UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica e

dell'Automazione

Progettazione e sviluppo di funzionalità di Citizen science per l'applicazione mobile "Ammoniti di strada"

Design and development of Citizen science functionalities for the mobile app "Ammoniti di strada"

Relatore:

Prof. Emanuele Storti

Tesi di laurea di:

Daniele Sergiacomi

ANNO ACCADEMICO

2021/2022

SOMMARIO

Introduzione 8

Capitolo 1 - Analisi dei requisiti 11

1.1 | Requisiti 11

1.1.1| **Punto di partenza** 11

1.1.2 | **Requisiti funzionali** 12

1.1.3| **Requisiti non funzionali** 13

1.1.4| **Mobile** 14

1.1.5| **Android** 14

1.2| Applicazioni Native, Ibride e Web App 15

1.2.1| **Web App** 16

1.2.2| **App Native** 16

1.2.3| **App Ibride** 17

Capitolo 2 – Casi d'uso e Collegamenti Logici 18

2.1| Attori e Casi d'uso 18

2.1.1| **Diagrammi dei Casi d'uso per l'utente non autenticato** 19

2.1.2| **Diagrammi dei Casi d'uso per l'utente autenticato** 19

2.2| Mockup 20

2.3| Collegamenti Logici 20

Capitolo 3 – Strumenti Utilizzati 21

3.1| Android Studio 21

3.2| Dart 23

3.3| Flutter 25

3.4| Firebase 27

3.4.1| **Storage** 28

3.4.2| **Autenticazione** 29

3.5| GitHub 29

3.5.1| **Controllo delle versioni** 29

3.5.2| **Git** 29

3.6| Package e Plugin 30

Capitolo 4 – Progettazione 33

4.1| **Architettura dell'applicazione** 33

4.2| **Progettazione dei dati** 37

4.3| Logica dell'applicazione 42

4.3.1| **la funzione main** 42

4.3.2| **Lo Widget OnboardingScreen** 42
4.3.3| **Lo Widget MyHomePage** 42
4.3.4| **I metodi getZones, getFossils e createMarkers** 44
4.3.5| **Processo di invio delle notifiche** 45
4.3.6| **Lo Widget ZoneScreen** 45
4.3.7| **Lo Widget FossilScreen** 46
4.3.8| **Login, Registrazione, Logout** 46
4.3.9| **Lo Widget ImageUpload** 46
4.3.10| **Lo Widget ShowUploads** 47
4.3.11| **Navigazione fra le schermate** 47
4.4| Dettagli implementativi 48
4.4.1| **WidgetsFlutterBinding, MyApp e initScreen** 48
4.4.2| **il metodo build di OnboardingPage** 49
4.4.3| **il metodo build di MyHomePage** 50
4.4.4| **Tecniche utilizzate in getZones, getFossils e createMarkers (MyHomePage)** 52
4.4.5| **Costruzione della lista delle zone nelle vicinanze** 56
4.4.6| **Il metodo di ricerca per le zone e per i fossili (MyHomePage)** 58
4.4.7| **Gestione del Back Button con WillPopScope** 61
4.4.8| **Il metodo build di ZoneScreen** 63
4.4.9| **Tecniche utilizzate in getFossils (ZoneScreen)** 63
4.4.10| **Costruzione della lista dei fossili associati ad una zona** 64
4.4.11| **IL METODO BUILD DI FOSSILSCREEN** 66
4.4.12| **Tecniche utilizzate in signIn, signUp, postDetailsToFirestore, logout** 67
4.4.13| **il metodo build di Profile** 69
4.4.14| **il metodo build di ImageUpload** 70
4.4.15| **Tecniche utilizzate in imagePickerMethod e uploadImage** 72
4.4.16| **il metodo build di ShowUploads** 73
4.5| Screenshot delle schermate 75
Conclusione: 80
Bibliografia: 82

Figura 0.1: Ammonite 8
Figura 0.2: Belemnite 8
Figura 1.1: Collegamenti logici dell'applicazione nella sua prima versione 11
Figura 1.2: Stack di Android 15
Figura 1.3: Classificazione tipologie di app 16
Figura 2.1: Diagramma dei casi d'uso per l'utente non autenticato 19
Figura 2.2: Diagramma dei casi d'uso per l'utente autenticato 19
Figura 2.3: Collegamenti logici delle schermate 20
Figura 3.1: Interfaccia IDE Android Studio 23
Figura 3.2: Architettura Flutter 25
Figura 3.3: Modello di dati NoSQL adottato da Firestore 27
Figura 3.4: Interfaccia della Firebase Console 28
Figura 3.5: Dipendenze dichiarate nel file 'pubspec.yaml' 31
Figura 4.1: Architettura applicazione Ammoniti di strada 34
Figura 4.2: Organizzazione classi dart 34
Figura 4.3: Organizzazione delle collection nel Cloud Firestore 38
Figura 4.4: Documento delle zone nel database Cloud Firestore 38
Figura 4.5: Documento dei fossili nel database Cloud Firestore 39
Figura 4.6: Documento delle segnalazioni nel database Cloud Firestore 40
Figura 4.7: Documento degli utenti nel database Cloud Firestore 40
Figura 4.8: Cartelle dello Storage 41
Figura 4.9: Contenuto della cartella delle foto dei fossili 41
Figura 4.10: Contenuto della cartella delle foto delle segnalazioni 41
Figura 4.11: Dichiarazioni delle immagini della cartella assets 42
Figura 4.12: Il metodo main 48
Figura 4.13: Lo Widget MyApp, il suo metodo build e la rotta iniziale 49
Figura 4.14: Prima parte dello Widget build di Onboarding 49
Figura 4.15: Seconda parte dello Widget build di Onboarding 49
Figura 4.16: Terza parte dello Widget build di Onboarding 50
Figura 4.17: Widget WillPopScope ritornato dal metodo build 50
Figura 4.18: Codice che definisce la struttura dello Scaffold 52
Figura 4.19: Prima parte del metodo getZones (recupero della posizione) 53
Figura 4.20: Seconda parte del codice di getZones (recupero dei dati) 54
Figura 4.21: Codice di getFossils (recupero dei dati) 55
Figura 4.22: Parte del codice di createMarkers (recupero dei dati) 55
Figura 4.23: Parte del codice di createMarkers (gestione della InfoWindow) 56
Figura 4.24: Codice con controllo della disponibilità dei dati e verifica errori 56

Figura 4.25: Codice per mostrare messaggi di errore sulla geolocalizzazione 57
Figura 4.26: Codice con filtraggio zone in base alla posizione e costruzione lista 58
Figura 4.27: Parte di codice della funzione di ricerca nell'appBarSetter 58
Figura 4.28: Prima parte di codice della funzione di ricerca per zone nel bodySetter 59
Figura 4.29: Seconda parte di codice della funzione di ricerca per zone nel bodySetter (verificazione dei dati) 59
Figura 4.30: Prima parte di codice della funzione di ricerca per fossili nel bodySetter 60
Figura 4.31: Seconda parte di codice della funzione di ricerca per fossili nel bodySetter (form di cave di provenienza) 60
Figura 4.32: Terza parte di codice della funzione di ricerca per fossili nel bodySetter (form di tipi di rocce) 60
Figura 4.33: Quarta parte di codice della funzione di ricerca per fossili nel bodySetter 61
Figura 4.34: Prima parte di codice di OnWillPopScope 62
Figura 4.35: Seconda parte di codice di OnWillPopScope 62
Figura 4.36: Build di ZoneScreen 63
Figura 4.37: Codice di getFossils 64
Figura 4.38: Prima parte di codice relativa alla costruzione della lista dei fossili associata ad una zona 65
Figura 4.39: Seconda parte di codice relativa alla costruzione della lista dei fossili associata ad una zona 65
Figura 4.40: Inizializzazione e definizione del _tabController 66
Figura 4.41: Metodo build di FossilScreen (relativo all'ImageSlider) 67
Figura 4.42: Metodo signIn, per loggarsi nell'app 67
Figura 4.43: Metodo signUp, per registrare un profilo utente nell'app 68
Figura 4.44: Metodo postDetailsToFirestore, per inviare al database i dati dell''utente che si sta registrando 68
Figura 4.45: Metodo signOut, per effettuare il logout 68
Figura 4.46: Prima parte del codice del profilo utente (estrazione dati utente loggato) 69
Figura 4.47: Seconda parte del codice del profilo utente (implementazione grafica) 70
Figura 4.48: Prima parte del codice di ImageUpload (inizializzazione variabili) 70
Figura 4.49: Seconda parte del codice di ImageUpload (appbar) 71
Figura 450: Terza parte del codice di ImageUpload (parte del body) 71
Figura 4.51: Metodo ImagePickerMethod 72
Figura 4.52: Prima parte di codice del metodo uploadImage (campionamento dati relativi alla segnalazione) 73
Figura 4.53: Seconda parte di codice del metodo uploadImage (caricamento segnalazione nel cloud Firestore) 73
Figura 4.54: Codice di ShowUploads 74
Figura 4.55: Indici del Cloud Firestore in cui si dichiara l'ordine decrescente nell'ordinamento delle segnalazioni 74
Figura 4.56: Onboarding (prima schermata)
Figura 4.57: Onboarding (seconda schermata) 75
Figura 4.58: Onboarding (terza schermata)
Figura 4.59: Onboarding (quarta schermata) 76
Figura 4.60: Onboarding (ultima schermata)
Figura 4.61: Home 76

Figura 4.62: Zone nelle vicinanze
Figura 4.63: Tutti i reperti 77
Figura 4.64: Ricerca per zone
Figura 4.65: Ricerca per fossili 77
Figura 4.66: ZoneScreen
Figura 4.67: Dettaglio fossile 78
Figura 4.68: Login
Figura 4.69: Registrazione 78
Figura 4.70: Profilo
Figura 4.71: Invia segnalazioni 79
Figura 4.72: Mostra segnalazioni 79

INTRODUZIONE

Nella città di Ancona, in particolare nei monumenti storici e nelle pavimentazioni, le rocce sono sede di reperti antichi, chiamati anche fossili.

Il termine fossile (dal latino fodere, "scavare") in paleontologia viene abitualmente usato per indicare resti integri o parziali di organismi un tempo viventi; più in generale, viene usato per una qualsiasi testimonianza di vita geologicamente passata (antecedente all'epoca attuale): resti animali, quali ossa, denti, uova, conchiglie; resti vegetali, quali foglie, tronchi, pollini.

I fossili si trovano inglobati nelle rocce sedimentarie abbondantemente presenti nella parte superiore della crosta terrestre. Utili per la datazione delle rocce calcaree mesozoiche sono: stromatoliti, ammoniti, belemniti, bivalvi, gasteropodi, brachiopodi, echinodermi, crinoidi, denti isolati di squalo e microfossili [1].

Molte delle pietre ornamentali sono di origine marina; in particolare Ancona risulta ricca di ammoniti e belemniti.

Le ammoniti (Figura 0.1) sono un gruppo di Molluschi Cefalopodi dell'ordine Ammonoidea ormai estinti; più precisamente sono comparse nel periodo Devoniano Inferiore (circa 400 milioni di anni fa), e si sono estinte 65,5 milioni di anni fa, verso la fine del Mesozoico [2].

Figura 0.1: Ammonite

Mentre i belemniti (Figura 0.2) sono cefalopodi fossili comparsi nel Triassico (225-190 milioni di anni fa) ed estinti nel Cretacico (135-65 milioni di anni fa). Erano cefalopodi marini, lontani parenti di polpi, seppie e calamari, caratterizzati dalla presenza di una conchiglia interna, spesso indicata come rostro; questa conchiglia costituisce l'unica parte che solitamente si rinviene fossilizzata [3].

Figura 0.2: Belemnite

Grazie alla loro repentina evoluzione e all'ampia area di distribuzione, questi fossili risultano essere uno strumento fondamentale per geologi e paleontologi nei rilevamenti biostratigrafici. Nel problema di datazione delle rocce sedimentarie, le ammoniti e i belemniti consentono di collegare gli strati di roccia a precise età geologiche.

In questo contesto agisce il progetto 'Ammoniti di strada', finanziato dal Ministero della Ricerca e dell'Università, il quale ha come obiettivo la promozione del patrimonio geologico e il coinvolgimento delle persone residenti in questi luoghi.

Oltre alla conoscenza di tali reperti, che sicuramente sono caratterizzati da un notevole fascino dovuto alla loro storia, questo progetto mira a far comprendere ai cittadini l'importanza delle scienze geologiche. Queste, infatti, attraverso lo studio del passato, permettono di prevedere possibili estinzioni o altri scenari futuri.

L'applicazione 'Ammoniti di strada' si inserisce in questo progetto, in quanto potrà servire come strumento tecnologico per far avvicinare le persone ai reperti storici di Ancona. L'app, inoltre, è in collaborazione con il Dipartimento di Scienze della Vita e dell'Ambiente dell'Università Politecnica delle Marche, il quale si occupa della parte scientifica di recupero e studio dei dati sulle ammoniti.

l'obiettivo del lavoro descritto nel presente documento è stato quello di sviluppare una serie di nuove funzionalità per un'applicazione mobile volta alla valorizzazione del patrimonio paleontologico della città di Ancona, l'applicazione esisteva già ed è stata inizializzata dal precedente tirocinante Matteo Cirilli [4].

Il progetto comprendeva già alcune funzionalità di base, ovvero le seguenti: visualizzazione di una mappa in cui i reperti sono geolocalizzati, l'invio di notifiche all'utente all'avvicinarsi a un reperto, la visualizzazione dei reperti più vicini, la lista di tutti i fossili e la possibilità di visualizzarne foto e dettagli, le quali sono state in seguito modificate per adattarsi al cambiamento di funzionalità dell'applicazione; ad esempio in seguito ad un problema di raggruppamento e rappresentazione dei reperti si è deciso di porre la geolocalizzazione sulle varie zone sparse per la città di Ancona che a loro volta contengono i reperti, e non direttamente sui fossili, evitando così un eccessivo affollamento di Marker sulla mappa, inoltre in origine era prevista solo un'immagine per fossile, ma dal momento che capire la posizione di un reperto conoscendo solo la zona e vedendo solo un'immagine è risultato complicato si è deciso di mettere 2 foto per fossile: uno per il dettaglio del fossile, e l'altra più generale per far capire meglio all'utente la sua collocazione nella zona.

Le nuove funzionalità che sono state aggiunte invece sono: la possibilità dell'utente di registrarsi e autenticarsi per permettere delle ulteriori funzionalità specifiche, come ad esempio fare delle vere e proprie segnalazioni di altri fossili da mandare ad un team di ricercatori con lo scopo di espandere il database con nuovi reperti e visualizzare in seguito tali segnalazioni, oltre ad un sostanzioso cambio dell'aspetto grafico e l'aggiunta di nuove schermate, Floating Action Bar ed elementi della bottom Navigation Bar per rendere più organico e user-friendly l'intera applicazione.

Nel documento di tesi, le ammoniti saranno chiamate anche fossili, in modo più generale, oppure reperti.

Lo sviluppo dell'applicazione è stato realizzato passando per diverse fasi, descritte in modo accurato nei capitoli di questo documento di tesi.

- Capitolo 1: si parla innanzitutto in modo più specifico di cosa è stato già precedentemente implementato, dei problemi da risolvere e delle funzionalità aggiuntive da implementare, successivamente si discute la raccolta dei requisiti dell'app, distinguendo quelli funzionali da quelli non funzionali; i requisiti funzionali sono le caratteristiche che l'app deve avere, mentre quelli non funzionali rappresentano i vincoli ad essi collegati di cui tenere conto; infine si parlerà della programmazione mobile, di Android e di tutti i tipi di applicazione (native, ibride e Web app).

- Capitolo 2: è descritto il diagramma dei casi d'uso, in cui vengono riassunte le possibili interazioni tra l'utente e l'applicazione, nel caso in cui l'utente è autenticato oppure no. In questo capitolo non verranno visualizzati dei Mockup grafici delle schermate, in quanto la struttura del programma è stata già precedentemente definita, ma oltre a ciò, si è deciso di implementare i collegamenti logici, immaginando i possibili percorsi all'interno dell'app causati dall'interazione dell'utente.

- Capitolo 3: vengono elencati gli strumenti scelti per la scrittura effettiva del codice. In quanto l'applicazione in futuro dovrà essere cross-Platform, si è deciso di optare per il framework Flutter con il linguaggio di programmazione Dart, integrato nell'ambiente di sviluppo Android Studio. Poi si sono sfruttati i servizi della piattaforma Firebase per la gestione dei dati. Il tutto utilizzando GitHub per la condivisione, archiviazione e gestione del codice. Infine vengono elencati i vari plugin e package utilizzati per l'applicazione.

- Capitolo 4: riassume la fase di progettazione dell'applicazione. Questa fase è composta da diverse parti. Prima di tutto si è definita l'architettura, distinguendo le componenti modulari dell'app. In seguito, si è passati alla progettazione dei dati, ed infine si è proceduto con lo sviluppo dell'applicazione, parlando in modo più tecnico delle varie classi e metodi implementati. Nell'ultima sezione di questo capitolo sono mostrati gli screenshot dell'app.

CAPITOLO 1 - ANALISI DEI REQUISITI

In questa sezione verranno analizzati i requisiti necessari per la progettazione e l'implementazione dell'applicazione da realizzare. Si partirà quindi dal descrivere l'obiettivo dell'applicativo, per poi parlare dell'applicazione a partire dalla prima versione elencandone le funzionalità da sviluppare e infine parlare dei requisiti funzionali e non.

1.1 | REQUISITI

Un requisito è detto funzionale se descrive cosa deve e cosa non deve fare un sistema software e come deve reagire agli stimoli esterni, mentre i requisiti non funzionali sono le proprietà del sistema che devono essere soddisfatte, quindi pongono vincoli su come si comporterà il sistema. In questa sezione verranno analizzati i requisiti che caratterizzeranno l'applicazione.

1.1.1| PUNTO DI PARTENZA

Quando ho iniziato a sviluppare l'applicazione per il tirocinio erano presenti già alcune funzionalità tra cui:

- Possibilità di visualizzare tutti i fossili presenti nel database, sia tramite una lista sia tramite una mappa.
- Possibilità di visualizzare i fossili più vicini all'utente.
- Funzioni di filtraggio in base al nome, descrizione, cava di provenienza e tipo di roccia del fossile.
- Possibilità di consultare i dettagli di un determinato fossile.

Tuttavia, anche se l'applicazione risultava già alquanto efficiente ed efficace, era ancora in una fase embrionale e dovevano essere sistemati alcuni problemi, inoltre andavano aggiunte nuove funzionalità (che saranno elencate nel capitolo sottostante). Quindi, quando è stato ripreso il progetto la situazione di partenza era esattamente come viene mostrato nella *Figura 1.1*:

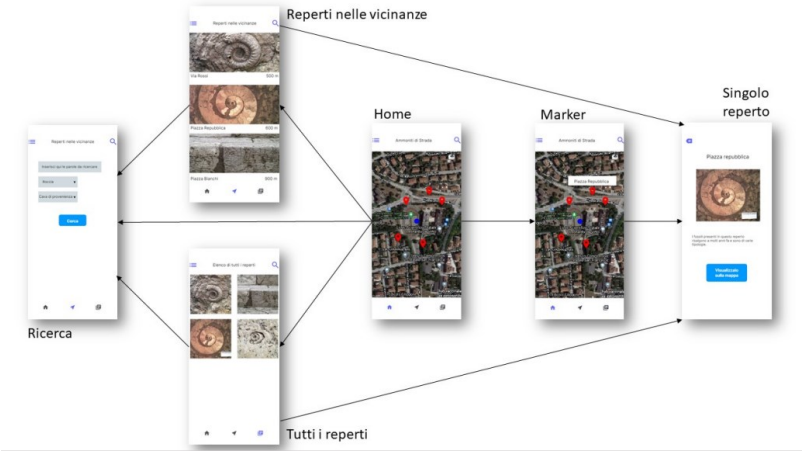

Figura 1.1: Collegamenti logici dell'applicazione nella sua prima versione

Il più grande problema era la saturazione della mappa di fossili e la poca chiarezza nel determinare la loro posizione precisa, in quanto in una città come Ancona è presente un notevole numero di reperti. Ciò comporta la difficoltà di riuscire a visualizzare i corrispondenti marker su mappa in modo preciso, rendendo difficoltosa la fruizione dei dati all'utente.

Inoltre, con solo delle informazioni sulla latitudine e la longitudine risulta abbastanza complicato risalire al singolo fossile; quindi si è dovuti ricorrere ad un raggruppamento per zone: ovvero catalogare tutti i reperti in una determinata area che potrebbe essere una scalinata, una via, il muro di un monumento ecc., in questo modo sulla mappa non verrebbero visualizzati più i singoli reperti ma solamente le zone che li contengono, e una volta che si ha selezionato una determinata area sarà possibile visualizzare di tutti i fossili contenuti in essa, con un'ulteriore descrizione che specifica la sua corretta locazione.

Un altro problema risiedeva nella schermata di dettaglio del singolo fossile: infatti nella schermata è presente solo un'immagine relativa a quel fossile, e anche questo rende poco chiaro dove si trova precisamente il reperto nella zona, così si è deciso di aggiungere anche una seconda foto per evidenziare più precisamente dove andare per trovare il reperto.

Una funzionalità aggiuntiva da implementare (e sulla quale si è basata gran parte del lavoro), è stata quella di permettere la registrazione di utenti che potranno contribuire al progetto di mappatura dei fossili fotografandoli, dando una breve descrizione sul loro posizionamento e inviandoli al Dipartimento di Scienze della Vita e dell'Ambiente dell'Università Politecnica delle Marche; infine questi fossili verranno studiati e messi ufficialmente nel database contenente tutti i reperti.

1.1.2 | **REQUISITI FUNZIONALI**

L'applicazione finale, quindi, dovrà garantire le seguenti funzionalità:

- *Visualizzazione* di una schermata di Onboarding, ovvero un tutorial che spiegherà in modo elementare il funzionamento dell'app all'utente, questa schermata apparirà al primo accesso nell'app dopo l'installazione (e solo per una volta).
- *Visualizzazione* di tutte le zone che contengono dei reperti:
    - Mediante una mappa che contiene i Marker di tutte le zone (Monumento ai caduti, chiesa di san Ciriaco ecc.).
    - Mediante una lista delle zone più vicine in ordine crescente.
- *Ricerca* delle zone mediante una funzione di filtraggio per nome e descrizione della zona.
- *Visualizzazione* di tutti i fossili mediante una lista di tutti i reperti.
- *Visualizzazione* di una lista dei fossili appartenenti ad una determinata zona di interesse che è stata selezionata dalla mappa o dalla schermata delle zone più vicine.
- *Ricerca* dei fossili mediante una zona di filtraggio per nome, descrizione, cava di provenienza e tipo di roccia.
- *Consultazione* dei dettagli di un singolo fossile.

- *Funzione di notifica* che segnala se sono presenti delle zone che contengono fossili nelle vicinanze (con un raggio di 3km), dando anche il numero di tali zone (per funzionare il GPS deve essere attivo).
- *Creazione* di un account.
- *Login.*
- *Logout.*
- *Profilo* personale dell'utente autenticato con possibilità di:
    - *Inviare* delle segnalazioni, ovvero delle foto di fossili con un'eventuale descrizione sulla locazione.
    - *Vedere* tutte le segnalazioni dell'utente precedentemente effettuate in ordine di tempo (dalla più recente alla più remota).

1.1.3| REQUISITI NON FUNZIONALI

I seguenti requisiti non funzionali dovranno essere presenti nell'applicazione:

- *Mobile*: Ammoniti di strada sarà un'applicazione destinata ai dispositivi mobile.
- *Dart*: il linguaggio che verrà utilizzato per programmare l'applicazione è Dart, utilizzando il framework Flutter.
- *Android*: l'applicativo è stato sviluppato principalmente per il sistema operativo Android, anche se essendo stato usato Flutter, ideale per lo sviluppo cross-Platform, che potenzialmente potrebbe essere utilizzato anche per dispositivi iOS.
- *Velocità*: l'app dovrà essere fluida e veloce nell'esecuzione (questo implica anche che le foto che andranno caricate su un database chiamato Firebase, del quale parleremo in seguito e le segnalazioni che effettueranno gli utenti dovranno essere poco pesanti, massimo 500kb).
- *Design*: esso dovrà essere intuitivo, accattivante e facile da utilizzare anche per gli utenti meno esperti.
- *Adattabilità*: l'applicazione dovrà essere in grado di adattarsi ad eventuali modifiche ai dati contenuti nel database.
- *Espandibilità*: l'applicazione dovrà essere facilmente espandibile nel caso in cui siano disponibili nuovi dati relativi ad annate future.
- *Account*: la registrazione di un utente va a buon fine soltanto se tutti i campi sono compilati nel modo corretto (la password deve avere minimo 6 caratteri e la conferma della password deve corrispondere alla password, e l'email deve rispettare il formato tipico, ossia contenere la @ e terminare in .com o .it, o qualcosa di analogo).
- *Database*: si necessita di un database remoto che contenga tutti i dati necessari al funzionamento corretto dell'applicazione, in questo caso si è deciso di usare Firebase.
- *Internet*: si necessita di una connessione ad Internet per il corretto funzionamento dell'applicazione.
- *GPS*: la funzione di geolocalizzazione (GPS) può essere attivata soltanto se l'utente acconsente a fornire i permessi dell'applicazione.

1.1.4| MOBILE

Le applicazioni mobile sono dei programmi software progettati e sviluppati per funzionare su appositi dispositivi mobili quali smartphone, tablet, smartwatch e simili.

Tali dispositivi sono caratterizzati da memorie poco capienti e bassa capacità di elaborazione e processamento, inoltre, il loro funzionamento è vincolato sui consumi energetici a causa della limitata energia fornita dalla batteria. Dal momento che tali dispositivi sono stati pensati per seguire la mobilità dell'utente, risultano essere di piccole dimensioni, maneggevoli e quindi di più facile utilizzo rispetto ai tradizionali PC.

Infatti, come suggerito nell'articolo [2], adesso la maggior parte del traffico web deriva da dispositivi mobile e tenderà a crescere ancora, nasce quindi la necessità di spostare una grande quantità di servizi sul web mobile. Per questo motivo, oggi, esistono applicazioni mobile in grado di soddisfare la maggior parte delle necessità degli utenti. In particolare, l'applicativo sarà sviluppato per dispositivi mobili perché pensato per l'utilizzo in movimento.

1.1.5| ANDROID

L'applicativo si rivolge agli utenti che possiedono dispositivi mobili con sistema operativo Android (per ora). I due sistemi operativi che attualmente dominano sul mercato sono Android ed iOS. Il primo è di proprietà di Google ed è il più diffuso nel mondo mobile rappresentando una fetta di mercato del 72.84% circa (Giugno 2021), mentre il secondo è sviluppato da Apple in esclusiva per i propri dispositivi mobile. Android fu lanciato per la prima volta nel 2008, è basato su kernel Linux e segue il modello di sviluppo open source, quindi offre all'utente una vasta libertà di utilizzo ed una grande flessibilità declinata soprattutto nei termini di progettazione e rilascio di software ad esso dedicati [5]. Tale sistema operativo si è evoluto rapidamente, fino all'ultima versione rilasciata: Android 12.

Per gli scopi prefissati, pur avendo utilizzato Flutter che permette con lo stesso codice di ottenere sia applicazioni per Android che per iOS, si è ritenuto più opportuno focalizzarsi principalmente sulla realizzazione di un'applicazione in Android, poiché permette di avere un'utenza maggiore dovuta al fatto che come sistema operativo mobile è il più diffuso al mondo ed `e supportato da più dispositivi diversi sia a livello di hardware che a livello di prezzo. Inoltre il fatto di essere più aperto e di avere uno store ufficiale meno controllato facilita lo sviluppo e la diffusione dell'applicazione. È, quindi, opportuno descrivere in breve l'architettura alla base di tale sistema operativo mobile.

Figura 1.2: Stack di Android

Come si può vedere dall'immagine precedente, vi sono i seguenti livelli:

- **Apps**: sia di sistema che installate dall'utente
- **Framework Android**: framework Android: insieme di librerie in Java o in Kotlin per realizzare l'interfaccia grafica, per gestire le notifiche, per gestire la navigazione tra schermate e altro
- **Librerie native**: insieme di librerie in C o C++ per accedere ai servizi e ai componenti nativi del sistema
- **HAL**: permette di interfacciarsi con componenti hardware come la fotocamera o il modulo bluetooth
- **Kernel Linux**: gestisce le funzionalità più importanti del sistema, come la gestione della memoria e la sicurezza.

Un aspetto finale, ma non meno importante, è la scelta della versione di Android. Ogni applicazione può infatti funzionare a partire da una certa versione minima di Android, specificata alla sua prima creazione. Nel caso seguente si `e scelta come versione minima di Android la 4.4W, che permette quindi all'applicazione di funzionare correttamente su praticamente tutti i dispositivi mobile Android e di non essere neanche obsoleta.

1.2| APPLICAZIONI NATIVE, IBRIDE E WEB APP

Con il termine mobile application si intende un'applicazione software dedicata a dispositivi mobile. Essa è progettata secondo specifici criteri legati alle caratteristiche hardware dei dispositivi (che spesso possono rappresentare un limite) ed alle modalità di fruizione. Quest'ultime, a causa dei contesti più disparati in cui diventa necessario interagire con una app, sono mobili, dinamiche e discontinue [6].

Volendo classificare le applicazioni, sarebbe possibile dividere quest'ultime in tre macro-blocchi:

- Web app
- App ibride
- App native

Figura 1.3: Classificazione tipologie di app

Di seguito si andranno ad analizzare nel dettaglio tutte le tipologie di app appena elencate.

1.2.1| WEB APP

Dovendo definire una delle caratteristiche più importanti delle applicazioni web si dovrebbe necessariamente far rifermento al concetto di distribuzione: le web app, infatti, sono accessibili via web per mezzo di una rete; in definitiva, sono la versione mobile di siti web.

Esse hanno diversi vantaggi come l'essere in grado di funzionare su qualsiasi piattaforma o il non richiedere alcuno spazio di memoria per l'installazione o per il download.

Al contempo, però, presentano anche degli svantaggi: non sono in grado di interagire con l'hardware del dispositivo (come ad esempio il giroscopio), non possono funzionare senza connessione, hanno tipicamente prestazioni inferiori rispetto alle App Native o Ibride di cui parlerò qui di seguito.

1.2.2| APP NATIVE

Le app native sono applicazioni sviluppate specificamente per un sistema operativo (es: Android, iOS), per mezzo di uno specifico linguaggio di programmazione.

Esse si interfacciano al O.S. mobile nel modo più completo possibile. Essendo sviluppate per una specifica piattaforma possono interfacciarsi del tutto con le API della stessa, andando, così, ad interagire totalmente con l'hardware e il software. Ne consegue che potranno sfruttare tutte le funzionalità accessibili del dispositivo (fotocamera, NFC, geolocalizzazione, giroscopio, ecc.), integrandole ed eventualmente potenziandole.

Tali applicazioni, inoltre, funzionano sempre, anche off-line in assenza di connettività e sono molto più veloci ed affidabili rispetto alle altre app, e quindi offrono una User Experience di più alto livello. Altra caratteristica che viene annoverata tra i vantaggi che sviluppare un'applicazione nativa porta con sé è sicuramente la possibilità di utilizzare degli elementi nativi della piattaforma in termini di user-interface. Un look and feel comune ad altre applicazioni che l'utente è già abituato ad utilizzare, renderà l'ambiente con cui quest'ultimo andrà ad interfacciarsi più familiare e intuitivo.

Di contro, però, svilupparle è tipicamente più costoso di altre tipologie di app. Infatti, nonostante il risultato finale sia più o meno lo stesso in tutte le piattaforme, l'applicazione dovrà essere scritta in

linguaggi differenti. La logica alla base dell'app sarà la stessa ma a cambiare saranno, oltre al linguaggio, l'architettura ed il processo di sviluppo. Stesso discorso vale per la manutenzione e l'aggiornamento: sviluppare un app diversa per ogni sistema operativo per cui si intende rendere disponibile l'applicativo, infatti, richiede di manutenere ed aggiornare singolarmente tutte le applicazioni.

1.2.3| **APP IBRIDE**

Le App ibride sono realizzate per adattarsi a dispositivi differenti. Spesso sono sviluppate come app native provviste di interfacce HTML pari alle web app. Esse rappresentano un buon compromesso in termini di funzionalità tra app native e web. Sono più rapide da manutenere delle app native, in quanto il processo di sviluppo è unico, anche se le performance sono inferiori e presentano una stabilità ridotta.

CAPITOLO 2 – CASI D'USO E COLLEGAMENTI LOGICI

Una volta definiti i requisiti dell'applicazione da migliorare o da aggiungere, in particolare quelli funzionali, è stato possibile realizzare i diagrammi dei casi d'uso, ovvero lo schema in cui vengono rappresentate tutte le possibilità che l'applicazione fornisce [7]. Dai casi d'uso poi si è potuto sviluppare un concept delle varie schermate e il collegamento logico tra di esse, causato all'interazione dell'utente con gli elementi grafici.

2.1| ATTORI E CASI D'USO

Innanzitutto, prima di far vedere il diagramma dei casi d'uso, definiamo cosa sono gli attori, ossia i ruoli assumibili dagli utenti o da qualsiasi entità che interagisce con l'applicazione, e i casi d'uso, ossia le possibili azioni che possono essere compiute dagli attori. Essendo presente un meccanismo di autenticazione e registrazione degli utenti, saranno presenti due tipi di attori:

- Utente non autenticato.
- Utente autenticato.

Nell'immagine successiva è raffigurato il diagramma dei casi d'uso relativo all'applicazione. Come in ogni diagramma di questo tipo:

- Gli attori sono rappresentati tramite degli omini stilizzati.
- I casi d'uso sono rappresentati tramite delle ellissi.
- Le relazioni di inclusione ed estensione sono rappresentate tramite delle frecce direzionali.

2.1.1| DIAGRAMMI DEI CASI D'USO PER L'UTENTE NON AUTENTICATO

Figura 2.1: Diagramma dei casi d'uso per l'utente non autenticato

2.1.2| DIAGRAMMI DEI CASI D'USO PER L'UTENTE AUTENTICATO

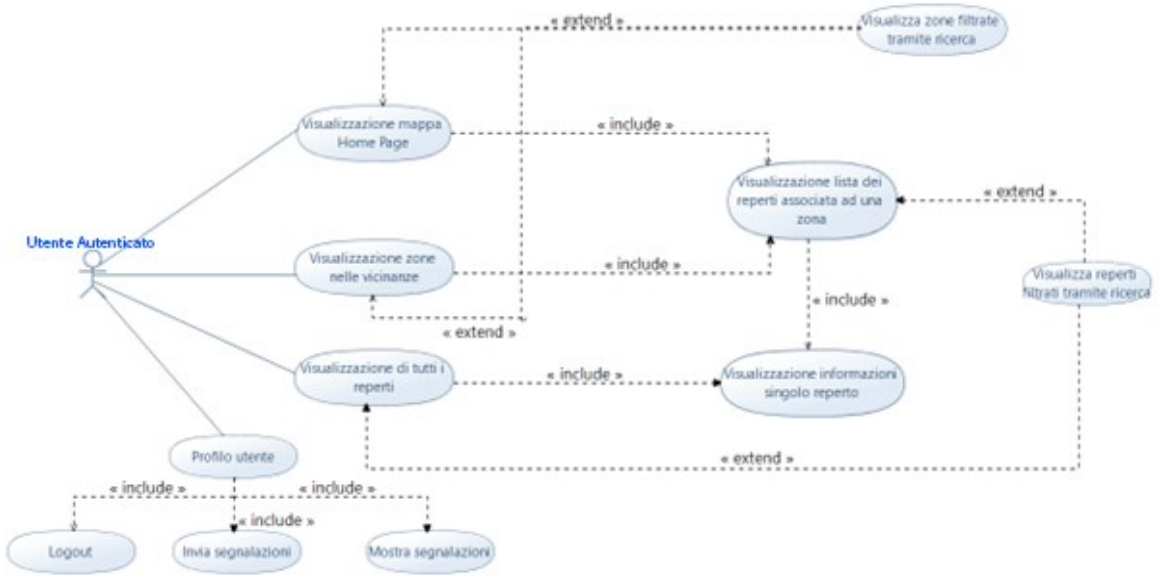

Figura 2.2: Diagramma dei casi d'uso per l'utente autenticato

2.2| MOCKUP

Siccome l'applicazione è stata precedentemente inizializzata, non è stato necessario effettuare il Mockup dell'applicazione, in quanto essendoci già una base solida si è passati direttamente alla parte dei collegamenti logici, che vengono mostrati nel capitolo successivo.

2.3| COLLEGAMENTI LOGICI

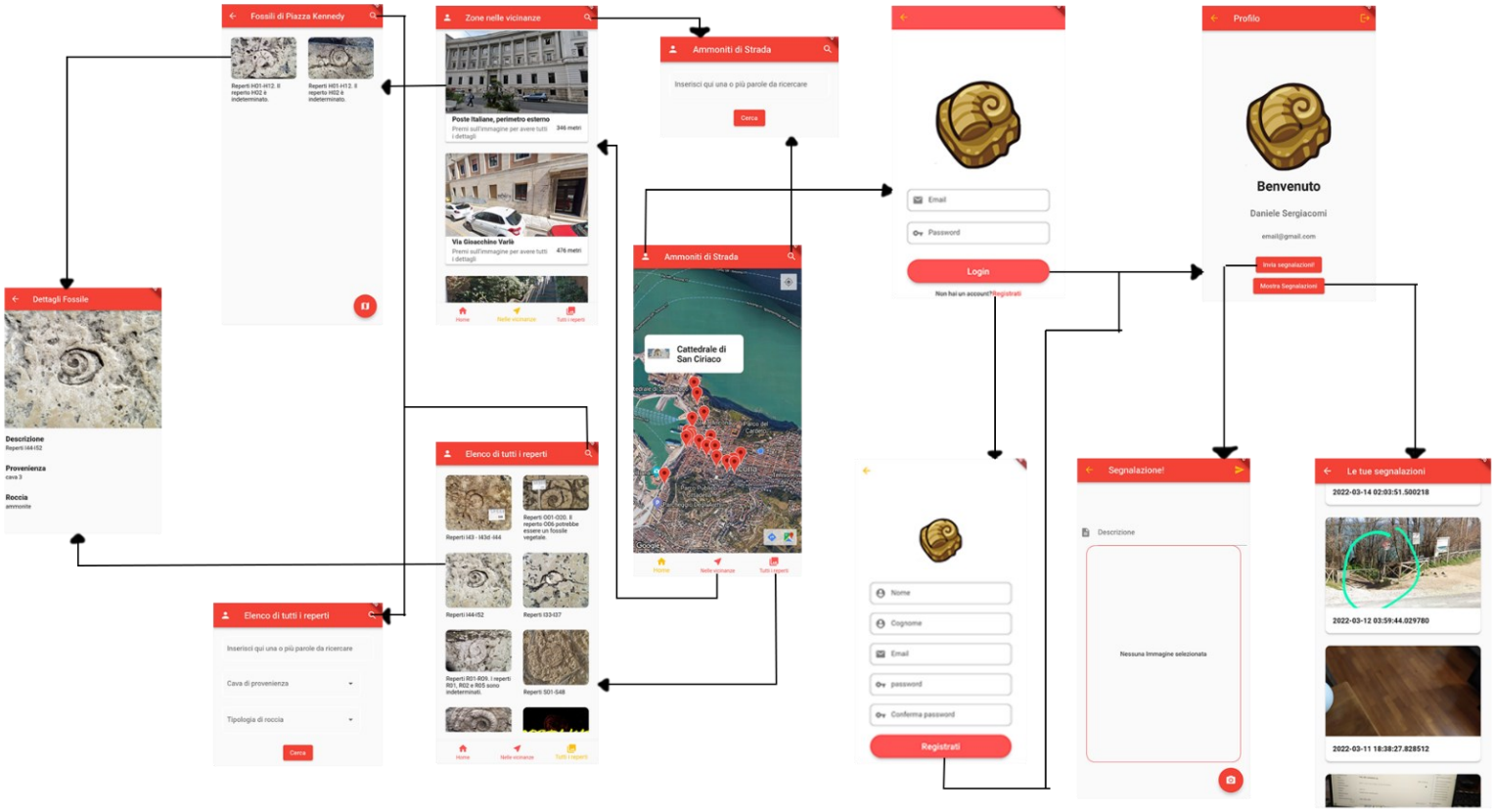

Figura 2.3: Collegamenti logici delle schermate

Questo diagramma contiene i collegamenti logici dell'applicazione, ovvero, a quale azione sullo schermo corrisponde una determinata schermata (ad esempio cliccando "Nelle Vicinanze" l'utente verrà mandato sulla schermata con la lista di tutte le zone in ordine dalla più vicina alla più lontana rispetto alla posizione corrente dell'utente). Purtroppo non è stato possibile mettere l'immagine dei collegamenti logici riguardante l'ultima versione dell'applicazione in quanto le frecce sarebbero risultate troppo annidate e non avrebbero permesso una visione organica dell'immagine.

CAPITOLO 3 – STRUMENTI UTILIZZATI

In questo capitolo verranno analizzati gli strumenti e le tecnologie che sono state utilizzate per la realizzazione di 'Ammoniti di strada'. Nello specifico:

- **Android Studio** come IDE di programmazione.
- **Dart** come linguaggio di programmazione.
- **Flutter** come framework di lavoro.
- **Firebase** per la memorizzazione e la gestione dei dati.
- **GitHub** per aiutare gli sviluppatori ad archiviare e gestire.

3.1| ANDROID STUDIO

Esistono degli strumenti e dei particolari software in grado di supportare gli sviluppatori durante la fase di programmazione dei propri applicativi. In primis vengono utilizzati degli ambienti di sviluppo integrati, detti anche IDE (Integrated Development Environment), che supportano i programmatori durante la fase di debugging del codice, spesso segnalando errori di sintassi in fase di scrittura. La maggior parte degli IDE presenta i seguenti tool integrati:

- Un *editor di codice sorgente*, Si tratta di un editor di testo che agevola la scrittura di codice software, grazie ad utili funzionalità come:
  - *Evidenziatore della sintassi* con suggerimenti visivi (syntax highlighter) che consente di presentare i diversi elementi sintattici del codice (proprietà, tag e attributi) in colori differenti per facilitare la scrittura e il controllo del codice.
  - *Strumenti di automazione della build*, per velocizzare le attività semplici o ripetitive nella fase di creazione di una build del software.
  - *Strumenti di individuazione di bug* durante la scrittura.

- Un *compilatore* (compiler) che traduce una serie di istruzioni (codice sorgente), scritte in un linguaggio di programmazione (linguaggio sorgente), in istruzioni (codice oggetto) di un altro linguaggio (linguaggio oggetto, o target).

- Un *interprete* (interpreter) che esegue direttamente il codice sorgente senza bisogno di tradurlo.

- Un *Linker* (o link editor) che effettua il collegamento tra il codice oggetto, e le librerie del linguaggio necessarie per l'esecuzione del programma.

- Un *Debugger*, un programma utilizzato per testare altri programmi; esso, infatti, è in grado di visualizzare graficamente la posizione di un bug nel codice originale. Il compito principale del debugger è quello di mostrare il frammento di codice macchina che genera il problema. Il codice può essere mostrato nella sua forma nativa, tradotto in linguaggio assembly o

perfino sotto forma di codice sorgente nel linguaggio di programmazione in cui il programma analizzato è stato scritto. Poiché la compilazione con debug inserisce nel programma grandi quantità di istruzioni in più, un programma eseguito in modalità debug è tipicamente molto più lento di quando è eseguito direttamente sul processore per cui è stato sviluppato. Quasi tutti i debugger consentono l'esecuzione del programma analizzato a piccoli passi, mostrando nelle interruzioni il codice relativo all'istruzione sorgente corrente e lo stato attuale della RAM ovvero lo stato o valore delle variabili associate alle rispettive nelle celle di memoria. L'interruzione dell'esecuzione può avvenire ad ogni singola istruzione, entrando eventualmente in altri sottoprogrammi, oppure in punti esplicitamente assegnati dall'utente (breakpoint).

Oggi uno degli IDE più utilizzati dagli sviluppatori è Android Studio [8], basato su IntelliJ IDEA [9]. È un software gratuito su licenza Apache 2.0 ed il download è disponibile per la maggior parte dei sistemi operativi in circolazione, come Windows, Linux e MacOS. Con la sua uscita Google ha voluto sostituire il precedente Eclipse mettendo a disposizione degli sviluppatori un ambiente di sviluppo sensibilmente migliorato sia dal punto di vista delle funzionalità che dell'efficienza nello sviluppare gli applicativi. inoltre il sistema garantisce una maggiore facilità di utilizzo, grazie ad una interfaccia più pulita e ottimizzata anche per gli utenti meno esperti (*Figura 3.1*). Le caratteristiche che rendono l'editor di Android Studio semplice, innovativo ed intuitivo, sono:

- Il sistema di build è flessibile e basato su *Gradle*.
- L'emulatore è veloce e ricco di funzionalità ed è possibile simulare i dispositivi Android direttamente sul computer in modo da poter testare l'applicazione su molteplici terminali e livelli *API Android* senza dover disporre di ogni dispositivo fisico.
- È presente un'integrazione con *GitHub.*
- È presente anche un'integrazione con *Firebase.*

Inoltre sono presenti come strumenti:

- La tab dei documenti.
- La barra di stato.
- L'area testuale.
- La barra laterale di validazione.
- Strumenti per la cattura di prestazioni, usabilità, compatibilità di versione e altri problemi.
- Funzionalità di Pro Guard e app-signing.
- Supporto per Google Cloud Platform e semplicità di integrazione con Google Cloud. Messaging e App Engine.

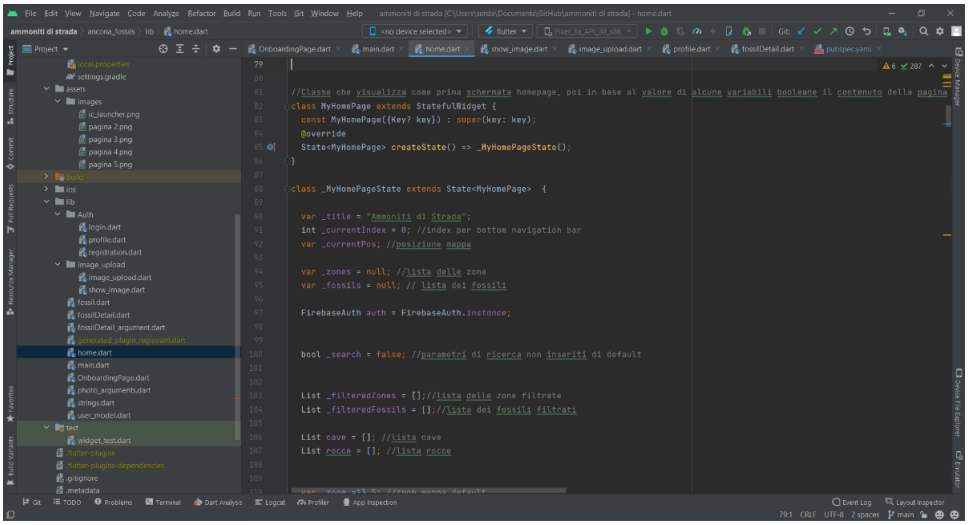

Figura 3.1: Interfaccia IDE Android Studio

La praticità di utilizzo è stata posta come uno degli obiettivi primari in questo ambiente. Infatti, l'IDE è molto comodo e veloce nell'interazione con l'utente. Altro aspetto degno di nota è che esiste un efficiente meccanismo di preview dei layout. L'aspetto dell'interfaccia viene velocemente mostrato in anteprima. È, inoltre, possibile visualizzare tale anteprima per diversi modelli di dispositivi così da valutare come l'interfaccia che si sta sviluppando si adatta a schermi di diverse proporzioni e grandezza (*Figura 3.2*).

Inoltre nello sviluppo di applicazioni è necessario scrivere molto codice; quindi, l'aspetto semplice ed intuitivo dell'ambiente di sviluppo rende tutto più facile e veloce, il che abbrevia i tempi di realizzazione dell'applicativo. Un'altra caratteristica fondamentale di questo IDE è la presenza di funzionalità che rendono più agevoli diversi aspetti come la navigazione all'interno del codice o la rilevazione degli errori che vengono evidenziati in tempo reale.

3.2| DART

Dart [10] (originariamente chiamato Dash) è un linguaggio di programmazione sviluppato da Google. È stato presentato alla conferenza "GOTO Aarhus 2011" dal 10 al 12 ottobre 2011. Dart è inoltre un linguaggio ad oggetti, con sintassi simile a Java, ed è ottimizzato per costruire interfacce utente, con possibilità di effettuare logging e debugging del codice.

Dart offre infatti varie feature importanti, come ad esempio la Sound Null Safety, una caratteristica di programmazione secondo la quale dei valori non possono essere nulli finché non viene detto esplicitamente dallo sviluppatore. Questo è un modo per evitare eccezioni e quindi crash dell'applicazione, reso possibile da un'analisi del codice prima dell'esecuzione.

Lo scopo di Dart è quello di "sostituire JavaScript come lingua franca per lo sviluppo web sulla piattaforma web aperto" – dal documento "Future of Javascript" doc from our internal "JavaScript Summit" last week[11] (Mail interna di Google trapelata).

Il compilatore Dart permette di scrivere programmi sia per il web che per desktop e server, attraverso l'uso di due diverse piattaforme:

- **Dart Native**: Per i dispositivi (smartphone, desktop, server, and altro), Dart Native include sia la Dart VM con una compilazione JIT (just-in-time) e un compilatore AOT (ahead-of-time) per la produzione di codice eseguibile.
- **Dart Web**: Per il web, Dart Web include sia un compilatore per lo sviluppo (dartdevc) che uno per la produzione (dart2js).

Esistono quattro modi per eseguire il codice Dart:

- **Compilato come JavaScript**: Per funzionare nei browser web tradizionali, Dart si affida a un compilatore da sorgente a sorgente per JavaScript. Dart è stato "progettato per essere strumenti di sviluppo facili da scrivere, adatto allo sviluppo di app moderne e capace di implementazioni ad alte prestazioni" - "Perché?", Dart lang (FAQ) [12]. Quando si esegue il codice Dart in un browser Web, il codice viene precompilato in JavaScript utilizzando il compilatore dart2js. Compilato come JavaScript, il codice Dart è compatibile con tutti i principali browser senza bisogno che i browser adottino Dart. Attraverso l'ottimizzazione dell'output JavaScript compilato per evitare controlli e operazioni costosi, il codice scritto in Dart può, in alcuni casi, essere eseguito più velocemente del codice equivalente scritto a mano utilizzando gli idiomi JavaScript.

- **Indipendente**: Il kit di sviluppo software (SDK) Dart viene fornito con una macchina virtuale Dart autonoma, che consente l'esecuzione del codice Dart in un ambiente di interfaccia della riga di comando. Poiché gli strumenti del linguaggio inclusi in Dart SDK sono scritti principalmente in Dart, la Dart VM autonoma è una parte fondamentale dell'SDK. Questi strumenti includono il compilatore dart2js e un gestore di pacchetti chiamato pub. Dart viene fornito con una libreria standard completa che consente agli utenti di scrivere app di sistema completamente funzionanti, come i server Web personalizzati.

- **Compilato in anticipo**: Il codice Dart può essere compilato AOT in codice macchina (set di istruzioni nativi). Le app create con Flutter, un SDK per app per dispositivi mobili creato con Dart, vengono distribuite negli app store come codice Dart compilato da AOT.

- **Nativo**: Dart 2.6 con compilatore dart2native per compilare in codice eseguibile nativo autonomo. Prima di Dart 2.6, questa funzione mostrava questa funzionalità solo su dispositivi mobili iOS e Android tramite Flutter. Nel caso di questo progetto, il codice Dart è stato eseguito in questo modo.

3.3| FLUTTER

Flutter [13] è un framework open-source creato da Google per la creazione di interfacce native per iOS e Android. Il suo vantaggio principale risulta essere la singola scrittura del codice, che poi può essere mandato in esecuzione sia su dispositivi Android che iOS (sviluppo cross-Platform).

Con la versione 1.9, Google introduce il supporto per le applicazioni web e per siti statici scritti in linguaggio Dart. Così, con questo aggiornamento, si è ampliata la lista delle piattaforme supportate.

Le app Flutter sono scritte nel linguaggio Dart e utilizzano molte delle funzionalità più avanzate del linguaggio.

In Flutter vi è un controllo su ogni pixel dello schermo. Gli elementi dello schermo (testi, icone, appbar…), cioè i Widget, sono costruiti da zero dal framework. Mentre nell'approccio tradizionale le componenti native sono direttamente visualizzate sullo schermo, in Flutter sono costruite pixel per pixel in modo tale da replicarle in modo perfetto. Questo rende possibile la programmazione cross-Platform e garantisce la stessa visualizzazione sui vari sistemi operativi. Sicuramente le performance risultano essere leggermente inferiori rispetto al caso delle piattaforme native, soprattutto perché l'interazione con i componenti e i servizi del dispositivo (camera, batteria, notifiche, GPS, microfono…) non avviene in modo diretto dal codice dell'app, ma vengono utilizzati dei Platform-Channels, cioè dei canali che permettono lo scambio di messaggi tra l'app e la piattaforma nativa.

Durante la scrittura e il debug di un'applicazione, Flutter viene eseguito nella Dart Virtual Machine (DVM), che dispone di un motore di esecuzione Just-in-Time. Ciò consente tempi di compilazione rapidi e "hot reload", con cui le modifiche ai file sono visibili durante l'esecuzione del programma.

Flutter estende ulteriormente questo concetto con il supporto di "hot reload" *con stato*, dove nella maggior parte dei casi le modifiche al codice sorgente si riflettono immediatamente nell'app in esecuzione senza richiedere un riavvio o alcuna perdita di stato.

Per prestazioni migliori, le versioni di rilascio delle app Flutter su tutte le piattaforme utilizzano la compilazione Ahead-of-time (AOT).

Nell'immagine sottostante troviamo l'architettura del framework Flutter (*Figura 3.2*)

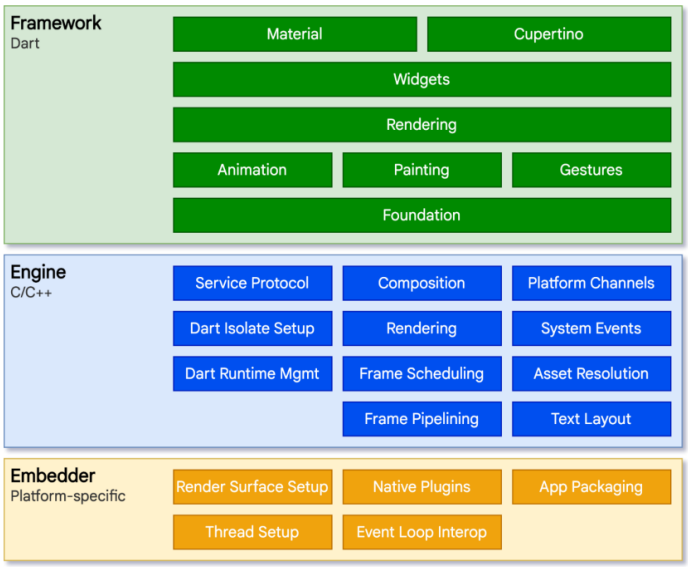

Figura 3.2: Architettura Flutter

Lo strato più vicino al sistema operativo è un Embedder, specifico per la tipologia di sistema operativo, che si occupa della distribuzione dell'app, cioè dell'incorporazione delle componenti per la compatibilità, del collegamento di plugin nativi e di altri servizi specifici per il sistema operativo. L'Embedder funge cioè da collegamento con le componenti native del sistema operativo; per questo è scritto nel linguaggio nativo della piattaforma, ovvero Java o C++ nel caso di Android.

Il core di Flutter è costituito dal Flutter Engine, scritto principalmente in C++, che fornisce tutti i metodi e gli strumenti di base necessari come supporto per lo sviluppo di un'applicazione Flutter.

Nell'Engine sono raccolte tutte le API di grafica, layout, generazione e sostituzione di frame e compilazione. L'Engine è esposto allo strato superiore, ovvero il Flutter Framework, tramite *'dart:ui'* una libreria che raggruppa tutto il codice C++ sottostante in delle classi Dart messe a disposizione del programmatore. Lo sviluppatore, infatti, interagisce con il Flutter Framework, scritto in Dart, potendo sfruttare la semplicità del linguaggio e la grande ricchezza per quanto riguarda classi e plugin esterni utilizzabili.

L'elemento principale di Flutter è lo Widget. Ogni elemento grafico è uno Widget, comprese le schermate e l'app; il compito dello sviluppatore è quindi comporre le schermate tramite Widget e gestirne poi la logica, i dati alla base e soprattutto l'interazione con l'utente, che porterà modifiche in ciò che viene visualizzato.

Flutter usa un'infinita varietà di Widgets per fornire un'applicazione completamente funzionante e performante. Questi Widgets fanno parte dell'architettura vera e propria di Flutter, quindi sostanzialmente si può dire che un'applicazione sviluppata in flutter altro non è che un insieme di Widget tutti annidati tra loro.

Sostanzialmente esistono tre tipi di Widget:

- **Stateful Widgets**: mantengono uno stato sotto forma di un insieme variabili. Se questo muta, viene ricostruito il widget stesso, riflettendo pertanto la modifica dello stato sull'interfaccia grafica.
- **Stateless Widgets**: agiscono come una costante e si creano sempre allo stesso modo una volta che sono stati configurati
- **Inherited Widgets**: funziona consentendo ad un altro Widget di annidarsi allo stato del Widget ereditato (genitore) consentendo che lo stato venga trasmesso ai suoi figli

Andando più nel dettaglio per quanto riguarda i Widget: uno Widget è un'istanza della classe che lo definisce. Esistono una serie di Widget messi a disposizione da Flutter, che sono disponibili nel Widget catalog. Poi ne possono essere definiti alcuni personalizzati nel codice, come ad esempio nel caso delle schermate, estendendo Stateful Widget o Stateless Widget. In ogni Widget è implementato il metodo build, che lo descrive in termini di composizione di altri Widget di livello più basso. In quelli con lo stato la funzione build si trova nell'istanza di stato, ovvero la classe che viene costruita automaticamente oltre allo Widget stesso, mentre in quelli senza stato si trova ovviamente nell'unica classe che rappresenta lo Widget. Il nome dell'istanza di stato è *'(nome Widget)***State**'. Uno Widget, dal punto di vista del codice, viene costruito istanziando una serie di

parametri. Più nello specifico, uno Widget è un oggetto, e la sua creazione consiste nell'istanziare la classe corrispondente passando valori ai suoi parametri. Ad ogni parametro può essere assegnato un valore vero e proprio, oppure un altro Widget, ed in questo modo che si creano le composizioni.

3.4| FIREBASE

Firebase è un servizio open source offerto da Google per semplificare lo sviluppo di applicazioni mobili e web. Sfrutta l'infrastruttura di Google e il suo cloud per fornire una suite di strumenti per scrivere, analizzare e gestire applicazioni cross-Platform. Consente agli sviluppatori di concentrarsi solo sul Front-end, permettendo di evitare tutta la gestione della parte di Back-end, aiutandoli inoltre nella gestione del server, delle chiavi API e nell'archiviazione dei dati. Il tutto è molto intuitivo così da facilitarne l'utilizzo ai meno esperti. In particolare gode delle seguenti proprietà:

- **Flessibilità**: il modello di dati adottato supporta strutture di dati flessibili e gerarchiche.
- **Interrogazione espressiva**: possono essere utilizzate delle query per recuperare singoli documenti specifici o tutti i documenti in una raccolta che corrispondono a dei parametri specificati (le query possono includere più filtri concatenati e combinare filtri e ordinamenti).
- **Aggiornamenti in tempo reale**: viene utilizzata la sincronizzazione dei dati per aggiornare i dati su qualsiasi dispositivo connesso (possono essere eseguite anche delle query di recupero semplici e una tantum in modo efficiente).
- **Supporto offline**: vengono memorizzati nella cache i dati che l'applicazione sta utilizzando attivamente, in modo che l'app possa scrivere, leggere, ascoltare ed eseguire query sui dati anche se il dispositivo `e offline; quando il dispositivo torna online vengono sincronizzate tutte le modifiche locali sul database.
- **Scalabilità.**

Tra i vari servizi forniti da Firebase vi è infatti Cloud Firestore, un database NoSQL a cui le app possono accedere direttamente tramite SDK nativi, quindi i dati vengono archiviati in documenti che contengono campi mappati a valori. Questi documenti sono archiviati in raccolte, che non sono altro che dei contenitori di documenti utilizzabili per organizzare i dati e creare query. I documenti supportano diversi tipi di dati, come stringhe, numeri semplici o complessi e oggetti nidificati. Possono anche essere create sotto-raccolte all'interno di documenti per costruire strutture di dati gerarchiche che si ridimensionano man mano che il database cresce.

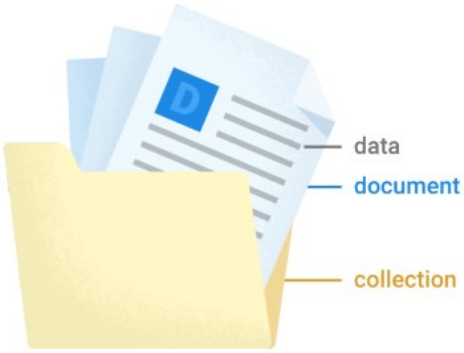

Figura 3.3: Modello di dati NoSQL adottato da Firestore

Firebase è facilmente integrabile in un progetto Flutter andando appunto a dichiararne le dipendenze.

Nell'app 'Ammoniti di Strada' si fa uso di due servizi forniti da Firebase, ovvero Cloud Firestore e Cloud Storage. Questi servizi possono essere utilizzati dalla Firebase Console, cioè l'interfaccia web che consente di sfruttare le varie applicazioni della piattaforma di Google.

L'interfaccia è rappresentata nella *Figura 3.4.*

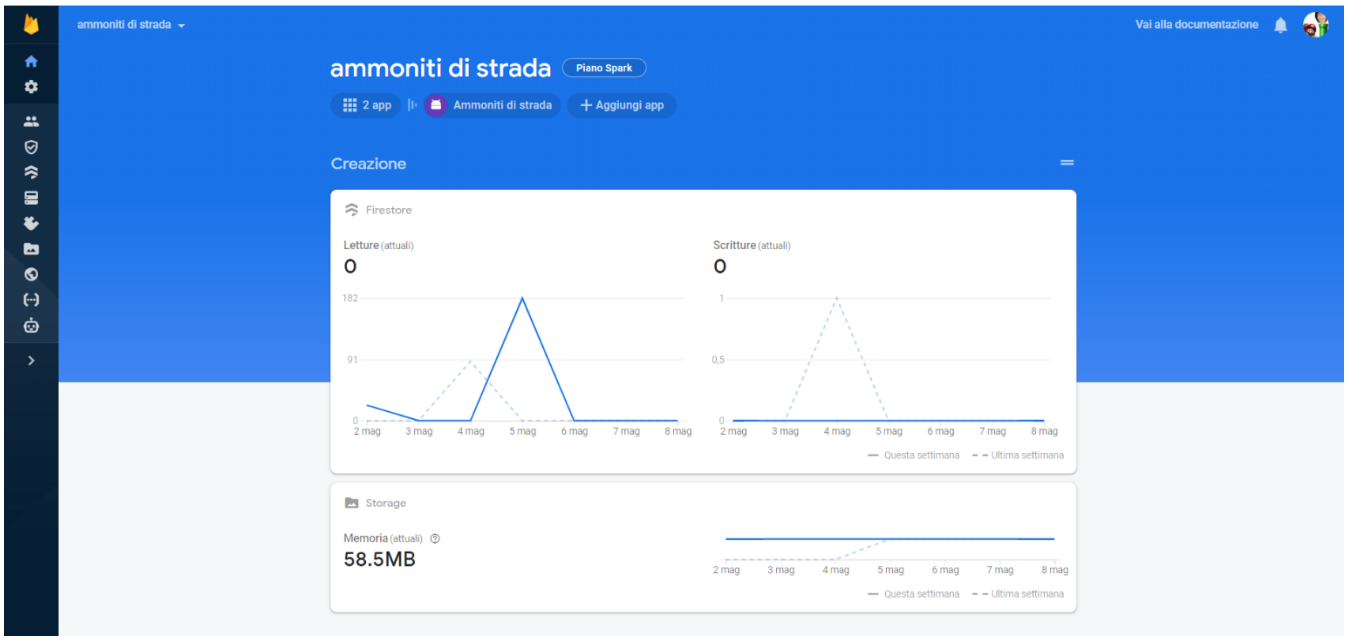

Figura 3.4: Interfaccia della Firebase Console

L'integrazione di questa piattaforma è stato un aspetto fondamentale nella progettazione dell'applicazione.

Tale piattaforma di sviluppo, in questo progetto, è stata utilizzata:

- Per fornire all'applicazione un database remoto contenente tutti di dati fondamentali sulle zone, sui fossili, e sulle segnalazioni effettuate dagli utenti.
- Per fornire all'applicazione un meccanismo semplice e veloce per autenticare e registrare gli utenti.

3.4.1| STORAGE

In Firebase possono essere archiviate diverse tipologie di file: immagini, documenti di testo e file in formato JSON. È inoltre disponibile un sistema di regole di sicurezza nel quale è possibile gestire i privilegi di scrittura e lettura al fine di proteggere tale archivio.

Per questo progetto la parte di Storage è stata utilizzata per caricare le foto che poi andranno visualizzate nell'app.

3.4.2| AUTENTICAZIONE

Offre diversi sistemi di autenticazione integrati come Google, Twitter, Facebook e molti altri, ma solitamente si utilizza quello basato su Email e Password, come nel caso dell'applicazione sviluppata.

3.5| GITHUB

GitHub [14] è un servizio web e cloud-based che aiuta gli sviluppatori ad archiviare e gestire il loro codice e a tracciare e controllare le modifiche apportate ad esso. In sostanza, rende molto più facile per individui e team utilizzare Git per il controllo delle versioni e la collaborazione. Attraverso la sua interfaccia è possibile visualizzare le varie ramificazioni (branch) su cui si sta lavorando, così come i commit (in pratica dei "salvataggi") di chiunque si stia adoperando per il progetto. Una volta apportate alcune modifiche, è possibile inviare il codice ad un branch facendo una pull request, che consiste fondamentalmente nel chiedere all'utente responsabile del branch di includere il proprio codice.

Per capire esattamente cos'è GitHub, è necessario introdurre due principi che vi sono alla base:

- Controllo delle versioni
- Git

3.5.1| CONTROLLO DELLE VERSIONI

Il controllo delle versioni aiuta gli sviluppatori a tracciare e gestire le modifiche al codice di un progetto software. Esso permette agli sviluppatori di lavorare in sicurezza attraverso il branching (ramificazione) e il merging (fusione). Con il branching, uno sviluppatore duplica parte del codice sorgente (chiamato repository); esso può quindi apportare in modo sicuro modifiche a quella parte del codice senza influenzare il resto del progetto. Infine, una volta che le modifiche funzionano correttamente, la sezione di codice che le contiene può essere fusa nel codice sorgente principale così da renderla ufficiale. Tutte queste modifiche vengono monitorate e, se necessario, possono essere ripristinate.

3.5.2| GIT

Git è un sistema di controllo versioni distribuito, il che significa che l'intero codice base e la cronologia sono disponibili sul computer di ogni sviluppatore; ciò permette di creare facilmente ramificazioni e fusioni.

3.6| PACKAGE E PLUGIN

Dopo aver scelto ambiente di sviluppo e linguaggio per il progetto, bisogna pensare a quali package possano servire all'app per poter essere in grado di offrire quei servizi che sono stati ricavati dai requisiti funzionali.

I package rendono possibile la creazione di codice modulare che può essere facilmente condivisibile. Un package è composto almeno dalle seguenti componenti: il '*pubspec.yaml'*, cioè un file di metadati che dichiara nome, versione ed autore del package; la directory '*lib'*, che contiene il codice pubblico del package, contenuto in almeno un file Dart. Il package, per essere utilizzato in un altro progetto Flutter, deve essere dichiarato nel '*pubspec.yaml*' di quest'ultimo, nella sezione '*dependencies'*.

I package possono essere di due tipi:

- **Dart Packages (package)**: package generici scritti in Dart. Alcuni di questi possono contenere delle funzionalità specifiche di Flutter, e le loro dipendenze risultano solo essere sul framework Flutter.
- **Plugin packages (plugin)**: un package Dart che contiene codice Dart combinato con una o più implementazioni specifiche per la piattaforma. I plugin possono cioè essere scritti per Android, iOS, web… oppure per una combinazione di sistemi operativi [15].

Flutter mette a disposizione molte librerie aggiuntive in Dart [16], facilmente integrabili nel progetto. Quando viene creato un progetto Flutter, automaticamente viene incluso il file '*pubspec.yaml'*, che viene inserito nel livello più alto delle cartelle per quanto riguarda la gerarchia del progetto. Questo file, scritto appunto nel formato *YAML*, specifica le dipendenze che il progetto richiede, cioè i package e i plugin, ma anche altre cose, come ad esempio font e percorsi per risalire alla posizione nel file system di immagini. È nel *pubspec* che si vanno quindi a definire le classi aggiuntive e i tool che l'applicazione utilizzerà per essere in grado di fornire all'utente determinate funzionalità.

```yaml
dependencies:
  google_maps_flutter: ^2.0.1
  geolocator: ^7.7.1
  photo_view: ^0.13.0
  giffy_dialog: ^1.8.0
  fluttertoast: ^8.0.9
  flutter_secure_storage: ^5.0.2
  shared_preferences: ^2.0.13

  flutter:
    sdk: flutter
  firebase_core: ^1.0.0
  firebase_auth: ^1.0.0
  cloud_firestore: ^1.0.0
  flutter_local_notifications: ^5.0.0+1
  custom_info_window: ^1.0.1
  workmanager: ^0.4.1
  image_picker: ^0.8.4+10
  image_slider: ^1.0.2
  geoflutterfire2: ^2.2.2
  exif: ^3.0.1
  smooth_page_indicator: ^1.0.0+2
  cupertino_icons: ^1.0.2
  firebase_storage: ^10.2.3
```

Figura 3.5: Dipendenze dichiarate nel file 'pubspec.yaml'

Nel caso di "Ammoniti di strada" sono stati utilizzato i seguenti package e plugin:

- *firebase_core*: un plugin per usare API Firebase Core, che attiva la connessione a molteplici servizi di Firebase.
- *cloud_firestore*: un plugin Flutter per usare il database Cloud Firestore.
- *firebase_storage*: un plugin che consente di utilizzare lo Storage di Firebase in un progetto Flutter.
- *firebase_auth*: un plugin Flutter che rende molto più di facile implementazione tutta la parte di programmazione riguardante il login, il logout e la registrazione di un nuovo account.
- *google_maps_flutter*: è un plugin che fornisce un Widget con la mappa di Google Maps. Una volta aggiunta la dipendenza, è possibile inserire la mappa nel Widget tree dell'app. La mappa è completamente personalizzabile, si possono andare infatti a modificare zoom, tipologia di mappa, posizione iniziale. La visualizzazione e le interazioni sono poi facilmente gestibili da un controller, il GoogleMapController.
- *flutter_local_notification*: è un plugin che consente all'app di inviare notifiche al dispositivo dell'utente. Questo plugin consente di personalizzare varie caratteristiche delle notifiche, si può infatti a settare la priorità, modificare il suono, oppure mandare una notifica periodicamente con intervalli di tempo a scelta.
- *workmanager*: Flutter Workmanager è un involucro del Workmanager nativo del sistema Android, che permette l'esecuzione in background di codice Dart. Il Workmanager di Android è infatti una componente con la quale si possono eseguire dei task in background, periodici o immediati, e con questo plugin lo si può integrare in un'applicazione Flutter.
- *geolocator*: è un plugin di geolocalizzazione che fornisce accesso ai servizi di localizzazione della piattaforma specifica (in questo caso Android). Permette di ottenere l'ultima posizione conosciuta oppure la posizione in tempo reale, con varie alternative di precisione, ma può anche calcolare la distanza tra due coordinate geografiche.

- *photo_view*: package che mette a disposizione uno Widget che rappresenta un'immagine o un contenuto zoomabile.
- *custom_info_window*: package che dà la possibilità di realizzare una finestra di informazione personalizzata per i marker nelle mappe di Google Maps.
- *giffy_dialog*: per creare dei dialog che non siano solo testuali, ma anche immagini gif (da implementare in futuro).
- *fluttertoast*: plugin utile per aver "pieno controllo" dei Toast in Flutter.
- *shared_preferences*: package utilizzato per la schermata di Onboarding, ovvero un tutorial generale sul funzionamento dell'app, e questo package permette di ricordare all'app che dall'installazione questa schermata dovrà essere visualizzata solo una volta.
- *image_picker*: package che consente di interagire con la fotocamera del dispositivo.
- *image_slider*: package che permette di scorrere un insieme di foto facendo sliding verso destra o verso sinistra dallo smartphone
- *geoflutterfire2*: package di Flutter che è stato utile per prendere dei dati di latitudine e longitudine separati e convertirli in un unico GeoPoint.
- *exif*: package che non è stato ancora implementato in quanto servirebbe per estrapolare le informazioni exif riguardanti la latitudine e la longitudine delle foto presenti nella galleria del dispositivo, ma ancora non risulta di facile programmazione e presenta molte imperfezioni ed errori.
- *smooth_page_indicator:* un semplice package che rende più gradevole alla vista lo scorrimento delle immagini quando ci si trova nella schermata di tutorial (Onboarding).
- *cupertino_icons*: plugin che è servito per cambiare l'immagine dell'applicazione.

CAPITOLO 4 – PROGETTAZIONE

In questo capitolo verranno descritte le fasi di progettazione dell'app.

Una volta stabilite la struttura dei dati che l'app utilizzerà (in modo tale da sapere come trattarli nel codice e quindi poterli gestire in modo efficiente), le funzionalità da garantire e gli strumenti che verranno utilizzati, si deve andare a definire l'architettura dell'app, cioè le componenti software principali e le modalità con cui esse comunicheranno fra loro.

La fase centrale sarà quella di definizione del funzionamento vero e proprio delle classi. In questa parte sarà descritta la logica dell'applicazione, quindi i funzionamenti dei vari metodi e di come essi possano utilizzare i dati e gestire le interazioni dell'utente per modificare dinamicamente l'interfaccia.

Infine, si dovrà andare a scrivere il codice, l'ultima fase sarà quindi quella dell'implementazione. Questa fase consiste nell'utilizzare tutti gli strumenti a disposizione e, tramite il codice Dart, implementare dei metodi che svolgano il lavoro necessario al fine del funzionamento dell'app.

4.1| ARCHITETTURA DELL'APPLICAZIONE

In questa sottosezione l'obiettivo sarà quello di definire, in modo sintetico e chiaro, l'architettura alla base dell'applicazione. Considerando che tutta l'architettura viene ricondotta dal framework stesso ai Widget, si è scelto di associare ad ogni funzionalità dell'applicazione un Widget specifico. Non si può quindi parlare di un vero e proprio modello architetturale, ma piuttosto di un insieme di Widget distinti interoperanti tra loro, essendo Flutter costituito da una serie di strati (layer), ognuno dei quali comunica con quello sottostante e ne sfrutta i servizi messi a disposizione.

Per quanto riguarda l'architettura dell'app 'Ammoniti di Strada' (*Figura 4.1*), si devono distinguere due componenti principali. La prima è la parte che racchiude logica e grafica dell'applicazione, costituita dalle classi definite nei file Dart del progetto Flutter. Le classi, come detto in precedenza, sono dei Widget che rappresentano poi effettivamente le schermate dell'app e ne gestiscono la logica.

La seconda componente fondamentale dell'applicazione è la sorgente persistente di dati, formata dai database messi a disposizione tramite i servizi Firebase. Le classi Dart interagiscono con i database (Cloud Firestore e Firebase Storage) utilizzando dei package disponibili in Flutter, e una volta prelevati i dati li possono utilizzare per popolare le varie schermate.

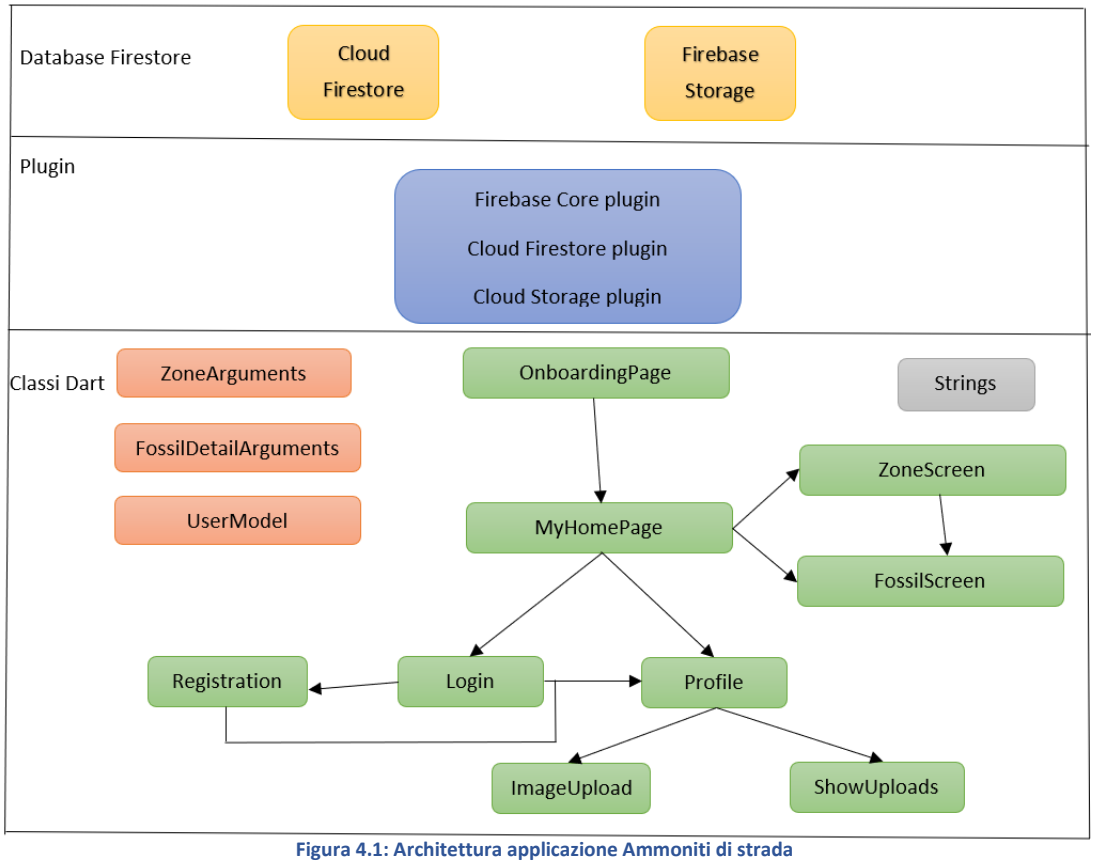

Figura 4.1: Architettura applicazione Ammoniti di strada

I file dart sono stati organizzati in questo modo:

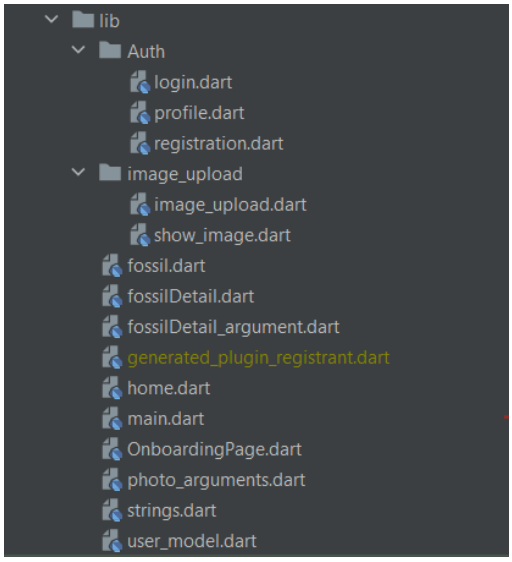

Figura 4.2: Organizzazione classi dart

- *MyHomePage*: definisce la schermata principale dell'applicazione, in cui viene iniettato il contenuto che è richiesto di volta in volta dall'utente. Questa classe (come tutte le altre classi dell'applicazione) utilizza il concetto di stato per modificare ciò che viene visualizzato in base all'interazione dell'utente. Di default viene mostrata la mappa contenente i marker delle zone che contengono i reperti, ma poi in base a cosa si preme sulla barra di navigazione in basso possono essere visualizzate anche la lista delle zone nelle vicinanze (con un raggio di 3km), la pagina con la lista di tutti i fossili o tutte le segnalazioni effettuate da un utente che si è autenticato (solo se precedentemente si è effettuato il Login, altrimenti si aprirà la

schermata di Login per permettere all'utente di autenticarsi). È presente anche una Floating Action Bar in fondo a destra che apparirà solo se l'utente è loggato e che permette di inviare delle segnalazioni. In MyHomePage sono implementate due funzioni di ricerca, uno per le zone (per la schermata 'Home' e 'Zone nelle vicinanze') e uno per i fossili (nella schermata 'Tutti i reperti'). Inoltre in questa classe si instaura il collegamento con il database di Firebase e preleva i dati necessari. Il download viene fatto solo una volta: poi i dati vengono salvati in locale in modo da averli subito disponibili quando devono essere utilizzati. Questa classe gestisce anche la navigazione verso altre schermate separate a quelle appena descritte, ovvero la classe contenente la lista dei fossili associati ad una determinata zona che si è voluto selezionare, denominata '*ZoneScreen'*, e quella del singolo reperto, denominata '*FossilScreen'*, poi quella di '*Login'* (se l'utente non è loggato) o, se l'utente è già loggato, quella di '*Profilo',* di 'invia segnalazioni' o di 'mostra segnalazioni' (se l'utente è già loggato).

- *OnboardingPage*: è la schermata introduttiva di tutorial e spiega in modo sintetico il funzionamento dell'app, con il Package shared_preferences, è stato possibile fare in modo da far vedere la schermata di Onboarding solo al primo accesso dopo l'installazione, dopodiché la schermata iniziale sarà sempre a comunque MyHomePage.

- *ZoneScreen*: definisce la schermata in cui viene mostrata la lista dei fossili che appartengono ad una determinata zona che è stata cliccata in precedenza, e questa schermata viene visualizzata in seguito al tocco dell'utente su un elemento della lista della schermata 'Zone nelle vicinanze' oppure su una finestra di informazione della mappa delle 'Home'. La navigazione da '*MyHomePage'* e '*ZoneScreen'* si serve di un'altra classe: '*ZoneArguments'*.

- *ZoneArguments*: è la classe che funge da contenitore per i dati che dovranno essere trasportati dalla home alla schermata della lista dei fossili di una zona. Così ogni volta che l'utente vuole accedere alla lista, i dati vengono salvati in un'istanza di questa classe e poi vengono recuperati e utilizzati per popolare la nuova pagina, in particolare informazioni relative a:
    - Nome della zona.
    - Descrizione della zona.
    - Latitudine.
    - Longitudine.
    - Id (univoco).
    - Link url dell'immagine.

- *FossilScreen*: definisce la schermata in cui vengono rappresentate le informazioni e le foto di un singolo reperto. Questa schermata può essere acceduta in seguito al tocco dell'utente su un elemento della lista di 'Tutti i reperti' (*MyHomePage*) oppure su un elemento della lista della classe '*ZoneScreen'*. La navigazione da '*MyHomePage'* (o '*ZoneScreen'*) a '*FossilScreen'* si serve di un'altra classe: '*FossilDetailArguments'*.

- *FossilDetailArguments*: è la classe che funge da contenitore per i dati che dovranno essere trasportati dalla '*MyHomePage'* o dalla schermata del singolo reperto. Ogni volta che l'utente vuole vedere le informazioni di un reperto in particolare, i dati vengono salvati in un'istanza di questa classe e poi vengono recuperati e utilizzati per popolare la nuova pagina, in particolare informazioni relative a:
    - Descrizione del reperto.
    - Cava di provenienza del fossile.
    - Tipo di roccia.
    - Link url dell'immagine del fossile.
    - Id della zona a cui è associato il fossile.

- *Login*: Schermata accessibile dalla '*MyHomePage'* <u>se l'utente non è ancora loggato</u>, e permette l'autenticazione per poter usufruire di ulteriori funzioni (che verranno spiegate più avanti), una volta effettuato il Login inserendo email e password, l'utente finirà sulla schermata di profilo, se invece l'utente non ha un account potrà registrarsi e creare un nuovo profilo.

- *Registration*: Schermata accessibile dal Login, in cui sarà possibile creare un nuovo account inserendo nome, cognome, email, password e conferma password.

- *UserModel*: è la classe che funge da contenitore di dati che dovranno essere trasportati dalla pagina di Registrazione a Firebase per l'immagazzinamento del nuovo utente, e per recuperare da Firebase le informazioni dell'utente loggato nella pagina di profilo e nell'applicazione durante tutta la sessione (fino al logout).

- *Profile*: Schermata accessibile dalla '*MyHomePage'* <u>se l'utente è loggato</u>, da qui sarà possibile effettuare il logout e ritornare nella homepage come utente non loggato, oppure finire nelle ultime due classi:
    - *ImageUpload*: per inviare delle segnalazioni ai ricercatori di fossili con foto ed un'eventuale descrizione per aiutare la ricerca del reperto per i ricercatori.
    - *ShowUploads*: per vedere tutte le segnalazioni precedentemente effettuate dall'utente in ordine di data.

- *Strings*: è una classe che contiene dei dati statici utili all'applicazione, come ad esempio testi che servono per alcune parti dell'interfaccia.

L'applicazione utilizza poi due fonti persistenti di dati:

- **Cloud Firestore**: è un database remoto messo a disposizione da Firebase, in cui è possibile salvare dati in documenti raccolti in delle collection (*collezioni*).
- **Firebase Storage**: è un servizio di archiviazione remoto di oggetti, particolarmente utilizzato per foto e video. Il database che l'app utilizza è utilizzato appunto per salvare le *foto* di tutte le zone, fossili e segnalazioni presenti in Firestore.

La classe '*MyHomePage'* non può comunicare direttamente con i servizi Firebase, ma ha bisogno di alcuni plugin esterni aggiuntivi. Per questo vengono utilizzati dei *package*, ovvero delle componenti software aggiuntive che permettono la comunicazione tra il codice Dart e i database. In particolare, vengono utilizzati i seguenti package, definiti prima:

- *firebase_core.*
- *cloud_firestore.*
- *firebase_storage.*
- *firebase_auth.*

4.2| PROGETTAZIONE DEI DATI

Questa sezione è dedicata a descrivere il modello utilizzato per la rappresentazione dei dati relativi ai reperti. Come detto in precedenza, l'applicazione 'Ammoniti di strada' utilizza due database che fanno parte dei servizi Firebase, ovvero **Cloud Firestore** e **Firebase Storage**. Per poter utilizzare i servizi Firebase, si deve prima creare un progetto e poi collegarlo al proprio progetto Flutter. Questo avviene inserendo nella directory *android/app* il file '*google-services.json*', fornito automaticamente da Firebase.

Nel progetto saranno presenti anche altre collection riguardanti versioni embrionali dell'app, che ho comunque deciso di tenere nel caso in cui in futuro si vogliano riprendere quei particolari dati, esse sono ('*ammoniti'* e '*fossili'*).

In particolare, il database è strutturato nelle seguenti raccolte, come mostrato nella *Figura 4.3*:

- **Zone**: per rappresentare tutti i luoghi di interesse in cui sono presenti fossili.
- **Dettaglio**: per rappresentare i singoli fossili.
- **Segnalazioni**: per contenere tutte le segnalazioni effettuate dagli utenti.
- **Users**: per immagazzinare tutti gli utenti registrati in Firestore.

Figura 4.3: Organizzazione delle collection nel Cloud Firestore

Per quanto riguarda le ***zone***, sono stati aggiunti i seguenti campi (*Figura 4.4*):

- **descr** (*stringa*): ovvero la descrizione della zona.
- **fotozona** (*stringa*): dove è rappresentato il nome della foto presente nello Storage di Firebase (di cui parleremo più avanti), e che poi su Flutter verrà convertito in url in modo da permettere all'utente di visualizzare l'immagine dall'app.
- **id** (*stringa*): è rappresentato anche l'id generato automaticamente da Firebase in modo da poter capire successivamente a quale fossile corrisponde una determinata zona.
- **lat** (*double*): per la latitudine.
- **long** (*double*): per la longitudine.
- **nomezona** (*stringa*): il nome della zona.

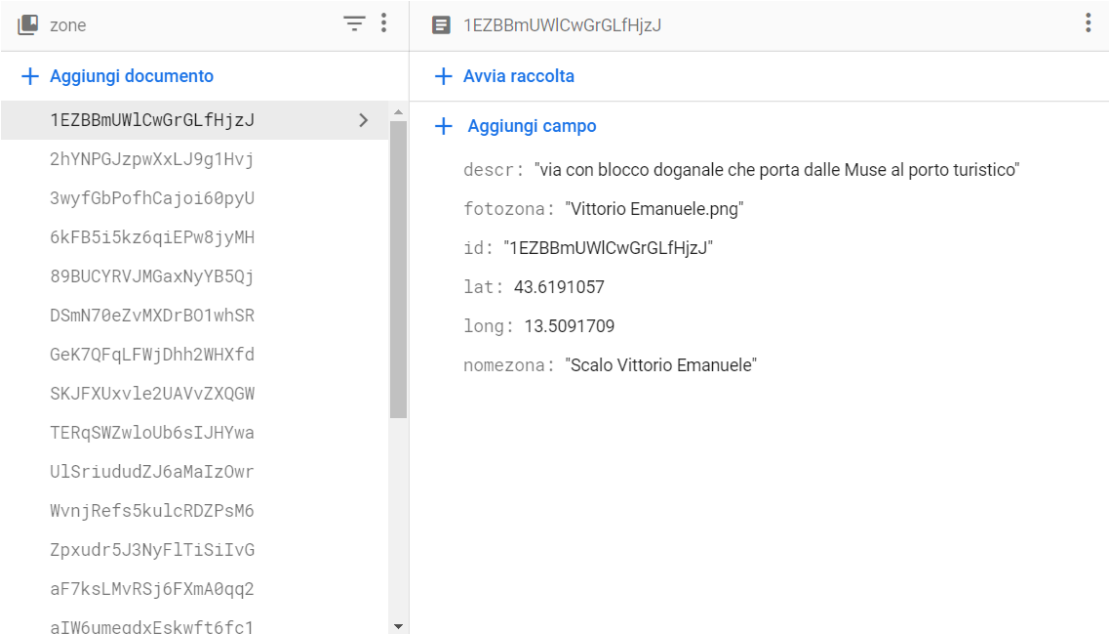

Figura 4.4: Documento delle zone nel database Cloud Firestore

Invece per la collection **Dettaglio** (*Figura 4.5*):

- **dateTime** (*timeStamp*): un timeStamp per rappresentare in che data e ora è stato aggiunto un fossile al DB.
- **descrfossile** (*stringa*): ovvero la descrizione del fossile.
- **fotofossile** (*stringa*): dove è rappresentato il nome della foto del fossile (come per la zona).
- **fotofossile2** *(stringa)*: rappresenta la seconda foto del fossile per permettere all'utente di trovare prima la posizione del reperto, anch'esso è rappresentato dal nome della foto del fossile.
- **provenienza** (*stringa*): che rappresenta a quale cava appartiene un fossile.
- **roccia** (*stringa*): che rappresenta il tipo di roccia del fossile.
- **zona** (*stringa*): dove è rappresentato l'id della zona che lo contiene. (relazione uno a molti tra la zona e il fossile).

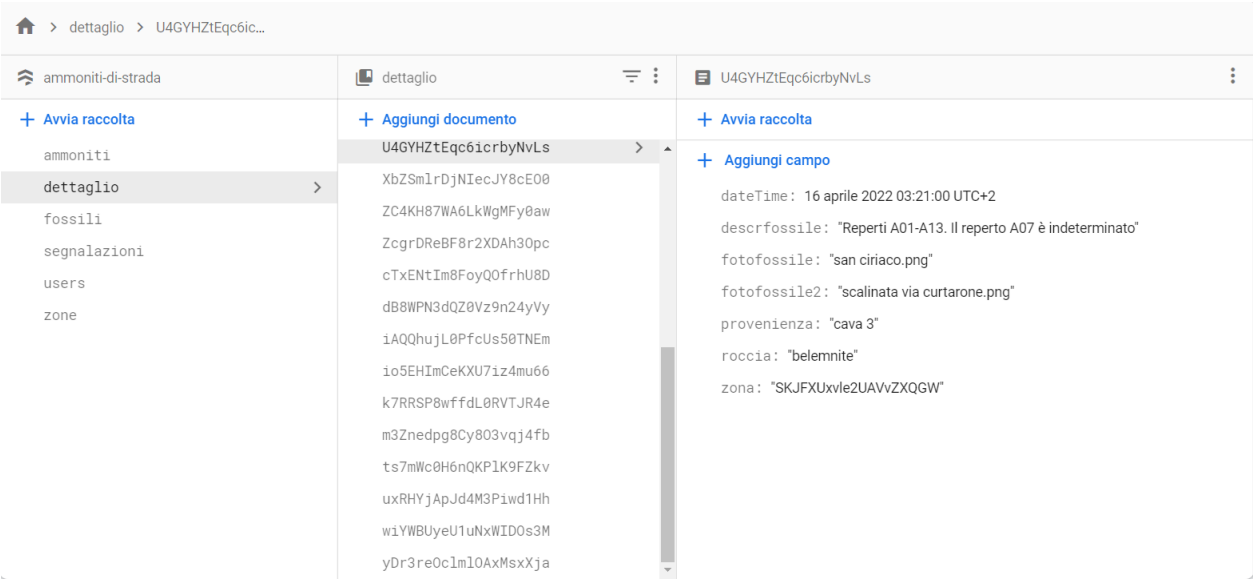

Figura 4.5: Documento dei fossili nel database Cloud Firestore

Invece per la collection **Segnalazioni** (*Figura 4.6*):

- **data** (*timeStamp*): un timeStamp per rappresentare in che data e ora è stata fatta la segnalazione del fossile.
- **descrizione** (*stringa*): una descrizione che l'utente può mettere facoltativamente per agevolare la ricerca del fossile da parte dei ricercatori.
- **downloadURL** (*stringa*): che rappresenta direttamente l'url della foto, invece del nome (come fatto prima).
- **position** (*punto geografico*): posizione istantanea dell'utente presa nel momento in cui viene scattata la foto.
- **user** (*stringa*): dove è rappresentato l'id dell'utente che ha eseguito la segnalazione, per far vedere nella sezione "mostra segnalazioni" tutte le segnalazioni di quel particolare utente.

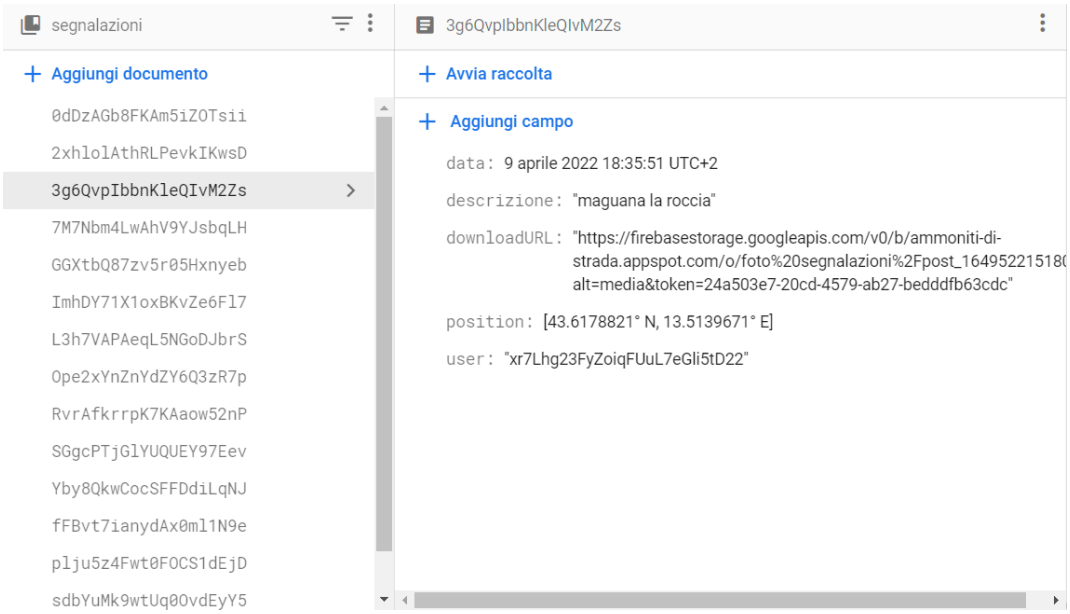

Figura 4.6: Documento delle segnalazioni nel database Cloud Firestore

E infine per gli **Users** (*Figura 4.7*):

- **email** (*stringa*): l'email dell'utente.
- **firstName** (*stringa*): il nome dell'utente.
- **secondName** (*stringa*): il cognome dell'utente.
- **uid** (*stringa*): è rappresentato anche l'id generato automaticamente da Firebase in modo da poter capire successivamente a quale utente corrisponde una segnalazione (relazione uno a molti tra l'id utente e l'id della segnalazione).

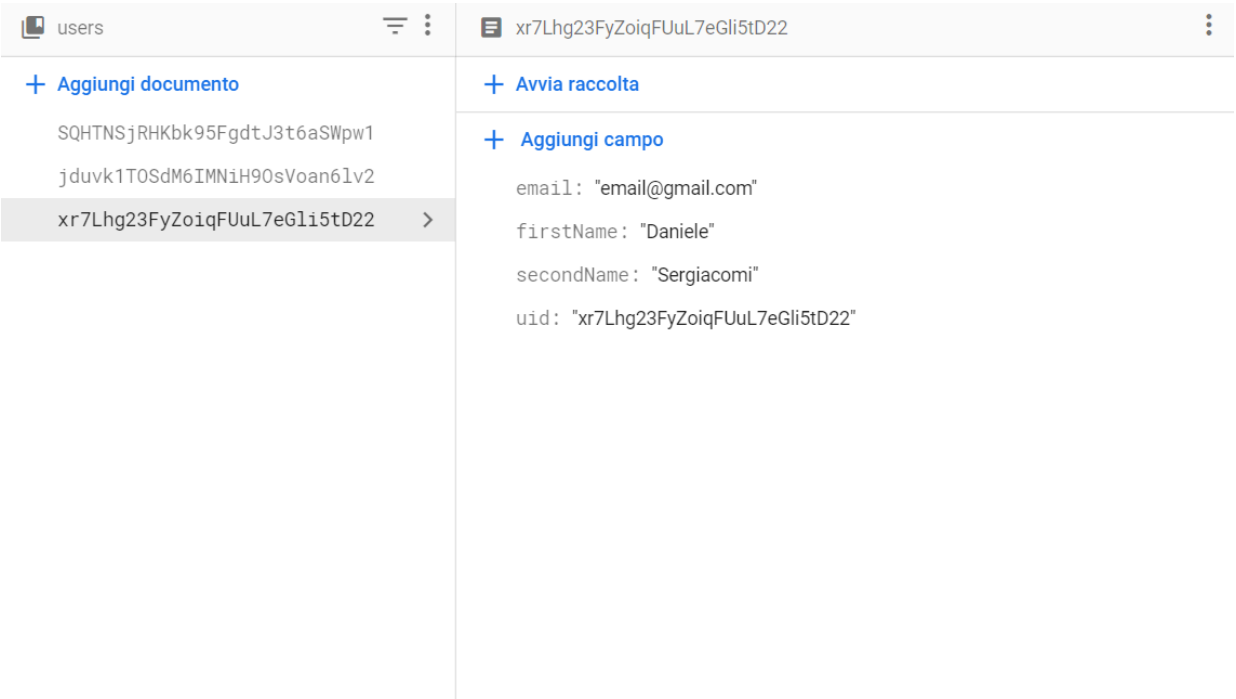

Figura 4.7: Documento degli utenti nel database Cloud Firestore

Lo Storage (*Figura 4.8*) è molto più semplice, in quanto viene utilizzato esclusivamente come servizio di archiviazione di oggetti, in questo caso saranno rappresentate le foto delle zone, dei fossili (*Figura 4.9*), e delle segnalazioni dell'utente (*Figura 4.10*), rappresentate rispettivamente nelle cartelle come:

- *foto zona 2/.*
- *foto dettaglio/.*
- *foto segnalazioni/.*

foto dettaglio/	–	Cartella	–
foto segnalazioni/	–	Cartella	–
foto zona 2/	–	Cartella	–

Figura 4.8: Cartelle dello Storage

gs://ammoniti-di-strada.appspot.com > foto dettaglio

Carica file

Nome	Dimensioni	Tipo	Ultima modifica
chiesa san francesco.png	415.33 KB	image/png	15 apr 2022
corso stamira.png	387.46 KB	image/png	15 apr 2022
fossile 1.png	648.02 KB	image/png	8 apr 2022
fossile 2.png	766.16 KB	image/png	8 apr 2022
fossile 3.png	277.62 KB	image/png	8 apr 2022
guglielmo marconi.png	489.15 KB	image/png	15 apr 2022
largo sacramento.png	466.16 KB	image/png	15 apr 2022
monumento caduti.png	452.9 KB	image/png	15 apr 2022
piazza kennedy 2.png	468.32 KB	image/png	15 apr 2022

Figura 4.9: Contenuto della cartella delle foto dei fossili

gs://ammoniti-di-strada.appspot.com > foto segnalazio...

Carica file

Nome	Dimensioni	Tipo	Ultima modifica
post_1646940483846	424.3 KB	image/jpeg	10 mar 2022
post_1646945653890	280.12 KB	image/jpeg	10 mar 2022
post_1647020307828	188.41 KB	image/jpeg	11 mar 2022
post_1647053984029	181.48 KB	image/jpeg	12 mar 2022
post_1647204198347	577.23 KB	image/jpeg	13 mar 2022
post_1647219831500	61.68 KB	image/jpeg	14 mar 2022
post_1647223558207	455.86 KB	image/jpeg	14 mar 2022
post_1647223573124	830.25 KB	image/jpeg	14 mar 2022

Figura 4.10: Contenuto della cartella delle foto delle segnalazioni

4.3| LOGICA DELL'APPLICAZIONE

4.3.1| LA FUNZIONE MAIN

Come in ogni applicazione realizzata in Flutter, l'entry point è la funzione **main**, dichiarata nel file '*main.dart*'. In questa funzione si vanno a inizializzare i servizi Firebase, cosicché poi durante l'esecuzione sia possibile utilizzarli. Successivamente, sempre dentro la funzione main, viene lanciata l'applicazione; questo avviene con il metodo '*run (MyApp ())*'.

MyApp, che rappresenta l'istanza dell'applicazione, è, come tutto in Flutter, uno Widget.

In MyApp sono definite delle **rotte**, che serviranno poi durante l'esecuzione dell'app per navigare da una schermata all'altra. Al primo avvio dell'applicazione la rotta iniziale è '*/onboarding'*, la quale istanzia la schermata iniziale di tutorial, tuttavia ciò avverrà solo una volta, in quanto con le *shared_preferences* si è riusciti ad istanziare una variabile '*initScreen'* che di default è settata a 0 che è in grado di memorizzare il primo avvio cambiando permanentemente il valore di questa variabile a 1 che poi cambierà la rotta iniziale che diventerà '/home', la quale istanzierà il Widget '*MyHomePage'*, cioè la schermata in cui inizialmente viene visualizzata la mappa.

4.3.2| LO WIDGET ONBOARDINGSCREEN

È la schermata di tutorial introduttivo, comprende 5 pagine, ed è possibile scorrerle una ad una per leggerne il contenuto, oppure è possibile saltare direttamente all'ultima pagina, dove sarà presente un tasto "INIZIA" che ci porterà nella Homepage.

Le varie immagini della schermata si trovano nella cartella assets, qui dichiarate nel '*pubspec.yaml*' (*Figura 4.11*), assieme al logo dell'app:

```
assets:
  - assets/images/ic_launcher.png
  - assets/images/pagina 2.png
  - assets/images/pagina 3.png
  - assets/images/pagina 4.png
  - assets/images/pagina 5.png
```

Figura 4.11: Dichiarazioni delle immagini della cartella assets

4.3.3| LO WIDGET MYHOMEPAGE

'*MyHomePage'* è uno Stateful Widget, definito nel file '*home.dart*'. La schermata è composta da più parti, alcune delle quali sono statiche e altre sono invece create dinamicamente tramite dei metodi. La **Bottom bar**, ad esempio, è un elemento statico, perché è sempre presente, anche quando l'utente interagisce. Il **Body** (contenuto centrale), l'**App bar** e la **Floating Action Bar** sono invece

creati tramite dei metodi, perché dipendono dall'interazione dell'utente. Quando l'utente interagisce infatti, vengono cambiati i valori di alcune variabili di stato, e queste modifiche determinano la ricostruzione da zero dell'interfaccia.

Come già detto precedentemente, la schermata principale, che parte inizialmente con la visualizzazione della mappa, può modificare il suo contenuto. Questo avviene per esempio quando l'utente preme su una delle quattro icone della barra di navigazione posizionata in basso, ovvero 'Home', 'Zone nelle vicinanze', 'Tutti i reperti' e 'Mostra segnalazioni'. Quando questo accade, viene eseguito il metodo *onTabTapped*, che modifica il contenuto della variabile di stato *_currentIndex*, mettendolo a **0**, **1**, **2** o **3** in base all'icona sulla quale si è premuto.

Il metodo *bodySetter*, in base al valore della variabile *_currentIndex*, inietta nel **Body** un contenuto diverso. Se il valore di *_currentIndex* è **0** viene visualizzata la mappa, se è **1** la lista con le zone nelle vicinanze, se è **2** l'elenco con tutti i reperti, mentre se è **3** viene aperta una nuova classe Dart relativa alla lista delle segnalazioni effettuate dall'utente, ma solo se l'utente è autenticato perché in alternativa finirà nella schermata di Login.

Un'altra variabile di stato che determina cioè che viene visualizzato è *_search*, una variabile booleana, che può assumere cioè come valori **vero** o **falso**. Il suo valore è settato a vero quando l'utente preme sull'icona della lente di ingrandimento presente sull'**App bar**. In *bodySetter* viene controllato il valore di questa variabile, se il valore è vero, allora viene visualizzata la pagina di ricerca, inoltre siccome nella 'Home' e in 'Zone nelle vicinanze' sono presenti le zone e in 'tutti i reperti' invece vengono visualizzati prettamente i fossili, allora si è pensato di mettere una condizione nella funzione di ricerca, ovvero e *_currentIndex* è uguale a **0** o **1** allora la funzione di ricerca varrà solo per il nome e la descrizione della zona, mentre se invece _currentIndex è uguale a **2** allora la funzione di ricerca varrà per la descrizione del fossile, la cava di provenienza e il tipo di roccia del reperto.

Come altra variabile di stato abbiamo *isFabVisible* di tipo **bool**, il cui valore cambia a seconda che l'utente sia loggato (**true**) o meno (**false**), e ciò renderà visibile o invisibile la FAB dinamicamente, se sarà visibile, ovvero, l'utente è autenticato, il click su quel bottone permetterà all'utente di inviare le segnalazioni.

Per l'**App bar** vale lo stesso discorso del **Body**, ovvero esiste un metodo chiamato *appBarSetter*, che crea e restituisce l'istanza dello Widget, anche se in realtà nello Widget '*MyHomePage'* l'**App bar** rimane sempre costante dal punto di vista degli elementi. Ciò che varia sono:

- *il titolo*, cioè il testo raffigurato al centro della barra; per il titolo infatti esiste un metodo denominato *titleSetter*, il quale, in base al valore di *_currentIndex* (e quindi in base alla schermata visualizzata in un determinato momento), modifica il valore del testo.
- E *l'icona di Login*, dove se l'utente non è autenticato verrà mandato nella schermata di login, mentre se invece lo è verrà mandato nella schermata personale di profilo.

4.3.4| I METODI GETZONES, GETFOSSILS E CREATEMARKERS

Tre metodi molto importanti, utilizzati nel *bodySetter* e non solo, sono *getZones*, *getFossils* e *createMarkers*.

getZones si occupa di recuperare i dati relativi alle zone presenti nel database Cloud Firestore: prima di tutto in *getZones* viene controllato se l'utente ha fornito il permesso alla localizzazione, in quanto questo metodo ha anche il compito di calcolare la posizione dell'utente, in seguito si va a vedere se la geolocalizzazione è effettivamente attiva sul dispositivo. Una volta fatto questo controllo, vengono scaricati tutti i dati dei documenti presenti nella collection '*zone*' in Firestore. Oltre a questo, vengono salvati anche tutti gli url delle foto presenti nello Storage, in modo da averli disponibili quando dovranno essere mostrate le immagini dei reperti. Il download dei dati da Firestore viene comunque fatto solo una volta, perché poi vengono salvati nella variabile locale *_zones*. Il metodo fa quindi prima un controllo su questa variabile perché, nel caso in cui i dati siano già stati salvati, i database di Firebase non vengono per niente contattati. In questo modo il recupero dei dati necessari risulta molto più veloce, in quanto si trovano in locale. Oltre al controllo sulla variabile *_zones*, ne viene effettuato uno sulla variabile *_filteredZones*. A questa variabile vengono assegnati gli eventuali reperti che risultano dalla ricerca effettuata dall'utente. Nel caso in cui ci siano dei valori, cioè c'è appena stata una ricerca, il metodo ritorna indietro esclusivamente le zone trovate. La funzione *getZones* viene utilizzata sia nella lista dei reperti nelle vicinanze, sia nella home. Nel caso della lista che visualizza i reperti nelle vicinanze, viene poi effettuato un altro controllo, per verificare quali reperti si trovino nei pressi dell'utente.

getFossils, invece, esegue un compito simile alla funzione precedente, ma si occupa di recuperare i dati relativi ai reperti presenti nel database Cloud Firestore, inoltre non essendo i fossili georeferenziati, non c'è alcun bisogno di fare controlli sui permessi alla localizzazione e soprattutto non c'è bisogno di calcolare la posizione dell'utente. Quindi vengono direttamente salvati anche tutti gli url delle foto presenti nello Storage, in modo da averli disponibili quando dovranno essere mostrate le immagini dei reperti. Il download dei dati da Firestore viene comunque fatto solo una volta, perché poi vengono salvati nella variabile locale *_fossils*. Il metodo fa quindi prima un controllo su questa variabile perché, nel caso in cui i dati siano già stati salvati, i database di Firebase non vengono per niente contattati. In questo modo il recupero dei dati necessari risulta molto più veloce, in quanto si trovano in locale. Oltre al controllo sulla variabile *_fossils*, ne viene effettuato uno sulla variabile *_filteredFossils*. A questa variabile vengono assegnati gli eventuali reperti che risultano dalla ricerca effettuata dall'utente. Nel caso in cui ci siano dei valori, cioè c'è appena stata una ricerca, il metodo ritorna indietro esclusivamente ai reperti trovati. La funzione *getFossils* viene utilizzata nella lista di tutti i reperti.

La mappa (home) fa invece utilizzo del metodo *createMarkers*, che si occupa di creare i Marker nelle coordinate in cui si trovano le zone. Questo metodo fa gli stessi controlli sulla posizione di *getZones*, poi utilizza i dati dei reperti per creare i Marker. La posizione serve anche in questo caso perché la mappa inizialmente è centrata sull'utente. Poi viene verificato se in *_filteredZones* sono presenti i risultati della ricerca, in quanto il filtraggio è applicabile anche alla mappa.

Un altro metodo importante è *backButton*; questo metodo viene eseguito quando l'utente clicca sul **Back Button** di Android nell'**App bar**. In base al contenuto raffigurato, in *backButton* è definita una logica per la quale l'utente viene rimandato al contenuto precedente. Ovviamente se si trova sulla mappa l'applicazione viene chiusa, in quanto la mappa è la schermata iniziale.

4.3.5| PROCESSO DI INVIO DELLE NOTIFICHE

Questo metodo oltre a essere già stato implementato nella prima versione di questa app, ma se ne riporta di seguito il funzionamento per completezza:

Nell'istanza di stato della classe '*MyHomePage'* viene gestito anche il processo di invio delle notifiche. Il metodo *initState* viene eseguito quando lo Widget viene inserito nello Widget Tree, cioè l'albero che contiene tutti i Widget visualizzati in un determinato momento. Qui viene inizializzato il *Workmanager*, ovvero la componente che si occupa di eseguire codice Dart in background. Al *Workmanager* viene assegnato un task periodico, eseguito ogni 30 minuti, che a sua volta richiamerà una funzione di callback. La funzione di callback è definita sempre nel file '*home.dart*', ma fuori della classe MyHomePage e della sua istanza di stato, ed è denominata *callbackDispatcher*.

In *callbackDispatcher* è presente tutto il codice che andrà in esecuzione in background, che si occuperà della gestione delle notifiche. Prima di tutto, essendo fuori dalla classe '*MyHomePage'*, devono essere recuperati i dati dal database Firestore e deve essere calcolata la posizione dell'utente. Successivamente viene inizializzato il plugin delle notifiche. Poi viene fatto un confronto tra la posizione dell'utente e le posizioni di tutte le zone, per vedere quanti di essi si trovano nelle vicinanze. Se effettivamente ci sono zone nei dintorni, viene mandata una notifica con specificato il numero di tali punti di interesse.

4.3.6| LO WIDGET ZONESCREEN

Si tratta di uno Stateful Widget, definita nel file '*fossil.dart'*. In questa classe vengono prelevati i dati che sono passati con la navigazione, ovvero le informazioni sulla zona, in modo da poter capire quali fossili sono associati alla zona selezionata. Questi dati sono passati tramite un'oggetto, cioè un'istanza della classe '*ZoneArguments'*, che si trova nel file '*photo_arguments.dart*'.

Anche nella classe '*ZoneScreen'* è presente la stessa *getFossils*, con l'unica eccezione che, siccome sarà presente la lista di tutti i fossili appartenenti ad una determinata zona già selezionata prima, sarà necessario eseguire una Query per prendere solo i fossili che abbiano l'id (chiamato zona) che combacino con l'id univoco della zona.

Inoltre, la variabile *_search* è presente anche in questa classe, in quanto anche da questa schermata sarà possibile eseguire dei filtri sui fossili.

L'**App bar** in questo caso è costituita esclusivamente da un bottone che permette di tornare indietro, mentre è presenta una Floating Action Bar (**FAB**) che permette di tornare nella Home, dove verrà

mostrato il marker della zona sulla mappa. In entrambi i casi, quindi, viene fatto un pop che rimuove la schermata dallo stack, così come se si dovesse premere sul Back Button di Android.

4.3.7| LO WIDGET FOSSILSCREEN

Si tratta di uno Stateful Widget, definito nel file '*fossilDetail.dart*'. In questa classe vengono prelevati i dati che sono passati con la navigazione, ovvero le informazioni sul reperto, in modo da poterne mostrare le due foto, nome del luogo, descrizione, cava di provenienza e tipo di roccia. Questi dati sono passati tramite un'oggetto, cioè un'istanza della classe '*FossilArguments'*, che si trova nel file '*fossilDetail_arguments.dart*'. L'App bar in questo caso è costituito esclusivamente da un bottone che permette di tornare indietro.

4.3.8| LOGIN, REGISTRAZIONE, LOGOUT

La parte riguardante l'autenticazione è stata di facile implementazione, in quanto Firebase mette a disposizione il plugin *firebase_auth* che rende molto più semplice effettuare il *login*, il *logout* e la *registrazione* semplicemente usando dei metodi che sono già predefiniti nel plugin.

La registrazione è definita nel file '*registration.dart*', e per registrarsi sarà sufficiente inserire Nome, Cognome, indirizzo email (univoco), una password (con minimo 6 caratteri), e la conferma della password, mentre il Login è definito nel file '*login.dart*', e per autenticarsi sarà sufficiente inserire email e password.

Sia che effettuiamo la registrazione, sia che effettuiamo il login, l'utente verrà mandato nella pagina di profilo dell'utente, definito nel file '*profile.dart*', e sarà possibile eseguire tre azioni:

- Effettuare il logout dalla app bar e ritornare nella home come utente non autenticato
- Cliccare sul bottone "Invia segnalazioni!" che aprirà il Widget '*ImageUpload'*
- Cliccare sul bottone "Mostra segnalazioni" che aprirà il Widget '*ShowUploads'*

I dati dell'utente sono passati tramite un'oggetto, cioè un'istanza della classe '*UserModel'*, che si trova nel file '*user_model.dart*'.

4.3.9| LO WIDGET IMAGEUPLOAD

Si tratta di uno Stateful Widget, definita nel file '*image_upload.dart*'. In questa classe sarà possibile inserire una descrizione breve ed inviare una foto per segnalare la presenza di un fossile da parte dell'utente tramite una **FAB** che al clic eseguirà il metodo '*imagePickerMethod*' il quale aprirà la fotocamera del dispositivo, poi in seguito allo scatto e al caricamento dell'immagine sull'applicazione, abbasserà la risoluzione della foto per evitare l'eccessivo appesantimento del database e caricata sull'app.

Inoltre siccome dal primo avvio dell'applicazione quando ci si trova nella homepage, l'app chiederà l'autorizzazione alla localizzazione, al momento dello scatto della foto, l'app eseguirà direttamente

un “geoStamp” dell’utente ricavando così latitudine e longitudine della posizione relativa alla foto scattata, che poi viene “incapsulata” utilizzando il package ‘*geoflutterfire2*’ in un unico Geopoint.

Infine verrà effettuato un timeStamp del momento in cui è stata scattata la foto.

Quindi, una volta presa la descrizione, la foto della segnalazione, la posizione dell’utente in GeoPoint, il timeStamp, e infine l’ID dell’utente che ha effettuato la foto, tramite il metodo ‘*uploadImage*’ verrà caricato tutto su *cloud_firestore* e *firebase_storage*, con un bottone che si trova sull’**App bar** a forma di freccia.

4.3.10| LO WIDGET SHOWUPLOADS

L’ultimo Widget che è stato implementato per il progetto ‘Ammoniti di strada’ è ‘*ShowUploads’*, che è definito nel file ‘*show_uploads.dart*’. In questa classe sarà possibile visualizzare tutte le foto delle segnalazioni effettuate precedentemente dallo stesso utente autenticato, e la data dello scatto (ovvero il timeStamp) in ordine decrescente

4.3.11| NAVIGAZIONE FRA LE SCHERMATE

La navigazione verso altre schermate in Flutter è gestita da una componente chiamata ‘*Navigator’*. Il Navigator mette a disposizione vari metodi per passare da una schermata all’altra. Il suo funzionamento può essere descritto brevemente come uno stack, possono cioè essere fatte *push* (inserimento) e *pop* (rimozione) di schermate; se viene effettuata la *push* la schermata andrà in primo piano, con la pop viene rimossa e viene visualizzata quella che si trova appena sotto nella pila.

Una variante di *push* è *pushNamed*, che consente di sfruttare le rotte definite in ‘*MyApp’* per istanziare più facilmente lo Widget che si desidera mostrare.

Nell’app ‘Ammoniti di Strada’ il Navigator viene utilizzato per passare dalla schermata principale alle altre.

Il metodo *pushNamed* viene utilizzato in particolare per navigare verso la schermata di informazione del singolo reperto e nella schermata della lista dei fossili associati ad una zona (*ZoneScreen*). Quando l’utente preme su una finestra di informazione legata a un marker, su un elemento della lista delle zone nelle vicinanze oppure su una foto in tutti i reperti, tramite il Navigator si passa alla schermata della lista dei fossili associati ad una zona oppure di descrizione del fossile.

4.4| DETTAGLI IMPLEMENTATIVI

In questa sezione saranno analizzate in modo più dettagliato alcune tecniche utilizzate nell'app. Le funzionalità che l'app deve mettere a disposizione sono infatti realizzabili seguendo diverse vie; le scelte fatte sono il risultato di un lavoro progressivo, dettato da molteplici fonti di informazione e correzioni di errori pregressi. Andando avanti con il progetto, infatti, l'obiettivo è stato quello di rendere il codice il più ordinato possibile, mantenendo comunque il massimo dell'efficienza e un'ottimizzazione dei tempi di caricamento, per quanto la complessità di tutto il codice ha subito un incremento, e se l'app andrà avanti con nuove funzionalità sarà destinata ad aumentare.

4.4.1| WIDGETSFLUTTERBINDING, MYAPP E INITSCREEN

```
var initScreen;

//entry point che inizializza i servizi firebase e lancia l'app
Future<void> main() async {

  WidgetsFlutterBinding.ensureInitialized();

  SharedPreferences preferences = await SharedPreferences.getInstance();
  initScreen = await preferences.getInt('initScreen');
  await preferences.setInt('initScreen', 1);


  await Firebase.initializeApp();
  runApp(MyApp());
}
```

Figura 4.12: Il metodo main

Nel metodo main (*Figura 4.12)*, nella prima istruzione viene eseguito il metodo '*WidgetsFlutterBinding.ensureInitialized',* il quale era già presente sin dallo scorso tirocinio.

WidgetFlutterBinding è una componente utilizzata dal Framework Flutter per comunicare con il Flutter Engine. Siccome '*Firebase.initializeApp*' ha bisogno di utilizzare codice nativo per inizializzare i servizi Firebase, passando tramite i Platform Channels, prima si deve verificare che ci sia un'istanza di **WidgetsBinding**. Infatti, i Platform Channels si trovano nell'Engine, per questo risulta necessario avere a disposizione **WidgetsBinding**.

Infine nel metodo main viene chiamata la funzione '*runApp(MyApp())*'.

Inoltre viene istanziata una variabile di tipo *SharedPrecerences*, e poi associata ad una variabile '*initScreen'* che di default è settata a **0**, ma che dopo il primo avvio sarà impostato ad **1**, questo perché al primo avvio, nel metodo build si istanzia un oggetto **MaterialApp** che definisce un'applicazione che utilizza **Material Design**. '*MyApp'* rappresenta cioè l'istanza dell'applicazione, e vengono definite tutte le rotte che il Navigator poi potrà utilizzare, compresa quella iniziale, ovvero '*initialRoute'*, nel caso dell'app 'Ammoniti di strada' quando 'initScreen' è uguale a **0** la rotta iniziale sarà '*/onboarding'* (schermata di tutorial), mentre dal secondo avvio in poi, quando '*initScreen'* sarà uguale a **1**, la rotta iniziale diverrà '*/home'*, come viene mostrato nella *Figura 4.13.*

```
class MyApp extends StatelessWidget {



  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Ammoniti di Strada',
      initialRoute: initScreen == 0 || initScreen == null ? '/onboarding' : '/home',
      routes: {
        '/onboarding': (context) => OnboardingPage(),
        '/home': (context) => const MyHomePage(),
        '/zone': (context) => ZoneScreen(),
        '/info': (context) => FossilScreen(),
        '/login': (context) => Login(),
        '/profile': (context) => Profile(),
        '/segnalation': (context) => ImageUpload(),
      },
      theme: ThemeData(
        primarySwatch: Colors.red,
      ),  // ThemeData
    );  // MaterialApp
  }
}
```

Figura 4.13: Lo Widget MyApp, il suo metodo build e la rotta iniziale

4.4.2| IL METODO BUILD DI ONBOARDINGPAGE

```
@override
Widget build(BuildContext context) => Scaffold(
  body: Container(
padding: const EdgeInsets.only(bottom: 80),
  child: PageView(
    controller: controller,
    onPageChanged: (index){
      setState(() => isLastPage = index == 4);
    },
    children: [
      buildPage(
        color: Colors.red.shade400,
        urlImage: 'assets/images/ic_launcher.png',
        title: 'Benvenuto in Ammoniti di Strada!',
          subtitle: 'cerca nella mappa le zone che più ti interessano per ricercare i fossili'
      ),
```

Figura 4.14: Prima parte dello Widget build di Onboarding

Il metodo build consiste in un insieme di pagine (*Figura 4.14*) [17], la cui struttura viene definita in un altro Widget chiamato **buildPage**, il tutto racchiuso in una **PageView**, che l'utente può scorrere verso sinistra o verso destra fino all'ultima pagina (che è la quinta), ad ogni scorrimento l'indice di pagina avrà un incremento o un decremento a seconda del verso di scorrimento.

```
bottomSheet: isLastPage
? TextButton(
  style: TextButton.styleFrom(
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(300)
    ),  // RoundedRectangleBorder
    primary: Colors.white,
    backgroundColor: Colors.redAccent.shade700,
    minimumSize: const Size.fromHeight(80)
  ),
    onPressed: () async {

    final prefs = await SharedPreferences.getInstance();
    prefs.setBool('showHome', true);

      Navigator.of(context).pushReplacement(
          MaterialPageRoute(builder: (context) => MyHomePage()));
    },
    child: const Text(
      'Inizia!',
      style: TextStyle(fontSize: 24),
    ),  // Text
)  // TextButton
```

Figura 4.15: Seconda parte dello Widget build di Onboarding

Proseguendo nella seconda parte del codice (*Figura 4.15*), una volta che si è giunti nell'ultima pagina apparirà il bottone 'inizia' che ci permetterà di raggiungere la homepage.

Qui è presente un'altra variabile *sharedPreferences*, che a differenza di quella spiegata precedentemente non è una variabile intera, ma booleana e si setta la variabile '*showHome*' da **false** a **true**, e sostanzialmente ha la stessa funzione di '*initScreen*' già definita.

Per sicurezza ho preferito tenere entrambe le variabili, una nella '*OnboardingPage*' e un'altra nel '*MyApp*' (che è quella che agisce per prima).

```
TextButton(
    onPressed: () => controller.jumpToPage(4),
    child: const Text('SALTA')
),  // TextButton
Center(
  child: SmoothPageIndicator(
    controller: controller,
    count: 5,
    effect: WormEffect(
      spacing: 16,
      dotColor: Colors.black26,
      activeDotColor: Colors.redAccent.shade700,
    ),  // WormEffect
    onDotClicked: (index) => controller.animateToPage(
      index,
      duration: const Duration(milliseconds: 500),
      curve: Curves.easeIn,
    ),
  ),  // SmoothPageIndicator
),  // Center
TextButton(
    onPressed: () => controller.nextPage(
      duration: const Duration(milliseconds: 500),
      curve: Curves.easeInOut,
    ),
    child: const Text('CONTINUA')
),  // TextButton
```

Figura 4.16: Terza parte dello Widget build di Onboarding

Troviamo infine una riga con due bottoni, il primo è il tasto 'SALTA' che manderà l'utente direttamente nell'ultima pagina, mentre il secondo è il tasto 'CONTINUA' che ci manderà nella pagina successiva ad ogni click fino all'ultima (*Figura 4.16*).

Nel mezzo troviamo il Widget ***SmoothPageIndicator***, facente parte del package '*smooth_page_indicator*' che ci permette di cambiare pagina con lo swipe destro o sinistro del dito.

4.4.3| IL METODO BUILD DI MYHOMEPAGE

```
@override
Widget build(BuildContext context) {
  bool isFabVisible = true;
  if(FirebaseAuth.instance.currentUser != null) {
    isFabVisible = true;
  }
  else if (FirebaseAuth.instance.currentUser == null){
    isFabVisible = false;

  }


  return WillPopScope (onWillPop: ()  =>backbutton()

    ,child:  Scaffold(
```

Figura 4.17: Widget WillPopScope ritornato dal metodo build

Questa parte di codice era stata già precedentemente implementata sin dalla sua prima versione, con qualche modifica aggiuntiva, ma per completezza ho deciso di inserire anche le informazioni relative a questa parte.

In '*MyHomePage'*, conviene partire descrivendo proprio il metodo **build**, Notiamo subito che la prima istruzione è quella riservata a *isFabVisible*, la cui funzione è stata spiegata prima. Di solito questa funzione, nel caso delle schermate, fa una *return* dello **Scaffold**. Lo Scaffold è lo Widget che rappresenta la struttura di layout di base, ed è composto da altri Widget che definiscono gli elementi grafici di una schermata. In questo caso invece, lo Scaffold è contenuto come *child* dello Widget **WillPopScope** (*Figura 4.17*). Inserendo **WillPopScope**, ogni volta che l'utente preme il **Back Button** del dispositivo, viene chiamato il metodo assegnato al parametro *onWillPop*. Assegnando un metodo personalizzato ad *onWillPop*, è possibile quindi gestire come si vuole l'evento del **Back Button**, ed è quello che è stato fatto nell'app 'Ammoniti di Strada'.

Nello Scaffold sono poi definiti quattro parametri, cioè **App bar**, **body**, **bottomNavigationBar,** e **floatingActionBar**, a cui sono assegnati come valore i Widget che rappresentano quegli elementi grafici. La struttura dello Scaffold è visibile nel codice della *Figura 4.18*.

L'***App Bar*** e il ***Body*** sono costruiti tramite due metodi, ovvero *appBarSetter* e *bodySetter*, i quali, in base al valore di alcune variabili di stato, modificano l'elemento grafico nella schermata.

La ***Bottom bar*** è invece statica. Al suo parametro items è assegnato un Array di *BottomNavigationBarItem*, cioè le icone con testo che costituiscono la ***Bottom bar***. Al parametro *currentIndex* è associata la variabile *_currentIndex*; in questo modo, quando l'utente preme su un'icona, il valore della variabile di stato viene modificato e di conseguenza anche l'interfaccia.

La **Floating Action Bar**, presenta un attributo *visible* con il valore di *isFabVisible*, quindi a seconda che questa variabile sia **true** o **false**, la FAB sarà visibile o invisibile (dipende tutto se l'utente è loggato o no)

Il metodo '*onTabTapped'*, assegnato al parametro *onTap*, si occupa di fare ciò al tocco dell'utente su una delle icone della barra di navigazione.

```
,child:  Scaffold(

appBar: appBarSetter(),
body: bodySetter(),  // end of Align
floatingActionButton: Visibility(
  visible:  isFabVisible,
  child: FloatingActionButton(
      onPressed: () {
          Navigator.of(context).pushReplacement(MaterialPageRoute(
              builder: (context) => ImageUpload(),
          ));  // MaterialPageRoute
      },
      child: Icon(Icons.alarm_on)
  ),  // FloatingActionButton
),  // Visibility
```

```
bottomNavigationBar: BottomNavigationBar(
  items: const <BottomNavigationBarItem>[
    BottomNavigationBarItem(
      icon: Icon(Icons.home_rounded),
      label: 'Home',
    ),  // BottomNavigationBarItem
    BottomNavigationBarItem(
      icon: Icon(Icons.near_me_rounded),
      label: 'Nelle vicinanze',
    ),  // BottomNavigationBarItem
    BottomNavigationBarItem(
      icon: Icon(Icons.photo_library_rounded),
      label: 'Tutti i reperti',
    ),  // BottomNavigationBarItem
    BottomNavigationBarItem(
      icon: Icon(CupertinoIcons.tray_arrow_up_fill),
      label: 'Mostra segnalazioni',
    ),  // BottomNavigationBarItem

  ],  // <BottomNavigationBarItem>[]

  selectedItemColor: Colors.amber,
  unselectedItemColor: Colors.redAccent,
  currentIndex: _currentIndex,
  onTap: onTabTapped,

),  // BottomNavigationBar

);  // Scaffold, WillPopScope
```

Figura 4.18: Codice che definisce la struttura dello Scaffold

4.4.4| TECNICHE UTILIZZATE IN GETZONES, GETFOSSILS E CREATEMARKERS (MYHOMEPAGE)

Dal punto di vista implementativo, è sicuramente importante considerare il codice del metodo *getZones*. In precedenza, è stato detto che questo metodo restituisce i dati delle zone e la posizione dell'utente; viene utilizzato infatti nelle schermate 'Zone nelle vicinanze' e 'Home', che hanno bisogno di queste informazioni per poterle visualizzare. Oltre a ciò, ci sono poi dei dettagli sul codice da approfondire.

Prima di tutto, il tipo di ritorno di *getZones* è **Future**. Future, dove T è un tipo *generics* (un tipo generico, che permette la specificazione in fase di compilazione del codice), è il risultato di un processo asincrono. Un processo asincrono si ha quando il risultato non è fornito immediatamente; solitamente questa situazione si verifica quando il programma deve attendere un qualcosa di esterno, come ad esempio la lettura da un file, da un database, oppure la lettura dati da un servizio remoto.

Future è un tipo *wrapper*, ovvero un involucro che può contenere un qualsiasi altro tipo. In pratica Future serve per avvertire il compilatore che il risultato può non essere disponibile immediatamente, e che il tipo del risultato sarà proprio quello contenuto in Future. Quindi, nel caso di *getZones*, viene ritornato un tipo *Map*.

```
Future<Map> getZones() async {
  var pos;

  LocationPermission permission = await Geolocator.checkPermission();

  if (permission == LocationPermission.denied) {

    if(_currentIndex==2){
      pos = null;
    }
    else{

    permission = await Geolocator.requestPermission();

    if (permission== LocationPermission.denied||permission==LocationPermission.deniedForever) {
      pos ='denied';
    }}
  }

  else  {
  var serviceEnabled = await Geolocator.isLocationServiceEnabled();

  if(!serviceEnabled){
    pos = null;
  }
  else{
   pos = await Geolocator.getCurrentPosition(desiredAccuracy: LocationAccuracy.lowest);}}
```

Figura 4.19: Prima parte del metodo getZones (recupero della posizione)

Dopo il nome della funzione è presente la parola chiave *async*, che viene usata per dichiarare che la funzione ritornerà un Future. Quando invece una funzione asincrona viene chiamata, prima del suo nome viene utilizzata la parola chiave *await*, che serve per mettere l'esecuzione in pausa, in attesa che il risultato sia disponibile.

All'interno della funzione, prima di tutto viene controllato se è stato fornito il permesso di geolocalizzazione. Per fare ciò viene utilizzato il metodo '*checkPermission*' del package *Geolocator*. Il risultato è una costante, e viene salvato nella variabile *permission*, di tipo *LocationPermission*. Tutto ciò viene mostrato nella *Figura 4.19.*

Poi nell'if viene controllato se la variabile ha un valore pari a *LocationPermission.denied*. Nel caso sia così vengono distinti due casi. Se il valore di *_currentIndex* è pari a **2**, ovvero la schermata è quella di tutte le zone, in cui la posizione non è necessaria, allora la variabile *pos* viene valorizzata a **null**. Se il valore è **0** o **1** (home e reperti nelle vicinanze), il permesso per la geolocalizzazione viene richiesto all'utente.

Se anche dopo averlo richiesto l'utente non fornisce il permesso, a *pos* viene dato il valore **denied**, in modo che quando verranno visualizzate la mappa o la lista dei reperti nelle vicinanze, si terrà conto del mancato permesso di geolocalizzazione.

Se invece il permesso è stato fornito, si entra nel ramo else. Qui si controlla se la posizione è attiva, tramite il metodo '*isLocationServiceEnabled*' di Geolocator. Il risultato viene salvato nella variabile *serviceEnabled*. Se il servizio non è attivo in *pos* viene messo il valore **null**, altrimenti si calcola la posizione dell'utente, con il metodo di Geolocator '*getCurrentPosition*'.

E ciò avviene nelle righe di codice della *Figura 4.20:*

```
var list;
Map map = Map();


print(pos);
if (_zones == null) {
  // Get docs from collection reference
  QuerySnapshot querySnapshot = await collectionRefZone.get();

  // Get data from docs and convert map to List
  list = querySnapshot.docs.map((doc) => doc.data()).toList();
  for (var zone in list) {
    final ref = FirebaseStorage.instance.ref().child('foto zona 2').child(
        zone['fotozona']);
    var url = await ref.getDownloadURL();
    zone['url'] = url;
  }
  _zones = list;
  map['zones'] = list;
  map['position'] = pos;
  return map;
}
else {
  list = _zones;
  map['zones'] = list;
  map['position'] = pos;
  return map;
}
```

Figura 4.20: Seconda parte del codice di getZones (recupero dei dati)

Se la variabile *_zones* è **null** allora viene stabilita la connessione con il database. Prima di tutto si va a salvare in *querySnapshot* la collection '*zone*'.

Nella classe *_MyHomePageState*, infatti, sono definite due variabili. *FirebaseFirestore firestore* rappresenta l'istanza del database Firestore, mentre *collectionRefZone* è la collection '*zone*'. In *querySnapshot* viene messo il risultato del metodo *get* applicato a *collectionRefZone*, cioè un'istantanea della collection in quel momento.

Tutti i documenti presenti in '*zone*' vengono convertiti in mappe e messi in una lista denominata *list*. I documenti sono infatti costituiti da coppie chiave-valore; quindi, la traduzione in mappe risulta adatta.

Nel ciclo for si vanno a considerare i documenti uno ad uno. Il valore del campo '*fotozona*' nei documenti contiene il nome della foto presente nello Storage che corrisponde a quel reperto. In *ref* viene salvato l'url di quella foto, che viene poi aggiunto alle coppie chiave-valore del reperto, sotto la chiave '*url*'. Questo viene fatto per ogni zona. In seguito, i dati dei luoghi vengono salvati nella variabile *_zones*, in modo che, quando questo metodo sarà richiamato, non verrà utilizzato il database ma saranno recuperati direttamente i dati in locale.

Se invece in *_zones* i dati erano già presenti, vengono messi nella variabile *list*.

Alla fine, il metodo ritorna una mappa (coppie chiave-valore) contenente i dati delle zone (come valore della chiave '*zones*') e l'eventuale posizione (come valore della chiave '*position*').

Il metodo *getFossils* è molto simile (*Figura 4.21*), ma con la differenza che invece delle zone, si ricava una lista di tutti i fossili contenuti nell'apposita collection '*fossili*', rappresentata da *collectionRefFossil*, e verrà richiamato solo per *_currentIndex* uguale a **2**, e non essendo i fossili georeferenziati, non c'è alcun bisogno di tutta la prima parte relativa alla localizzazione.

```
Future<Map> getFossils() async {

  var listFossil;
  Map map = Map();


  if (_fossils == null) {
    // Get docs from collection reference
    QuerySnapshot querySnapshot = await collectionRefFossil.get();

    // Get data from docs and convert map to List
    listFossil = querySnapshot.docs.map((doc) => doc.data()).toList();


    for (var fossil in listFossil) {
      final ref = FirebaseStorage.instance.ref().child('foto dettaglio').child(
          fossil['fotofossile']);
      var urlfossil = await ref.getDownloadURL();
      fossil['urlfossil'] = urlfossil;
    }

    _fossils = listFossil;
    map['fossils'] = listFossil;
    return map;
  }
  else {
    listFossil = _fossils;
    map['fossils'] = listFossil;
    return map;
  }
```

Figura 4.21: Codice di getFossils (recupero dei dati)

Il metodo *createMarkers* è molto simile; l'unica differenza è che, essendo utilizzato nella home (con la mappa), al posto dei dati sulle zone ritorna direttamente i Marker che devono essere visualizzati (oltre alla posizione dell'utente), come mostrato nella *Figura 4.22:*

.

```
if(_filteredZones.isNotEmpty){
  zones=_filteredZones;

}
else  if (_filteredZones.isEmpty && _zones == null)
{var map = await getZones();
zones = map['zones'];
pos = map['position'];
}
else if (_filteredZones.isEmpty && _zones != null) {
  zones = _zones;

}

Set<Marker> markers = Set();
Map map = Map();
for (var zone in zones){
  var url = zone['url'];
  markers.add(Marker(
      markerId: MarkerId(zone['nomezona']),
      position: LatLng(zone['lat'],zone['long']),
```

Figura 4.22: Parte del codice di createMarkers (recupero dei dati)

Dopodiché, come viene mostrato nella *Figura 4.23,* al click dell'utente su uno dei marker, si aprirà una *InfoWindow* che mostrerà l'immagine e il nome della zona, e cliccandoci sopra finiremo nella classe '*ZoneScreen*' dove sarà presente la lista dei fossili associata alla zona selezionata.

```
onTap: (){
  _customInfoWindowController.addInfoWindow!(
    Column(
      children: [

        Expanded(
          child: Container(
            decoration: BoxDecoration(
              color: Colors.white,
              borderRadius: BorderRadius.circular(10),
            ),  // BoxDecoration
            child: Padding(
              padding: const EdgeInsets.all(8.0),
              child: Row(
                mainAxisAlignment: MainAxisAlignment.center,
                children: [
                  GestureDetector(onTap: () {
                    Navigator.pushNamed(context, '/zone', arguments: ZoneArguments(zone['url'], zone['nomezona'], zone['descr'], zone['lat'
                  },child: Container(decoration: BoxDecoration(borderRadius: BorderRadius.circular(10)),width:50, height:50, child:Image.ne
                  const SizedBox(
                    width: 18.0,
                  ),  // SizedBox
                  Flexible(child:
                    GestureDetector(onTap: () {
                      Navigator.pushNamed(context, '/zone', arguments: ZoneArguments(zone['url'], zone['nomezona'], zone['descr'], zone['lat'
                    },child:Text(
                      zone['nomezona'],
```

Figura 4.23: Parte del codice di createMarkers (gestione della InfoWindow)

4.4.5| COSTRUZIONE DELLA LISTA DELLE ZONE NELLE VICINANZE

Come detto in precedenza, *bodySetter* si occupa di costruire il **Body**, ovvero il contenuto centrale, a seconda del valore della variabile di stato *_currentIndex*.

Nel caso in cui *_currentIndex* sia pari a **1**, quindi nel caso in cui l'utente abbia appena premuto sull'icona 'Zone nelle vicinanze', deve essere disegnata sullo schermo la lista dei reperti nelle vicinanze, il che avviene nel modo seguente (*Figura 4.24*).

```
//lista con i reperti nelle vicinanze
if(_currentIndex==1)  {
  return FutureBuilder(future: getZones(),builder: (BuildContext context, AsyncSnapshot<Map> snapshot) {
    switch (snapshot.connectionState) {
      case ConnectionState.waiting:
        return Container(
          height: MediaQuery.of(context).size.height,
          width: MediaQuery.of(context).size.width,
          color: Colors.transparent,
          child: const Center(
            child: SizedBox(
              height: 60.0,
              width: 60.0,
              child:
              //Image.asset('assets/images/loader.gif',fit: BoxFit.fill,) // use you custom loader or default loader
              CircularProgressIndicator(
                valueColor: AlwaysStoppedAnimation(
                    Colors.redAccent),  // AlwaysStoppedAnimation
                strokeWidth: 5.0))));  // CircularProgressIndicator, SizedBox, Center, Container
      default:
        if (snapshot.hasError) {
          return Text('Error: ${snapshot.error}');
        }
```

Figura 4.24: Codice con controllo della disponibilità dei dati e verifica errori

In *bodySetter* ci sono degli if in successione. Se *_currentIndex* è **1**, si entra nel rampo che contiene il codice di realizzazione della lista. Da questo blocco di codice viene ritornato uno Widget **FutureBuilder**, che andrà a valorizzare il **Body**. FutureBuilder è uno Widget che costruisce sé stesso basandosi sull'ultima interazione con un oggetto di tipo Future. Il processo asincrono è passato come valore del parametro future. In questo caso il Future è il risultato dell'esecuzione del metodo

getZones. In questo modo i dati sulle zone, che sono il risultato di una chiamata asincrona, sono gestiti dal FutureBuilder, il quale li utilizzerà non appena risulteranno disponibili.

L'altro parametro di FutureBuilder è **builder**, che rappresenta la strategia di costruzione dello Widget, ed è valorizzato con il metodo che si occupa di costruire lo Widget. A questo metodo è passato un oggetto del tipo *AsyncSnapshot*, che rappresenta una 'fotografia' istantanea del dato proveniente dalla computazione asincrona. In base allo stato in cui questo oggetto si trova, il builder ritorna degli oggetti diversi da inserire nel **Body**.

Viene fatto infatti uno switch della proprietà *connectionState dell'AsyncSnapshot*, cioè lo stato della computazione asincrona. Lo switch analizza i possibili valori di *connectionState* e in base a ciò viene eseguita una porzione di codice o l'altra.

Se il valore è *ConnectionState.waiting*, cioè i dati non sono ancora pronti, viene ritornato un *CircularProgressIndicator*, ovvero una rotella di caricamento. In questo modo l'utente capisce che c'è un caricamento in corso. Se invece i dati non sono pronti, ma neanche stanno caricando, si entra nel ramo default; qui si valuta se c'è stato un errore di caricamento, e nel caso viene ritornato un *Text* (testo) con il messaggio di errore.

Qualora il risultato sia disponibile, si procede con la creazione della lista. Innanzitutto, viene dichiarata una variabile *zones*, che conterrà tutti i fossili da inserire nella lista. Poi in un if si valuta se ci sono fossili risultato di una ricerca in *_filteredZones*, e nel caso ci siano si prelevano quelli; altrimenti in *zones* vengono inseriti i luoghi che provengono dalla chiamata asincrona, cioè da *getZones*.

```
else {
  var zones;
  if (_filteredZones.isNotEmpty)
  {
    zones = _filteredZones;
  }
  else{zones = snapshot.data!['zones'];}
  var pos = snapshot.data!['position'];
  if(pos==null) {
    return Center(child: Text('Attiva la posizione per vedere i reperti nelle vicinanze!'));
  }
  if (pos=='denied'){
    return Center(child: Text('Devi fornire i permessi per la localizzazione!'));
  }
```

Figura 4.25: Codice per mostrare messaggi di errore sulla geolocalizzazione

Poi viene considerata la variabile position passata, che viene messa in *pos*. Se il suo valore è **null**, allora vuol dire che i permessi sono forniti ma non è attiva la posizione, quindi viene costruito un Text con scritto di attivare la posizione. Se invece il valore è **denied**, significa che non sono stati forniti i permessi di geolocalizzazione; quindi, viene sempre creato un *Text*, in cui però viene scritto di fornire i permessi. Tutto ciò avviene nella porzione di codice della *Figura 4.25.*

```
List filteredZones = [];
for (var zone in zones!) {
  var distance = Geolocator.distanceBetween(pos.latitude, pos.longitude, zone['lat'], zone['long']);
  if (distance<30*1000)
  {
    zone['distance']=distance;
    filteredZones.add(zone);
  }
}
if (filteredZones.isEmpty)
{
  return Center(child: Text('Non ci sono luoghi nelle vicinanze',style: TextStyle(fontSize: 15, )));
}
else {
  filteredZones.sort((a, b) => a['distance'].compareTo(b['distance']));
```

Figura 4.26: Codice con filtraggio zone in base alla posizione e costruzione lista

Se la posizione è disponibile, viene dichiarata una variabile *near*. Poi viene effettuato un ciclo for in cui vengono confrontate latitudine e longitudine di ogni luogo con quelle dell'utente, per capire la distanza. La distanza è calcolata con il metodo '*distanceBetween*' di *Geolocator*. Nel caso in cui questa distanza sia minore di 1km, il reperto viene aggiunto a *near*, che alla fine dei confronti conterrà tutte le zone che si trovano nelle vicinanze. Qualora capiti che *near* al termine dei confronti sia vuota, viene mostrato a schermo un *Text* con la scritta 'Non ci sono luoghi nelle vicinanze'; altrimenti, la lista dei luoghi viene ordinata in ordine crescente di distanza, utilizzando il metodo '*sort*' (*Figura 4.26*). La costruzione della lista da rappresentare sulla schermata viene poi fatta con il metodo '*Listview.builder*'.

4.4.6| IL METODO DI RICERCA PER LE ZONE E PER I FOSSILI (MYHOMEPAGE)

All'interno dell'applicazione, più precisamente nella '***MyHomePage***' sarà possibile effettuare una funzione di ricerca per zone e fossili, e in base a quale bottone della **BottomNavigationBar**, l'utente ha selezionato (ovvero in base al valore di ***_currentIndex*** che può essere **0**, **1** o **2*).***

Relativamente alla ricerca delle zone e dei fossili, all'interno dell'**App bar** è presente un'icona a forma di lente che implementa una **funzione di ricerca**, la cui parte iniziale del codice è presente nell'*appBarSetter*.

Se *_currentIndex* è uguale a **2**, cioè ci troviamo nella schermata di 'Tutti i reperti', si esegue un for che prende tutti i possibili valori delle cave di provenienza e dei tipi di roccia da Firebase aggiungendoli rispettivamente alle variabili '*cave*' e '*rocce*' per aggiungere le opzioni di inserimento nella form relative ad esse, come viene mostrato nella *Figura 4.27*.

```
textController.text='';
cave = [];
rocce = [];

if(_currentIndex == 2) {
  for (var fossil in _fossils) {
    if (!cave.contains(fossil['provenienza']))
      cave.add(fossil['provenienza']);
    if (!rocce.contains(fossil['roccia']))
      rocce.add(fossil['roccia']);
  }
}
```

Figura 4.27: Parte di codice della funzione di ricerca nell'appBarSetter

All'interno del *bodySetter* abbiamo il continuo della **funzione di ricerca** in cui si verifica se *_search* è uguale a **true**, se lo è, con un altro if si va a verificare se *_currentIndex* è uguale a **0** o **1**, ovvero se si trova nella schermata di 'Home', di 'Zone nelle vicinanze', se anche questa condizione è vera, si aprirà una form che ci permetterà di filtrare le zone per descrizione e il nome della zona, come viene mostrato nella *Figura 4.28*.

```
//metodo che setta il body dell'app, in base al valore di variabili booleana verranno visualizzate le varie schermate
Widget bodySetter() {

  //pagina di ricerca
  if(_search==true){
    if (_currentIndex == 0 || _currentIndex == 1) {
    return  Wrap(alignment: WrapAlignment.center,children: [Padding(padding: EdgeInsets.all(20),child:
    TextField(controller: textController,

      decoration: InputDecoration(

        //label: Text('Inserisci qui una o più parole da ricercare'),
            labelText: 'Inserisci qui una o più parole da ricercare',
        enabledBorder: OutlineInputBorder(
          borderRadius: BorderRadius.circular(10),
          // width: 0.0 produces a thin "hairline" border
          borderSide: BorderSide(color: Colors.white, width: 3.0),
        ),  // OutlineInputBorder

      ),)),  // InputDecoration, TextField, Padding
```

Figura 4.28: Prima parte di codice della funzione di ricerca per zone nel bodySetter

Poi si istanzia *_filteredZones*, e si esegue un ciclo for dove, per ogni zona presente nel database viene eseguito un controllo per verificare se le parole scritte nella form, coincidono con i valori degli attributi del nome della zona e della descrizione mediante un '*textController*'.

Se il risultato è positivo, la schermata in seguito alla schermata si popolerà solo delle zone corrispondenti al filtro, viceversa se la ricerca non ha fornito risultati allora non verrà eseguito nessun filtraggio (*Figura 4.29*).

```
Padding(padding: EdgeInsets.only(bottom: 20), child:  ElevatedButton(style: ButtonStyle(),onPressed: () {
  _filteredZones=[];
  for (var zone in _zones){

    if (((zone['nomezona'].contains(textController.text))||(zone['descr'].contains(textController.text)))) {
      _filteredZones.add(zone);
    }
  }
  if (!(textController.text.isEmpty))
  {
    if (_filteredZones.isEmpty){showDialog<void>(
      context: context,
      barrierDismissible: false, // user must tap button!
      builder: (BuildContext context) {
        return AlertDialog(
          title: const Text('Nessun Risultato'),
          content: SingleChildScrollView(
            child: ListBody(
              children: const <Widget>[
                Text('La ricerca non ha fornito risultati'),
                Text('Prova a modificare i parametri di ricerca'),
              ],  // <Widget>[]
            ),  // ListBody
          ),  // SingleChildScrollView
          actions: <Widget>[
            TextButton(
              child: const Text('Ok'),
              onPressed: () {
                Navigator.of(context).pop();
```

Figura 4.29: Seconda parte di codice della funzione di ricerca per zone nel bodySetter (verificazione dei dati)

La situazione cambia, invece, se *_currentIndex* è uguale a **2**, allora l'utente si troverebbe nella schermata di 'Tutti i reperti', e mediante un else if, cliccando sempre sulla lente di ingrandimento, si aprirà un ulteriore schermata di ricerca, che in questo caso non filtrerà per zone, ma bensì per reperti per descrizione del fossile, cava di provenienza e tipo di roccia (*Figura 4.30*).

```
else if (_currentIndex == 2) {
  return Wrap(alignment: WrapAlignment.center,children: [Padding(padding: EdgeInsets.all(20), child:
  TextField(controller: textController,

    decoration: InputDecoration(

      //label: Text('Inserisci qui una o più parole da ricercare'),
      labelText: 'Inserisci qui una o più parole da ricercare',
      enabledBorder: OutlineInputBorder(
        borderRadius: BorderRadius.circular(10),
        // width: 0.0 produces a thin "hairline" border
        borderSide: BorderSide(color: Colors.white, width: 3.0),
      ),  // OutlineInputBorder

    ),)),  // InputDecoration, TextField, Padding
```

Figura 4.30: Prima parte di codice della funzione di ricerca per fossili nel bodySetter

In questo caso la parte logica rimane comunque simile al caso precedente, le uniche cose che cambiano sono l'aggiunta del filtraggio per cava e roccia, e questo è visibile nelle figure *Figura 4.31* e *Figura 4.32*.

```
Padding(padding: EdgeInsets.only(bottom: 20, left: 20, right: 50),
  child: DropdownButtonFormField(items: cave
      .map<DropdownMenuItem<String>>((var value) {

    return DropdownMenuItem<String>(
      value: value?.toString(),
      child: Text(value.toString()),
    );  // DropdownMenuItem
  }).toList(),
      onChanged: (String? newValue) {

        cavaDropDown = newValue!;

      },
      hint: const Text('Cava di provenienza'),decoration: InputDecoration
        (enabledBorder: OutlineInputBorder(borderRadius: BorderRadius.circular(10),
        // width: 0.0 produces a thin "hairline" border
        borderSide: BorderSide(color: Colors.white, width: 3.0),
      )))),  // OutlineInputBorder, InputDecoration, DropdownButtonFormField, Padding
```

Figura 4.31: Seconda parte di codice della funzione di ricerca per fossili nel bodySetter (form di cave di provenienza)

```
Padding(padding: EdgeInsets.only(bottom: 20, left: 20, right: 50),child: DropdownButton
    .map<DropdownMenuItem<String>>((var value) {
  return DropdownMenuItem<String>(
    value: value?.toString(),
    child: Text(value.toString()),
  );  // DropdownMenuItem
}).toList(),
    onChanged: (String? newValue) {

      rocciaDropDown = newValue!;

    },
    hint: const Text('Tipologia di roccia'),decoration: InputDecoration
      (enabledBorder: OutlineInputBorder(
      borderRadius: BorderRadius.circular(10),
      // width: 0.0 produces a thin "hairline" border
      borderSide: BorderSide(color: Colors.white, width: 3.0),
    )))),  // OutlineInputBorder, InputDecoration, DropdownButtonFormField, Padding
```

Figura 4.32: Terza parte di codice della funzione di ricerca per fossili nel bodySetter (form di tipi di rocce)

Infine poi si istanzia *_filteredFossils*, e si esegue un ciclo for dove, per ogni fossile presente nel database viene eseguito un controllo per verificare se le parole scritte nella form, coincidono con i valori degli attributi della descrizione del fossile, del tipo di roccia e della cava di provenienza, mediante un '*textController*' e due variabili '*cavaDropDown*' e '*rocciaDropDown*'.

Se il risultato è positivo, la schermata in seguito alla schermata si popolerà solo dei fossili corrispondenti al filtro, viceversa se la ricerca non ha fornito risultati allora non viene eseguito nessun filtraggio, come per la funzione precedente (*Figura 4.33*).

```
Padding(padding: EdgeInsets.only(bottom: 20),
  child: ElevatedButton(style: ButtonStyle(), onPressed: () {
    _filteredFossils = [];
    for (var fossil in _fossils) {

      if (((fossil['descrfossile'].contains(textController.text)))&&((fossil['provenienza'] == cavaDropDown)||(cavaDropDown == ''))&&((fo

        _filteredFossils.add(fossil);
      }
    }


    if (!(cavaDropDown == '' && rocciaDropDown == ''&&textController.text.isEmpty))
    {
      if (_filteredFossils.isEmpty) {showDialog<void>(
        context: context,
        barrierDismissible: false, // user must tap button!
        builder: (BuildContext context) {
          return AlertDialog(
            title: const Text('Nessun Risultato'),
            content: SingleChildScrollView(
              child: ListBody(
                children: const <Widget>[
                  Text('La ricerca non ha fornito risultati'),
                  Text('Prova a modificare i parametri di ricerca'),
                ],  // <Widget>[]
```

Figura 4.33: Quarta parte di codice della funzione di ricerca per fossili nel bodySetter

4.4.7| GESTIONE DEL BACK BUTTON CON WILLPOPSCOPE

Come detto in precedenza, lo **Scaffold** di '*MyHomePage*' si trova dentro lo Widget **WillPopScope**; in questo modo, assegnando un metodo al parametro *onWillPop*, è possibile gestire l'evento del touch dell'utente sul **Back Button** del dispositivo. Il metodo realizzato per fare ciò è *backButton*.

Innanzitutto, la funzione *backButton* ritorna un **Future**, in quanto il parametro *onWillPop* richiede una funzione che abbia un tipo di ritorno del genere. Se il valore di ritorno è **false**, allora la schermata viene rimossa dallo Stack, quindi viene chiusa, altrimenti no. Questo metodo risulta necessario nel caso di '*MyHomePage*'.

Le schermate 'Zone nelle vicinanze', 'Home', 'Tutti i reperti' e 'Ricerca', fanno in realtà tutte parte della stessa schermata. È '*MyHomePage*' che in base al valore delle variabili di stato modifica il contenuto, ma non vengono aperte nuove schermate. Ci accade nel caso in cui si vada nella pagina di profilo, login, registrazione, invia segnalazioni, mostra segnalazioni, della '*ZoneScreen*' o del singolo reperto. '*MyHomePage*' inoltre è la prima schermata inserita nello Stack; quindi, se l'evento del **Back Button** non fosse gestito, al tocco del bottone l'app verrebbe chiusa.

La funzione in un primo if controlla se sia visualizzata la **pagina di ricerca**. Nel caso sia così, *_search* viene messo a **false**; quando cioè l'utente preme il Back Button, viene visualizzata la pagina in cui è stata premuta l'icona per la ricerca. Viene ritornato **false** perché non si esce dall'app ma si ritorna semplicemente indietro.

Successivamente si controlla se *_currentIndex* sia **0**, ovvero se quando è stato premuto il **Back Button** era visualizzata la home con la mappa. In questo caso si deve capire se è visualizzata la home normalmente o con solo i Marker risultanti da una ricerca. Nel primo caso viene ritornato il valore **true**, ovvero la schermata viene chiusa e quindi si esce dall'app; nel secondo caso, invece, *_filteredZones* viene svuotata e quindi si torna indietro alla home con tutte le zone (il valore di ritorno è **false**), come viene mostrato nella *Figura 4.34*.

```
Future<bool> backbutton () async {
  if(_search==true){
    setState(() {
      _search = false;
    });
    return false;
  }


  if (_currentIndex == 0) {

    if (_filteredZones.isEmpty) {
      return true; //parte il setState
    }
    else {
      setState(() {
        _filteredZones = [];
        _currentIndex = _currentIndex;
      });
      return false;
    }
  }
```

Figura 4.34: Prima parte di codice di OnWillPopScope

Qualora *_currentIndex* valga **1** o **2** si entra nel ramo else. Qui viene fatto lo stesso controllo dell'if precedente. Se ci sono reperti filtrati dalla ricerca, viene fatto un ripristino, e si torna indietro a 'Reperti nelle vicinanze' o 'Elenco di tutti i reperti'. Altrimenti *_currentIndex* viene messo a **0**, ovvero il tocco del **Back Button** ha come effetto quello di riportare l'utente alla home (*Figura 4.35*).

```
else {

    if (_filteredZones.isNotEmpty) {
      setState(() {
        _currentIndex = _currentIndex;
        _filteredZones=[];
      });
      return false;
    } else {
      setState(() {
        _currentIndex = 0;
      });
      return false;
    }

  }
}
```

Figura 4.35: Seconda parte di codice di OnWillPopScope

4.4.8| IL METODO BUILD DI ZONESCREEN

'ZoneScreen' è la classe riservata alla lista dei fossili associati ad una determinata zona selezionata dalla Homepage o dalla lista delle zone nelle vicinanze.

```
@override
Widget build(BuildContext context) {

  final args = ModalRoute.of(context)!.settings.arguments as ZoneArguments;

  Map result = {'url':args.url,'nomezona': args.nomezona,
    'descr':args.descr,'lat': args.lat,'long' :args.long, 'id' : args.id};

  return Scaffold(

    appBar: appBarSetter(args),
    body: bodySetter(args,result),
    floatingActionButton: FloatingActionButton(
      onPressed: () {
        Navigator.pop(context,result);
      },
      child: Icon(Icons.map)
    ),// end of Align  // FloatingActionButton
  );  // Scaffold

}
```

Figura 4.36: Build di ZoneScreen

Come prima cosa viene istanziata una variabile *args* che tramite '*ModalRoute.of(context)!.settings.arguments as ZoneArguments*' prende dalla rotta della classe precedente, ovvero '*MyHomePage'*, le informazioni relative alla zona selezionata, dopodiché istanziamo *result* di tipo *Map*, che ci permette di ricavare le singole informazioni della zona nel database, ovvero: l'url immagine, il nome della zona, la descrizione, la latitudine, la longitudine e l'id.

Nello **Scaffold** sono poi definiti tre Widget, cioè **appbar**, **body**, e floatingActionButton (**FAB**), a cui sono assegnati come valore i Widget che rappresentano quegli elementi grafici. La struttura dello **Scaffold** è visibile nel codice della *Figura 4.36*. L'**App Bar** e il **Body** sono costruiti tramite due metodi, ovvero *appBarSetter* e *bodySetter*, i quali, in base al valore di alcune variabili di stato, modificano l'elemento grafico nella schermata.

Invece per quanto riguarda la FAB abbiamo un '*Navigation.pop (context, result)*' che ci permetterà di tornare alla mappa con il focus sulla zona di interesse precedentemente selezionata, in quanto result punta alle informazioni relative alla zona.

4.4.9| TECNICHE UTILIZZATE IN GETFOSSILS (ZONESCREEN)

All'interno della classe '*ZoneScreen'* si trova un altro metodo *getFossils*, che sostanzialmente esegue la stessa funzione del *getFossils* all'interno della classe '*MyHomePage'*, tuttavia all'interno di *getFossils*, avremo le informazioni relative alla zona associata a tutti i reperti della lista tramite la variabile *args* che è stata precedentemente spiegata, dopodiché si va a salvare in querySnapshot la collection '*dettaglio'*, ovvero le informazioni relative ai reperti (*Figura 4.37*).

```
//metodo che recupera tutti i dati riguardanti i reperti dal Firestore database, oltre alla posizione attuale dell'utente
Future<Map> getFossils(args) async {

  final args = ModalRoute.of(context)!.settings.arguments as ZoneArguments;

  CollectionReference collectionRef = FirebaseFirestore.instance.collection('dettaglio');
  var list;
  Map map = Map();

  if (_fossilsDetail == null) {
    // Get docs from collection reference
    QuerySnapshot querySnapshot = await collectionRef.where('zona', isEqualTo: args.id ).get();

    // Get data from docs and convert map to List
    list = querySnapshot.docs.map((doc) => doc.data()).toList();
    for (var fossilDetail in list) {
      final ref = FirebaseStorage.instance.ref().child('foto dettaglio').child(fossilDetail['fotofossile']);
      var urlfossil = await ref.getDownloadURL();
      fossilDetail['urlfossil'] = urlfossil;
    }
    _fossilsDetail = list;
    map['fossilsDetail'] = list;
    return map;
  }
  else {
    list = _fossilsDetail;
    map['fossilsDetail'] = list;
    return map;
  }
```

Figura 4.37: Codice di getFossils

Nella classe *_ZoneScreenState*, infatti, sono definite due variabili. *FirebaseFirestore firestore rappresenta* l'istanza del database Firestore, mentre *collectionRef* è la collection '*dettaglio'* dove l'attributo '*zona'* (ovvero l'id della zona associato al fossile) sia uguale all'id della zona precedentemente selezionata, ovvero *args.id*. In *querySnapshot* viene messo il risultato del metodo *get* applicato a *collectionRef*, cioè un'istantanea della collection in quel momento.

Tutti i documenti presenti in '*dettaglio'* vengono convertiti in mappe e messi in una lista denominata *list*. I documenti sono infatti costituiti da coppie chiave-valore; quindi, la traduzione in mappe risulta adatta.

Nel ciclo for si vanno a considerare i documenti uno ad uno. Il valore del campo '*foto dettaglio'* nei documenti contiene il nome della foto presente nello Storage che corrisponde a quel reperto. In *ref* viene salvato l'url di quella foto, che viene poi aggiunto alle coppie chiave-valore del reperto, sotto la chiave '*urlfossil'*. Questo viene fatto per ogni reperto. In seguito, i dati dei fossili vengono salvati nella variabile *_fossils*, in modo che, quando questo metodo sarà richiamato, non verrà utilizzato il database ma saranno recuperati direttamente i dati in locale.

Se invece in *_fossils* i dati erano già presenti, vengono messi nella variabile list.

Alla fine, il metodo ritorna una mappa (coppie chiave-valore) contenente i dati dei reperti (come valore della chiave '*fossils'*).

4.4.10| COSTRUZIONE DELLA LISTA DEI FOSSILI ASSOCIATI AD UNA ZONA

Come detto in precedenza il *bodySetter* si occupa di costruire la parte centrale dell'applicazione, e proprio in questo punto avviene la costruzione della lista dei fossili associati ad una zona:

```
return FutureBuilder(future: getFossils(args),
    builder: (BuildContext context, AsyncSnapshot<Map> snapshot) {
      switch (snapshot.connectionState) {
        case ConnectionState.waiting:
          return Container(
            height: MediaQuery.of(context).size.height,
            width: MediaQuery.of(context).size.width,
            color: Colors.transparent,
            child: const Center(
              child: SizedBox(
                height: 60.0,
                width: 60.0,
                child: CircularProgressIndicator(
                    valueColor: AlwaysStoppedAnimation(
                        Colors.redAccent),  // AlwaysStoppedAnimation
                    strokeWidth: 5.0))));  // CircularProgressIndicator, SizedBox, Center, Container
        default:
          if (snapshot.hasError) {
            return Text('Error: ${snapshot.error}');
          }
```

Figura 4.38: Prima parte di codice relativa alla costruzione della lista dei fossili associata ad una zona

Viene ritornato uno Widget **FutureBuilder**, che andrà a valorizzare il **Body**. In questo caso il Future è il risultato dell'esecuzione del metodo *getFossils*. In questo modo i dati sui reperti, che sono il risultato di una chiamata asincrona, sono gestiti dal **FutureBuilder**, il quale li utilizzerà non appena risulteranno disponibili.

L'altro parametro di FutureBuilder è **builder**, che ha lo stesso funzionamento del builder nella '*MyHomePage*'.

Qualora il risultato sia disponibile, si procede con la creazione della lista. Innanzitutto, viene dichiarata una variabile *fossilsDetail*, che conterrà tutti i fossili da inserire nella lista. Poi in un if si valuta se ci sono fossili risultato di una ricerca in *_filteredFossilsDetail* (*Figura 4.38*), e nel caso ci siano si prelevano quelli; altrimenti in *fossilsDetail* vengono inseriti i reperti che provengono dalla chiamata asincrona, cioè da *getFossils* (*Figura 4.39*).

```
else {
  var fossilsDetail;
  if (_filteredFossilsDetail.isNotEmpty) {
    fossilsDetail = _filteredFossilsDetail;
  }
  else {
    fossilsDetail = snapshot.data!['fossilsDetail'];
  }
```

Figura 4.39: Seconda parte di codice relativa alla costruzione della lista dei fossili associata ad una zona

Come nel caso di '*MyHomePage*' la funzione di ricerca per fossili è analoga, sia per quanto riguarda l'*appBarSetter* che per quanto riguarda il *bodySetter*.

4.4.11| IL METODO BUILD DI FOSSILSCREEN

La classe FossilScreen è stata di facile implementazione, l'unica nota di interesse sta nell'utilizzo del package *image_slider* che permette all'utente si scorrere le foto relative al fossile, che per ora sono massimo **2**, anche se più avanti ne potranno essere messe anche di più.

Per prima cosa, per poter utilizzare questo package dobbiamo congiungere alla classe esistente un'altra mediante il '*with*' per poter utilizzare lo sliding delle foto come animazione, Questa classe si chiama *TickerProviderStateMixin*.

Dopodiché, come si evince dalla *Figura 4.40*, sono presenti 3 controller: due per lo zoom delle foto (*controller1* e *controller2*), e uno per lo scrolling delle foto, chiamato *_tabController*, poi nel metodo '*initState*' viene definito che il Tab Controller deve essere di lunghezza **2** e con riferimento alle due immagini dei fossili.

```
TransformationController controller1 = TransformationController();
TransformationController controller2 = TransformationController();

late TabController _tabController;

@override
void initState() {
  super.initState();
  _tabController =  new TabController(vsync: this, length: 2);


}

@override
void dispose() {
  _tabController.dispose();
  super.dispose();
}
```

Figura 4.40: Inizializzazione e definizione del _tabController

Infine, all'interno del Build viene definito l'**image slider** (*Figura 4.41*) dove vengono definiti i vari attributi dell'ImageSilder e viene definito *_tabController* come **controller** dell'**ImageSlider**, e all'interno di esso sono presenti due Widget **Flexible** che contengono le immagini dei fossili, e tramite il Widget **InteractiveViewer** sarà possibile zoomare tali foto.

:

```
ImageSlider(
    showTabIndicator: true,
    tabIndicatorColor: Colors.redAccent,
    tabIndicatorSelectedColor: Colors.amber,
    tabIndicatorHeight: 16,
    tabIndicatorSize: 16,
    curve: Curves.fastOutSlowIn,
    width: MediaQuery.of(context).size.width,
    height: 220,
    autoSlide: false,
    duration: new Duration(seconds: 6),
    allowManualSlide: true,
    tabController: _tabController,
    children: [
      Flexible(
        fit: FlexFit.loose,
        child: InteractiveViewer(minScale: 1.0,
          transformationController: controller1,
          onInteractionStart: (scale) {
            setState(() => _visible = false);
          },
          onInteractionEnd: (scale) {
            setState(() {
              controller1.value = Matrix4.identity();
              _visible = true;
            });
          },
          maxScale: 3.0,
          clipBehavior: Clip.none,
```

```
          child: Image.network(
            argsFossil.urlfossil,
            fit: BoxFit.cover,

          ),  // Image.network
        )  // InteractiveViewer
      ),  // Flexible
      Flexible(
        fit: FlexFit.loose,
        child: InteractiveViewer(minScale: 1.0,
          transformationController: controller1,
          onInteractionStart: (scale) {
            setState(() => _visible = false);
          },
          onInteractionEnd: (scale) {
            setState(() {
              controller1.value = Matrix4.identity();
              _visible = true;
            });
          },
          maxScale: 3.0,
          clipBehavior: Clip.none,
          child: Image.network(
            argsFossil.urlfossil2,
            fit: BoxFit.cover,

          ),  // Image.network
        )  // InteractiveViewer
      ),  // Flexible
```

Figura 4.41: Metodo build di FossilScreen (relativo all'ImageSlider)

4.4.12| TECNICHE UTILIZZATE IN SIGNIN, SIGNUP, POSTDETAILSTOFIRESTORE, LOGOUT

Dal punto di vista implementativo, anche se la maggior parte dei metodi è intrinseco nei plugin di *firebase_auth* [18][19] è sicuramente importante anche solo accennare i metodi che permettono all'utente di effettuare il *login*, *registrarsi* ed effettuare il *logout*.

```
//funzione di login
void signIn(String email, String password) async{
  if(_formKey.currentState!.validate())
{
  await _auth
      .signInWithEmailAndPassword(email: email, password: password)
      .then((uid) => {
      Fluttertoast.showToast(msg: "Login avvenuto con successo!"),
      Navigator.of(context).pushReplacement(MaterialPageRoute(builder: (context) => Profile())),
}).catchError((e)
  {
    Fluttertoast.showToast(msg: e!.message);
  });
}
}
```

Figura 4.42: Metodo signIn, per loggarsi nell'app

La schermata di *Login* altro non è che una form in cui inserire email e password per autenticarsi. Una volta premuto il bottone di Login partirà il metodo *signIn*, che prende come parametri l'email e la password inseriti nella form, poi li valuta e manda i valori della form a Firebase mediante il metodo *signInWithEmailAndPassword* che andrà a controllare se esiste un utente nel database che corrisponda a questi dati, dopodiché l'utente verrà portato nella schermata di profilo, Tutto ciò è visibile nella *Figura 4.42*.

Nella schermata di Login sarà anche possibile accedere alla schermata di registrazione nel caso in cui non si possedesse l'account per autenticarsi tramite un bottone.

```
void signUp(String email, String password) async{
  if(_formKey.currentState!.validate()){
    await _auth.createUserWithEmailAndPassword(email: email, password: password)
        .then((value) =>  {
          postDetailsToFirestore(),
    }).catchError((e){
      Fluttertoast.showToast(msg: e!.message);
    });

  }
}
```

Figura 4.43: Metodo signUp, per registrare un profilo utente nell'app

Per quanto riguarda la schermata di registrazione, avremo un'ulteriore form dove potremmo inserire: Nome, cognome, email, password e conferma password.

Al click sul pulsante 'Registrati' verrà eseguito il metodo *signUp*, che prende come parametri sempre l'email e la password, verifica che questi campi siano validi e verrà eseguito il metodo *createUserWithEmailAndPassword*. In seguito, viene eseguito un ulteriore metodo chiamato *postDetailsToFirestore*, come viene mostrato nella *Figura 4.43*.

```
postDetailsToFirestore() async{
  // calling our Firestore
  //Calling our user model
  //Sending these values

  FirebaseFirestore firebaseFirestore = FirebaseFirestore.instance;
  User? user = _auth.currentUser;

  UserModel userModel = UserModel();

  //writing all the values
  userModel.email = user!.email;
  userModel.uid = user.uid;
  userModel.firstName = firstNameEditingController.text;
  userModel.secondName = secondNameEditingController.text;

  await firebaseFirestore
  .collection("users")
  .doc(user.uid)
  .set(userModel.toMap());

  Fluttertoast.showToast(msg: "account creato con successo!");

  Navigator.of(context).pushReplacement(MaterialPageRoute(builder: (context) => Profile()));
}
```

Figura 4.44: Metodo postDetailsToFirestore, per inviare al database i dati dell''utente che si sta registrando

Questo metodo altro non fa che prendere i campi della form precedentemente compilata e metterli come valore per lo *UserModel*, definito nel file '*user_model.dart*'; infine si istanzia la collection '*users*' di Firebase dove vengono immagazzinati tutti gli utenti, e vengono anche inseriti tutti i dati relativi all'utente appena registrato, compreso anche un id univoco '*uid*' che poi ci servirà più avanti.

Infine l'utente, come nel caso di *signIn* visualizzerà la schermata di profilo, questa porzione di codice viene mostrata nella *Figura 4.44*.

```
Future<void> logout(BuildContext context) async{
  await FirebaseAuth.instance.signOut();
  Navigator.of(context).pushReplacement(
      MaterialPageRoute(builder: (context) => Login()));
}
```

Figura 4.45: Metodo signOut, per effettuare il logout

Infine, una volta giunti nella schermata di profilo sarà anche presente il metodo che permetterà all'utente di effettuare il logout semplicemente utilizzando il metodo *signOut*, che poi manderà l'utente nuovamente nella schermata di Login (*Figura 4.45*).

4.4.13| IL METODO BUILD DI PROFILE

Anche se questa classe non risulta per niente complicata dal punto di vista di codice, ho ritenuto importante menzionare anche questo passaggio perché ci servirà per le classi e metodi successivi.

Una volta giunti nella schermata di profilo viene subito istanziata una variabile *user* di tipo *User?* che mantiene il riferimento all'utente che in quel momento è loggato sull'applicazione e la variabile *loggedInUser* di tipo *UserModel* che viene messo uguale alla classe *UserModel*.

In seguito viene chiamata da Firebase la collection '*users*' e viene preso soltanto in considerazione l'utente loggato in quell'istanza, e infine vengono estrapolati i dati relativi all'utente (*Figura 4.46*).

```
User? user = FirebaseAuth.instance.currentUser;
UserModel loggedInUser = UserModel();

@override
void initState(){
  super.initState();
  FirebaseFirestore.instance
  .collection("users")
  .doc(user!.uid)
  .get()
  .then((value){
    this.loggedInUser = UserModel.fromMap(value.data());
    setState(){});
});
}
```

Figura 4.46: Prima parte del codice del profilo utente (estrazione dati utente loggato)

Il metodo build comprende la costruzione di una semplice **app bar**, che permette di tornare alla schermata precedente e di effettuare il *logout* (come detto sopra).

Nel **body** invece sarà semplicemente presente l'icona dell'applicazione, e il nome e cognome dell'utente loggato semplicemente utilizzando i dati precedentemente estrapolati da *loggedInUser*, come viene mostrato nell'immagine sottostante (*Figura 4.47*).

```
Padding(
  padding: EdgeInsets.all(16.0),
  child: Text(
    "${loggedInUser.firstName} ${loggedInUser.secondName}",
    style: TextStyle(
      fontSize: 20,
      color: Colors.black54,
      fontWeight: FontWeight.w500,
    ),  // TextStyle
  ),  // Text
),  // Padding

Padding(
  padding: EdgeInsets.all(16.0),
  child: Text(
    "${loggedInUser.email}",
    style: TextStyle(
      fontSize: 15,
      color: Colors.black54,
      fontWeight: FontWeight.w500,
    ),  // TextStyle
  ),  // Text
),  // Padding
```

Figura 4.47: Seconda parte del codice del profilo utente (implementazione grafica)

In fondo alla classe invece saranno presenti due bottoni:

- **Invia segnalazioni**: che aprirà la schermata relativa al Widget '*ImageUpload*'
- **Mostra Segnalazioni**: che aprirà la schermata relativa al Widget '*ShowUploads*'

4.4.14| IL METODO BUILD DI IMAGEUPLOAD

All'inizio della classe viene istanziata la variabile *userId* di tipo *String* che poi avrà il valore dello *userId* dell'utente loggato (come nella classe precedente).

Successivamente verranno inizializzate variabili per la *latitudine* e la *longitudine* (ancora senza valore), la variabile *_image* di tipo *File?*, la variabile *imagePicker* che servirà successivamente per prendere l'immagine scattata con la fotocamera grazie a dei metodi del package '*image_picker*', e infine la stringa *downloadURL* che conterrà l'url della foto scattata che verrà caricata nello ***Storage*** di Firebase (*Figura 4.48*).

```
class ImageUpload extends StatefulWidget{

  String? userId;

  ImageUpload({Key? key, this.userId}) : super(key: key);

  @override
  _ImageUploadState createState() => _ImageUploadState();
}

class _ImageUploadState extends State<ImageUpload>{

  var locationMessage = '';
  late String latitude;
  late String longitude;

  File? _image;
  final imagePicker = ImagePicker();

  var descrizione = '';


  String? downloadURL;
```

Figura 4.48: Prima parte del codice di ImageUpload (inizializzazione variabili)

Per quanto riguarda il metodo build di '*ImageUpload*' [19] invece, abbiamo sostanzialmente tre parti: una **app bar**, un **body** e una **FAB**.

```
return Scaffold(
  appBar: AppBar(
    title: const Text("Segnalazione!"),
    leading: IconButton(
      icon: const Icon(Icons.arrow_back, color: Colors.amber),
      onPressed: (){
        Navigator.of(context).pop();
      }),  // IconButton
    actions: <Widget>[
      IconButton( icon: const Icon(Icons.send, color: Colors.amber),
      onPressed: () {
      uploadImage();
      },
    ),  // IconButton
  // <Widget>[]
),  // AppBar
```

Figura 4.49: Seconda parte del codice di ImageUpload (appbar)

La **app bar** presenta semplicemente un bottone per tornare nella schermata precedente ed un bottone che attiverà il metodo '*uploadImage*', che caricherà la foto della segnalazione più le altre relative informazioni sulla data, la posizione geografica e la descrizione nel database Cloud Firestore (*Figura 4.49*).

```
Expanded(flex: 4,
  child: Container(
    width: 350,
    decoration: BoxDecoration(
        borderRadius: BorderRadius.circular(20),
        border: Border.all(color: Colors.redAccent)),  // BoxDecoration
  child: Center(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: [
        Expanded(
          child: _image == null
            ?const Center(
              child: Text("Nessuna Immagine selezionata"))  // Center
            : Image.file(_image!),
        ),  // Expanded
```

Figura 4.50: Terza parte del codice di ImageUpload (parte del body)

Nel **body** invece, sarà presente una form di testo per inserire una descrizione breve ed un Widget **Expanded**, al cui interno apparirà l'immagine scattata con la fotocamera, se non è stata scattata nessuna foto *(_image == null)* apparirà il testo ad avvisare che non è presente nessuna foto, altrimenti se *_image != null*, verrà visualizzata l'immagine mediante '*Image.file(_Image!)*' del package '*image_picker*' spiegato nei precedenti capitoli (*Figura 4.50*).

Infine nella parte inferiore della schermata sarà presente una **FAB** con l'immagine di una fotocamera che farà partire il metodo '*imagePickerMethod*', che interagirà con la fotocamera del dispositivo.

4.4.15| TECNICHE UTILIZZATE IN IMAGEPICKERMETHOD E UPLOADIMAGE

Il metodo '*imagePickerMethod*' semplicemente ha il compito di interagire con la fotocamera del dispositivo, per permettere all'utente di scattare la foto abbassando poi la qualità dell'immagine a massimo 500KB (*imageQuality: 50*).

Dopodiché, se l'immagine è stata scattata, essa verrà visualizzata nel **body**, altrimenti apparirà uno **snackbar** che notificherà l'utente che non ha selezionato nessun file. Tutto ciò viene mostrato nella porzione di codice rappresentato nella *Figura 4.51.*

```
//image picker
Future imagePickerMethod() async{
  //picking the file
  final pick = await imagePicker.pickImage(source: ImageSource.camera, imageQuality: 50,);

  setState(() {
    if(pick != null){
      _image = File(pick.path);
    }
    else{

      showSnackBar("Nessun File è stato selezionato", Duration(milliseconds: 400));

    }
  });
}
```

Figura 4.51: Metodo ImagePickerMethod

Invece il metodo '*uploadImage*' risulta essere più complesso:

Innanzitutto viene istanziata la variabile *time* alla quale viene associata la data corrente con '*DateTime.now()*', dopodiché si converte in *TimeStamp* istanziando la variabile *timeStamp*.

Dopodiché con il package *Geolocator* si prende la posizione corrente dell'utente, che poi verrà divisa in latitudine e longitudine con i metodi '*latitude*' e '*longitude*'. Poi viene istanziata la variabile *geo* del package '*GeoFlutterFire2*', ed infine sempre con questo package l'applicazione riesce a prendere i dati relativi alla latitudine e alla longitudine dell'utente per poi trasformarlo in un unico punto geografico.

Infine si carica la foto che era stata scattata con '*imagePickerMethod*' all'interno dello **Storage** nella cartella '/foto *segnalazioni*' prendendo la variabile *downloadURL* con la quale si prende il riferimento all'url della foto caricata si Firebase (*Figura 4.52*).

```
Future uploadImage() async{

  final postID = DateTime.now().millisecondsSinceEpoch.toString();

  final time = DateTime.now();
  Timestamp timeStamp = Timestamp.fromDate(time);

  FirebaseFirestore firebaseFirestore = FirebaseFirestore.instance;

    var position = await Geolocator.getCurrentPosition(
        desiredAccuracy: LocationAccuracy.high);
    var lat = position.latitude.toDouble();
    var long = position.longitude.toDouble();
    latitude = "$lat";
    longitude = "$long";
    final geo = GeoFlutterFire();

    GeoFirePoint myLocation = geo.point(latitude: lat, longitude: long);


    //writing all the values


    Reference ref = FirebaseStorage.instance
        .ref()
        .child("/foto segnalazioni")
        .child("post_$postID");

    await ref.putFile(_image!);
```

Figura 4.52: Prima parte di codice del metodo uploadImage (campionamento dati relativi alla segnalazione)

Come ultima operazione, vengono presi i dati relativi alla descrizione, all'url della pagina, alla data di scatto, alla posizione dell'utente che ha scattato la foto, che poi verrà caricata su Firebase nella collection '*segnalazioni*' (*Figura 4.53*).

```
downloadURL = await ref.getDownloadURL();

//Uploading to cloudFirestore
await firebaseFirestore
    .collection("segnalazioni")
    .add({
  'descrizione': descrizione,
  'downloadURL': downloadURL,
  'data': timeStamp,
  'position': myLocation.geoPoint,
  'user': widget.userId
})
    .whenComplete(() =>
    showSnackBar(
        "Immagine caricata con successo!",
        Duration(seconds: 2)));
}
```

Figura 4.53: Seconda parte di codice del metodo uploadImage (caricamento segnalazione nel cloud Firestore)

4.4.16| IL METODO BUILD DI SHOWUPLOADS

Ultima classe, ma non per importanza, da trattare è quella di '*ShowUploads*' [20], dove verranno visualizzate tutte le segnalazioni effettuate dall'utente autenticato.

Anche in questo caso dalla schermata di profilo viene passato lo '*uid*' dell'utente loggato, dopodiché da Firebase vengono chiamati tutti i documenti relativi alla collection '*segnalazioni*' in cui l'utente che ha effettuato la segnalazione (ricordando che lo '*uid*' è memorizzato anche come campo delle

segnalazioni) è uguale allo '*uid*' dell'utente loggato, dopodiché viene mostrata una lista di tutte le foto delle varie segnalazioni in ordine di data (*Figura 4.54*).

```
class _ShowUploadsState extends State<ShowUploads>{
  @override
  Widget build(BuildContext context){
    return Scaffold(
      appBar: AppBar(title: const Text("Le tue segnalazioni"),),
      body: StreamBuilder(
        stream: FirebaseFirestore.instance.collection("segnalazioni")
            .where("user", isEqualTo: widget.userID)
            .orderBy('data'.toString(), descending: true).snapshots(),

        builder:
          (BuildContext context, AsyncSnapshot<QuerySnapshot> snapshot) {
          if(!snapshot.hasData){
            return (const Center(child: Text("Nessuna segnalazione caricata")));

          } else {
            return ListView.builder(
              itemCount: snapshot.data!.docs.length,
              itemBuilder: (BuildContext context, int index){
        String url = snapshot.data!.docs[index]['downloadURL'];

        Timestamp t = snapshot.data!.docs[index]['data'];
        DateTime data = t.toDate();
```

Figura 4.54: Codice di ShowUploads

Bisogna notare che per ordinare le segnalazioni dalla più recente alla più remota si sono utilizzati i metodi '*orderBy('data'.toString(), descending: true).snapshot()*', tuttavia per effettuare l'ordinamento decrescente è stato necessario impostarlo negli indici del Cloud Firestore, in questo modo (*Figura 4.55*):

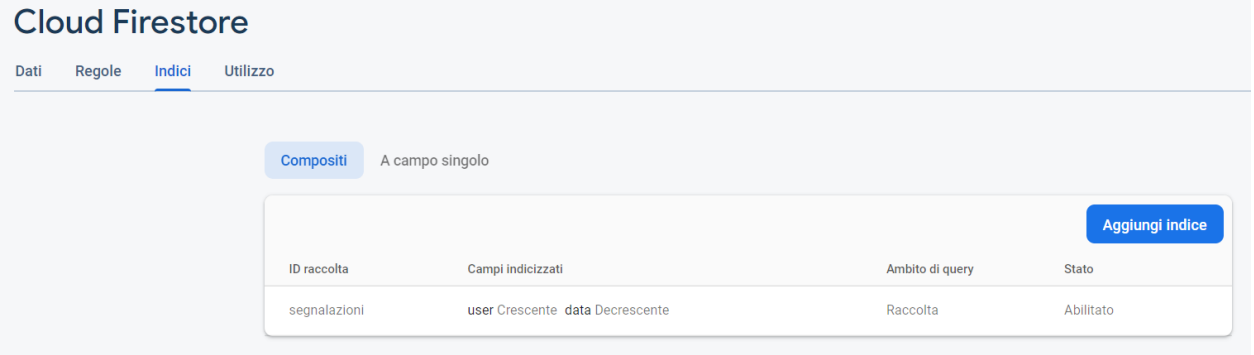

Figura 4.55: Indici del Cloud Firestore in cui si dichiara l'ordine decrescente nell'ordinamento delle segnalazioni

4.5| SCREENSHOT DELLE SCHERMATE

In questa sezione sono presenti gli screenshot delle varie schermate, al fine di mostrare il risultato finale dell'applicazione.

Figura 4.56: Onboarding (prima schermata)

Figura 4.57: Onboarding (seconda schermata)

Figura 4.58: Onboarding (terza schermata)

Figura 4.59: Onboarding (quarta schermata)

Figura 4.60: Onboarding (ultima schermata)

Figura 4.61: Home

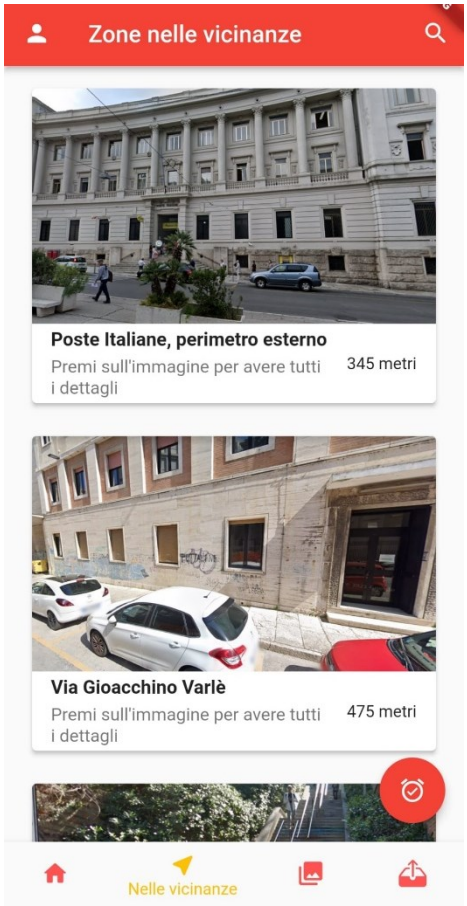

Figura 4.62: Zone nelle vicinanze

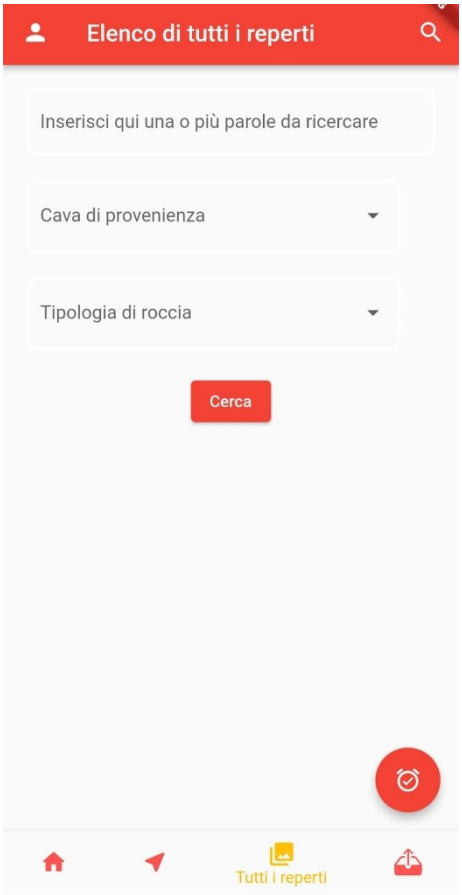

Figura 4.63: Tutti i reperti

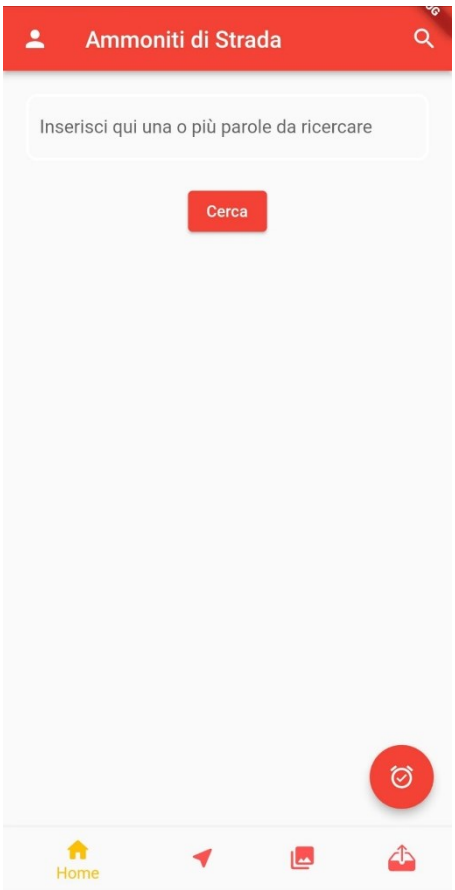

Figura 4.64: Ricerca per zone

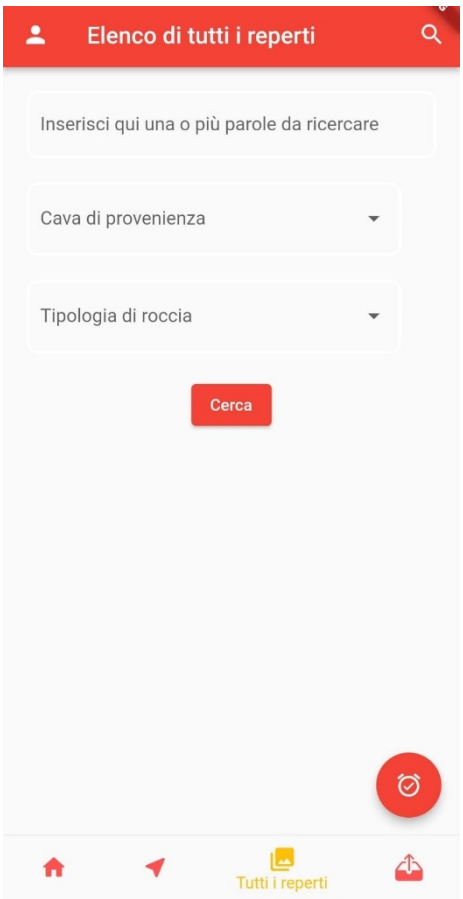

Figura 4.65: Ricerca per fossili

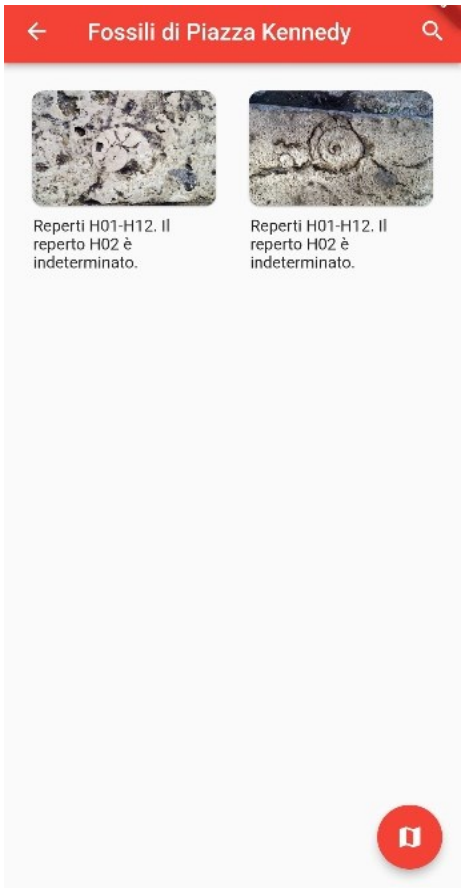

Figura 4.66: ZoneScreen

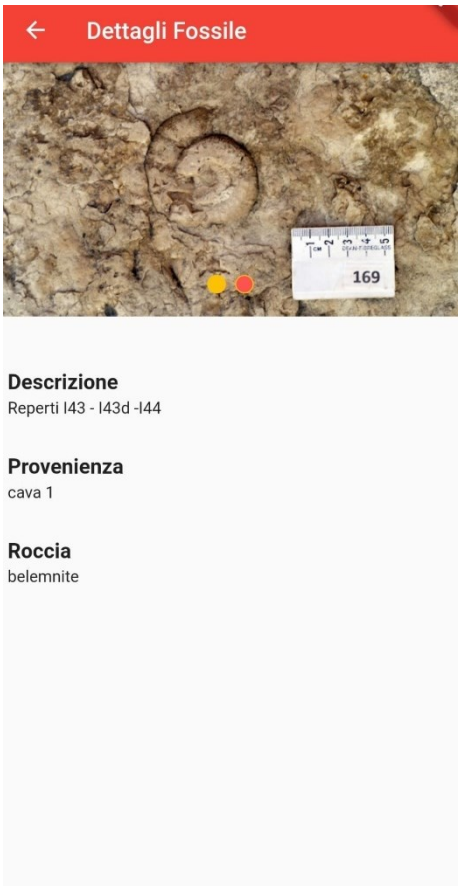

Figura 4.67: Dettaglio fossile

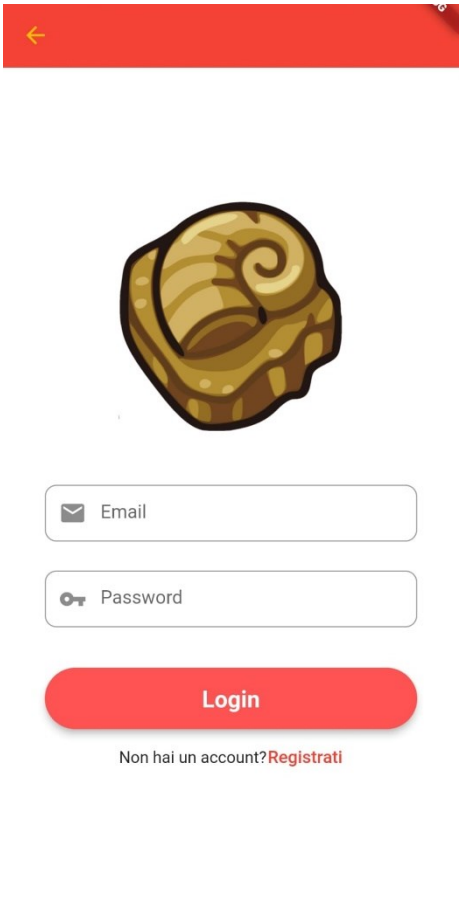

Figura 4.68: Login

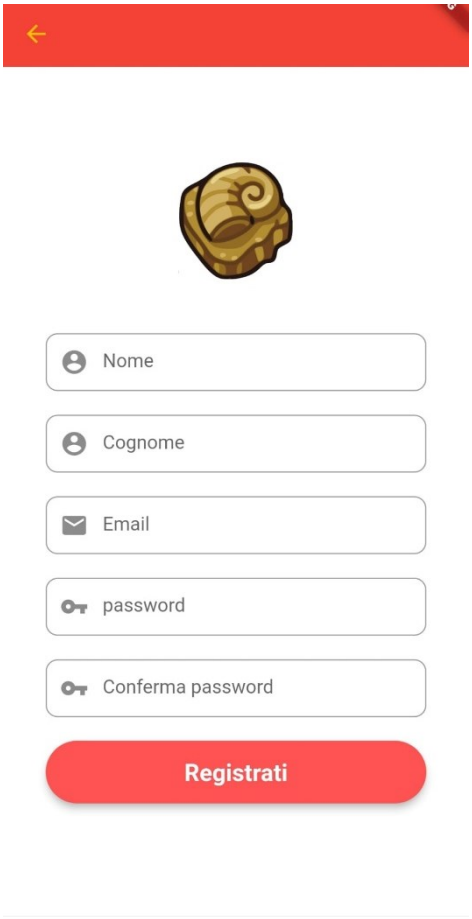

Figura 4.69: Registrazione

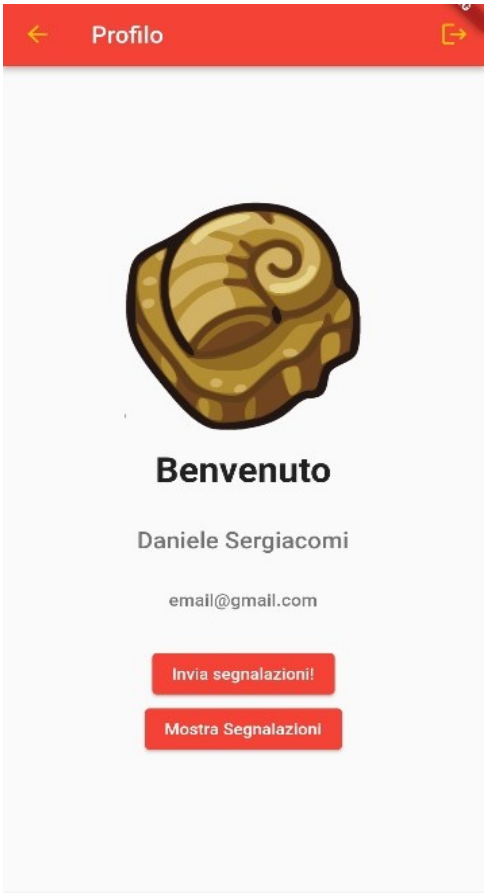

Figura 4.70: Profilo

Figura 4.71: Invia segnalazioni

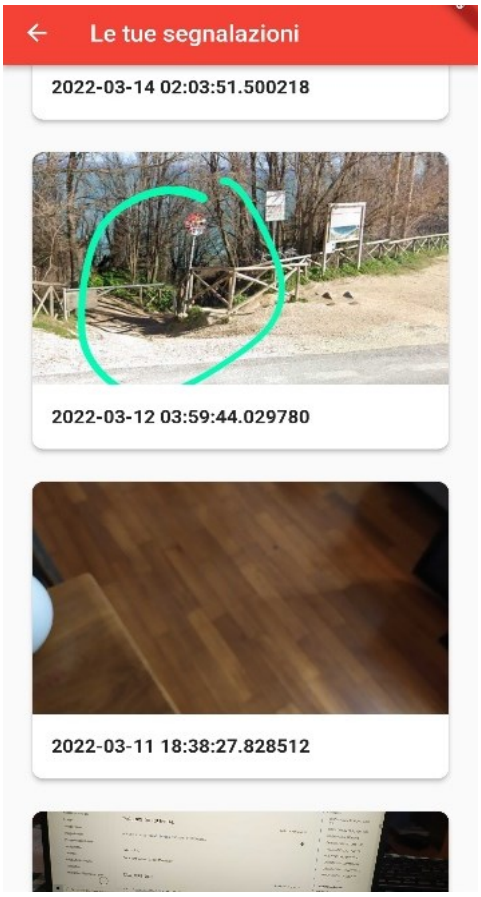

Figura 4.72: Mostra segnalazioni

CONCLUSIONE:

L'obiettivo di questa tesi è stato quello di discutere le fasi di progettazione e realizzazione di un'applicazione mobile che permettesse la fruizione del patrimonio paleontologico, ed in particolare fossili presenti nella città di Ancona e che offrisse inoltre agli utenti la possibilità di segnalare altri reperti, per poter arricchire sempre di più la cultura storica della città. Nel primo capitolo si è discusso dei requisiti minimi dell'applicazione e nel complesso l'app li ha rispettati, quindi oltre a modificare i precedenti aspetti già implementati, si è deciso di aggiungere una suddivisione per zone che a loro volta contengono altri fossili, e un'aggiunta di un ulteriore foto per il singolo fossile, oltre che (come detto qui sopra) un sistema di autenticazione per permettere all'utente di effettuare delle segnalazioni.

L'utilizzo di Android Studio, il linguaggio Dart con il framework Flutter e la piattaforma Firebase hanno giocato un ruolo fondamentale, in quanto hanno reso lo sviluppo dell'applicazione un'ottima occasione per accrescere le proprie competenze personali.

L'applicazione sviluppata ha sicuramente margini di miglioramento, tuttavia rappresenta un'ottima base di partenza per un'eventuale applicazione ufficiale della città di Ancona (e non solo) per la rappresentazione di reperti.

Sono presenti anche degli spunti per un possibile sviluppo futuro per il miglioramento dell'applicazione:

- Per ora nella notifica viene riportato il numero di zone nei dintorni, e quando si preme sulla notifica viene aperta l'homepage. Questa funzionalità in futuro potrebbe essere modificata, mettendo nella notifica un'anteprima di qualche zona nelle vicinanze, e consentendo l'apertura della pagina con la lista dei fossili dello specifico luogo al tocco dell'utente sulla notifica.

- Attualmente all'interno dell'app non è presente un'impostazione per disattivare le notifiche.

- Per migliorare l'usabilità, potrebbe anche essere interessante diversificare i Marker visualizzati sulla mappa, in futuro i Marker potrebbero essere di forme e colori differenti in base ad altri campi come ad esempio il tipo di luogo (muro, via, scalinata…).

- Un'ulteriore funzione da implementare sarebbe quella di geo-referenziare anche i fossili all'interno delle zone, per poter poi inserire nella schermata di Dettaglio fossili, un bottone "Visualizza sulla Mappa", che poi riporterà l'utente sulla mappa della homepage in corrispondenza della posizione del fossile.

- Si potrebbero geo-referenziare i reperti anche per poter creare delle vere e proprie tappe per visitare più fossili a prescindere dalla zona in cui si trovano

- Attualmente è possibile inserire solamente due immagini per fossile, in quanto se si decidesse di metterne solamente una l'applicazione crasherebbe, mentre non è comunque possibile mettere arbitrariamente 'n' foto, in quanto l'image slider è staticamente impostato con un limite di 2 immagini

- Infine, nella schermata di 'Invia segnalazioni' si potrebbe aggiungere una FAB aggiuntiva che permetterebbe di scegliere il reperto da segnalare tramite la galleria delle foto, e non solo direttamente tramite la fotocamera; tuttavia per fare ciò è necessario lavorare con le exif delle immagini per poter poi ricavare latitudine e longitudine da ciascuna foto da caricare.

BIBLIOGRAFIA:

[1] Wikipedia, Fossile. [Online]. Available: https://it.wikipedia.org/wiki/Fossile

[2] Wikipedia, Ammonoidea. [Online]. Available: https://it.wikipedia.org/wiki/Ammonoidea

[3] Wikipedia, Belemnoidea. [Online]. Available: https://it.wikipedia.org/wiki/Belemnoidea

[4] Tesi Matteo Cirilli, "Ammoniti di strada" – Università Politecnica delle Marche - *2022-02-24.* [Online]. Available: https://tesi.univpm.it/bitstream/20.500.12075/8120/1/Tesina%20Ammoniti%20di%20strada.pdf

[5] Raffaella Papa, Tecnoandroid, 14 Gennaio 2019. [Online]. Available: https://www.tecnoandroid.it/2019/01/14/androidcose-come-fatto-e-come-funziona-il-sistema-operativo-di-google-460374

[6] ZOE S.R.L.S., «3 TIPOLOGIE DI APP PER DISPOSITIVI MOBILE. QUALE È ADATTA AL TUO PROGETTO?» 2020. [Online]. Available: https://www.zoewebsolutions.it/3-tipologie-di-app-perdispositivi-mobile-quale-%C3%A8-adatta-al-tuo-progetto

[7] Lucidchart. UML Use Case Diagram Tutorial. [Online]. Available: https://www.lucidchart.com/pages/uml-use-case-diagram

[8] Google LLC, «Android Studio». [Online]. Available: https://developer.android.com/studio

[9] JetBrains s.r.o., «IntelliJ IDEA». [Online]. Available: https://www.jetbrains.com/idea/

[10] Dart, features overview. [Online]. Available: https://dart.dev

[11] "Future of JavaScript" doc from our internal "JavaScript Summit" last week. [Online]. Available: https://gist.github.com/paulmillr/1208618

[12] "Perché?", Dart lang (FAQ). [Online]. Available: https://dart.dev/faq#why-dart

[13] Google LLC, «Flutter» 2021. [Online]. Available: https://flutter.dev/

[14] C. Garau, «Git e il controllo versione distribuito» 2016. [Online]. Available: https://www.html.it/pag/52561/git-e-il-controllo-di-versione-distribuito/

[15] Flutter, using packages. [Online]. Available: https://docs.flutter.dev/development/packages-and-plugins/

[16] List of all Flutter packages. [Online]. Available: https://pub.dev/packages

[17] Johannes Milke <<Flutter Tutorial - Onboarding Screen UI, Introduction Screen, Walkthrough [2022]>> 2022 [Online]. Available: https://www.youtube.com/watch?v=AmsXazhGMQ0&t=113s

[18] BackSlash Flutter <<Flutter and Firebase Authentication | Email & Password Login/SignUp (Flutter 2.2) | Part 1>> 2021 [Online]. Available: https://www.youtube.com/watch?v=3DO6Th9A7mY

[19] BackSlash Flutter << Flutter & Firebase Auth | Implementing Cloud Firestore & Catching Errors (Flutter 2.2) | Part 2>> 2021 [Online]. Available: https://www.youtube.com/watch?v=DthKmtjth1k&t=2885s

[20] BackSlash Flutter << Flutter Firebase Image Upload/Retrieve Using Firebase Storage & StreamBuilder (Flutter 2.5)>> 2021 [Online]. Available: https://www.youtube.com/watch?v=fJr_-EP4QpI&t=1895s