni sull'approccio di sviluppo da utilizzare, <https://ralsware.com/blog/native-vs-hybrid-vs-cross-platform/>, Maggio 2020.

- [10] Tasso di abbandono di un'applicazione, <https://www.statista.com/statistics/751532/worldwide-application-user-retention-rate/>, Maggio 2020.
- [11] Email di un dipendente di Google relativo a *Dash* (Dart). <https://gist.github.com/paulmillr/1208618>, Maggio 2020.
- [12] Pub, gestore dei pacchetti in Dart. <https://pub.dev/>, Maggio 2020.
- [13] Homepage di Dart. <https://dart.dev/>, Maggio 2020.
- [14] Descrizione del linguaggio Dart. <https://dart.dev/guides/language/language-tour>, Maggio 2020.
- [15] Elenco dei paradigmi di Dart. <https://flutter.dev/docs/resources/faq#what-programming-paradigm-does-flutters-framework-use>, Maggio 2020.
- [16] Reactive Manifesto. <https://www.reactivemanifesto.org/>, Maggio 2020.
- [17] Programmazione reattiva. https://en.wikipedia.org/wiki/Reactive_programming, Maggio 2020.
- [18] Compilazione anticipata (AOT). https://en.wikipedia.org/wiki/Ahead-of-time_compilation, Maggio 2020.
- [19] Rappresentazione intermedia. https://en.wikipedia.org/wiki/Intermediate_representation#Intermediate_language, Maggio 2020.
- [20] Tipizzazione di Dart. <https://dart.dev/faq#q-is-dart-a-statically-typed-language>, Maggio 2020.
- [21] Compilazione Just-In-Time (JIT). https://en.wikipedia.org/wiki/Just-in-time_compilation, Maggio 2020.
- [22] Piattaforme su cui può essere eseguito il codice Dart. <https://dart.dev/platforms>, Maggio 2020.
- [23] Dartdevc. <https://dart.dev/tools/dartdevc/faq>, Maggio 2020.
- [24] *Isolate* e *Event Loop*. <https://medium.com/dartlang/dart-asynchronous-programming-isolates-and-event-loops-bfffc3e296a6a>, Maggio 2020.

- [25] *Isolate e Event Loop.* https://www.youtube.com/watch?v=vl_AaCgudcY, Maggio 2020.
- [26] Event loop. <https://webdev-angular3-dartlang-org.firebaseioapp.com/articles/performance/event-loop>, Maggio 2020.
- [27] Future. https://www.youtube.com/watch?v=0TS-ap9_aXc, Maggio 2020.
- [28] Sito ufficiale di Flutter. <https://flutter.dev/>, Maggio 2020.
- [29] Panoramica tecnica di Flutter. <https://flutter.dev/docs/resources/technical-overview>, Maggio 2020.
- [30] Skia. <https://skia.org/>, Maggio 2020.
- [31] Hot Reload. <https://flutter.dev/docs/development/tools/hot-reload>, Maggio 2020.
- [32] Sito ufficiale di Fuchsia. <https://fuchsia.dev/>, Maggio 2020.
- [33] Google Fuchsia. https://en.wikipedia.org/wiki/Google_Fuchsia, Maggio 2020.
- [34] Test di Fuchsia su Google Nest Hub. <https://www.tomshw.it/>, articolo: *Google Fuchsia si avvicina: l'OS verso l'ultima fase di test interni.* Maggio 2020.
- [35] Pacchetto url_launcher. https://pub.dev/packages/url_launcher#-installing-tab-, Maggio 2020.
- [36] Redux. <https://blog.novoda.com/introduction-to-redux-in-flutter/>, Maggio 2020.
- [37] Provider. <https://medium.com/flutter-community/understanding-provider-in-diagrams-part-1-providing-values-4379aa1e7fd5>, Maggio 2020.
- [38] Prima presentazione del pattern BLoC. <https://www.youtube.com/watch?v=PLHln7wHgPE>, Maggio 2020.
- [39] Repository pattern. <http://hannesdorfmann.com/android/evolution-of-the-repository-pattern>, Maggio 2020.

- [40] Architettura implementata. <https://www.toptal.com/cross-platform/code-sharing-angular-dart-flutter-bloc>, Maggio 2020.
- [41] Pacchetto connectivity 0.4.8+2. <https://pub.dev/packages/connectivity/versions/0.4.8+2>, Giugno 2020.
- [42] Future Microtask. <https://www.woolha.com/tutorials/dart-how-to-schedule-a-microtask>, Giugno 2020.
- [43] Schema Event Queue e Microtask Queue. <https://www.woolha.com/articles/dart-event-loop-microtask-event-queue>, Giugno 2020.
- [44] Aziende che utilizzano Flutter. <https://flutter.dev/showcase>, Maggio 2020.
- [45] L'abilità di sviluppare applicazioni in Flutter è la skill più richiesta nell'ultimo periodo. <https://learning.linkedin.com/blog/tech-tips/the-fastest-growing-skills-among-software-engineers--and-how-to->, Maggio 2020.
- [46] Interfacce implicite in Dart. <https://dart.dev/guides/language/language-tour#implicit-interfaces>, Giugno 2020.
- [47] AutomaticKeepAliveClientMixin. <https://api.flutter.dev/flutter/widgets/AutomaticKeepAliveClientMixin-mixin.html>, Giugno 2020.
- [48] AutomaticKeepAlive. <https://api.flutter.dev/flutter/widgets/AutomaticKeepAlive-class.html>, Giugno 2020.