UNIVERSITÀ
POLITECNICA
DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica e

dell'Automazione

'Sviluppo di un'applicazione mobile in Flutter per la valorizzazione del patrimonio paleontologico di Ancona tramite Realtà Aumentata e gamification'.

'Development of an mobile application in Flutter for the valorization of paleontological heritage of Ancona through, Augmented Reality and gamification'.

Relatore:
Prof. Emanuele Storti

Tesi di Laurea di:
Faccenda Andrea

A.A. 2022/2023

“I fossili sono la chiave di volta per capire l’evoluzione della vita.”

Alberto Angela

Sommario

Introduzione

La paleontologia è una scienza affascinante che ci permette di esplorare il passato della Terra attraverso le tracce fossili lasciate da piante, animali e organismi marini. Il nostro territorio è ricco di storia geologica e paleontologica, che offre una finestra su un mondo antico e sconosciuto.
Attraverso lo studio dei fossili locali, possiamo comprendere non solo la vita antica, ma anche qual è il nostro posto nella storia del pianeta. Preservare e capire questo patrimonio paleontologico è fondamentale per le generazioni future e per una comprensione più profonda della nostra storia naturale.
Ed è proprio dall'importanza scientifico-culturale di questo tema che ha inizio questo progetto.
Nella città di Ancona, in particolare nei monumenti storici e nelle pavimentazioni, le rocce sono sede di reperti antichi. Molte delle pietre ornamentali sono di origine marina; in particolare Ancona risulta essere ricca di ammoniti.
Le ammoniti sono oggetto di studio sia per i paleontologi che per gli appassionati di geologia e paleontologia. Le loro conchiglie sono spesso trovate in rocce sedimentarie e sono utilizzate sia come indicatori di età geologica sia come strumento per comprendere l'evoluzione e l'ambiente marino del passato.

Figura 0.1 ammonite

In tale ambito agisce il progetto 'AmmoniteExplorer', in collaborazione con il Dipartimento di Scienze della Vita e dell'Ambiente dell'Università Politecnica delle Marche, il quale ha come obbiettivo principe la valorizzazione del patrimonio geologico della città di Ancona e il coinvolgimento degli utenti tramite l'uso della gamification**.**
Per creare un'esperienza interattiva l'applicazione mira all'utilizzazione della realtà aumentata che combina il mondo reale con elementi virtuali o digitali mirando a migliorare l'engagement e arricchire la conoscenza delle persone verso la realtà geologica del territorio.
La realtà aumentata è stata utilizzata per creare esperienze educative e coinvolgenti, come la visualizzazione interattiva di modelli 3D delle varie ammoniti presenti nel territorio anconetano.
Nel documento di tesi, la realtà aumentata verrà chiamata con l'acronimo di AR(Augmented Reality).
L'app 'AmmoniteExplorer 'vanta una gamma di funzionalità all'avanguardia, tra cui:

- Elementi AR interattivi: gli utenti saranno in grado di interagire con le ammoniti virtuali e informazioni sovrapposte al mondo reale.

- Mappe interattive: gli utenti possono esplorare mappe che vanno oltre le immagini statiche, interagendo con layer di dati e con i marker di tutte le ammoniti geolocalizzate.

- Interfaccia user-friendly: l'esperienza AR è progettata per essere user-friendly e intuitiva, rendendola accessibile a un vasto pubblico.

- Servizio di navigazione GPS: fornisce indicazioni di navigazione passo-passo per varie modalità di trasporto, tra cui guida, camminare e andare in bicicletta. Calcola i percorsi ottimali verso i vari ammoniti in base ai dati sul traffico in tempo reale, alle condizioni stradali e alle preferenze dell'utente.

- Visualizzazione dei dettagli di ogni ammonite: fornisce la possibilità di osservare tutte le informazioni principali di ogni reperto analizzando quindi la storia, il tipo di specie, la roccia dove è sedimentata, il periodo storico rappresentando una vera enciclopedia virtuale di ogni specie di ammonite.

Il documento che è presentato al lettore consiste nella relazione di ogni singola fase, descritta accuratamente nei vari capitoli, che hanno portato allo sviluppo dell'applicazione mobile.

Di seguito è riportato un sunto del contenuto dei vari capitoli:

• Nel Capitolo 1 è descritta la fase di analisi dei requisiti, 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.

• Nel Capitolo 2 è descritto il diagramma dei casi d'uso, in cui vengono riassunte le possibili interazioni tra l'utente e l'applicazione. Oltre al diagramma, in questo capitolo sono visualizzati dei mockup grafici delle schermate. Essendo a conoscenza delle funzionalità che l'app avrebbe messo a disposizione, queste schermate sono state collegate logicamente, immaginando i possibili percorsi all'interno dell'app causati dall'interazione dell'utente.

• Nel Capitolo 3 vengono presentati i vari strumenti (editor grafici, IDE, frameworks…) utilizzati durante lo svolgimento del progetto.

• Nel Capitolo 4 è riportata 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. Le componenti sono state poi collegate logicamente, definendone le interazioni. In seguito, si è passati alla progettazione dei dati, ed infine si è proceduto con lo sviluppo dell'applicazione.

1. Analisi dei requisiti

Prima di affrontare un qualsiasi tipo di lavoro è bene riflettere anticipatamente sul risultato che si vuole ottenere o che ci è richiesto di raggiungere. Per questo motivo è fondamentale un'attenta analisi delle caratteristiche che il prodotto finale dovrà soddisfare affinché si possa considerare valido.
I requisiti sono dedotti dalle funzioni che l'applicazione deve garantire al termine dello sviluppo, in particolare sono una raccolta di tutti i servizi che si vogliono offrire all'utente (requisiti funzionali) e, dei vincoli a cui quest'ultimi sono inevitabilmente collegati requisiti non funzionali).

1.1 Requisiti funzionali

1.1.1 *Requisiti funzionali spiegazione*

Requisito	Descrizione
RF1 Registrazione Utente	*Il sistema dovrà gestire la registrazione di un nuovo utente a sistema.*
RF2 Autenticazione	*Il sistema dovrà gestire l'autenticazione degli utenti registrati .*
RF3 Recupero Password	*Il sistema dovrà gestire una procedura per il recupero della password.*
RF4 Visualizzazione Reperti sulla Mappa	*Il sistema dovrà gestire la visualizzazione dei marker dei reperti sulla mappa.*
RF5 Servizi di Geolocalizzazione	*Il sistema dovrà gestire i servizi di geolocalizzazione dei vari reperti archeologici*
RF6 Ricerca del Reperto	*Il sistema dovrà gestire la ricerca un reperto presente all'interno del database.*
RF7 Visualizzazione Posizione attuale	*Il sistema dovrà permettere all'utente di visualizzare in real time la posizione sulla mappa*
RF8 Visualizzazione Distanza dai Reperti	*Il sistema dovrà fornire la distanza dalla posizione attuale alla posizione del reperto.*
RF9 Navigatore per la Ricerca dei Reperti	*Il sistema dovrà gestire i servizi di navigazione a disposizione dell'utente.*
RF10 Visualizzazione Dettagli dei Reperti	*Il sistema dovrà fornire tutte le informazioni presenti nel database dei vari fossili.*
RF11 Visualizzazione 3D dei Reperti	*Il sistema dovrà consentire la visualizzazione 3D dei reperti.*
RF12 Catturare i Reperti	*Il sistema dovrà gestire la cattura dei reperti 3D.*
RF13 Interagire con i Modelli 3D dei Reperti	*Il sistema dovrà gestire le operazioni di interazione dell'utente con i reperti 3D.*
RF14 Visualizzazione dello Zaino	*Il sistema dovrà gestire la visualizzazione dello zaino contenente tutti i reperti catturati.*
RF15 Sistema di notifiche	*Il sistema dovrà garantire l'invio di notifiche nel momento in cui l'utente si trovi nei pressi dei reperti.*
RF16 Logout	*Il sistema dovrà permettere all'utente di disconnettersi dall'applicazione.*

L’utente deve avere la possibilità di visualizzare sulla mappa i marker corrispondenti ai vari reperti presenti nel database. L’app deve mettere a disposizione servizi di geolocalizzazione, che permettano all’utente di visualizzare in tempo reale la propria posizione su una mappa. Grazie alla conoscenza della sua posizione l’app deve essere in grado di visualizzare in una schermata i reperti che si trovano nelle vicinanze e deve fornire la funzione di navigazione per permettere all’utente di recarsi nella posizione dell’ammonite. Devono essere inoltre presenti funzionalità di ricerca, tramite le quali si possano rendere visibili solo i reperti che soddisfano i parametri di ricerca dell’utente, ovviamente se presenti nel database. Questo filtraggio dovrà essere applicabile alla lista dei reperti. Ovviamente per ogni ammonite deve essere presente la specifica pagina in cui sia possibile visualizzarne foto e descrizione.
Il sistema deve essere in grado di rilevare superfici piane o irregolari nel mondo reale, come pavimenti, pareti o tavoli, in modo che gli oggetti virtuali possano essere ancorati a queste superfici.
L'app deve consentire agli utenti di interagire con gli elementi virtuali, ad esempio toccandoli, trascinandoli e , inoltre deve gestire le operazioni per tracciare ammoniti virtuali nel mondo reale.
Il sistema deve far apparire il reperto 3D solo quando l’utente è molto vicino e gestire tutte le possibili operazioni di interazione con l’utilizzatore.
L’app deve avere inoltre la funzionalità che permette all’utente di catturare un oggetto virtuale e di inserire tale modello 3D all’interno dello zaino.

1.2 Requisiti non funzionali

1.2.1 *Requisiti non funzionali spiegazione*

Requisito	Descrizione
RNF1 Implementazione	*Il sistema dovrà essere realizzato in tecnologia Flutter.*
RNF2 Interfaccia grafica	*Il sistema dovrà essere dotato di interfaccia grafica.*
RNF3 Firestore Database	*Il sistema dovrà gestire una procedura per il recupero della password.*
RNF4 Permessi	*Al sistema dovranno essere garantiti i permessi per permettere il corretto funzionamento di alcuni servizi dell'app.*

2. Casi d'uso

Il diagramma dei casi d'uso è uno strumento che rappresenta la relazione tra un utente, definito come attore, e le sue richieste o aspettative rispetto al sistema. Questo diagramma è adatto a rappresentare le funzioni principali e/o gli obiettivi di un sistema software in maniera chiara [2]. Il diagramma dei casi d'uso specifica come un sistema si comporterà, ed è per questo che mostra solo la funzionalità del sistema.

2.1 *Diagramma dei casi d'uso*

Nel diagramma riportato in Figura 2.1 l'attore viene rappresentato come un omino stilizzato, secondo le convenzioni dell'ingegneria del software sui casi d'uso, mentre le funzioni dell'applicazione sono rappresentate tramite delle ellissi con all'interno la relativa descrizione.

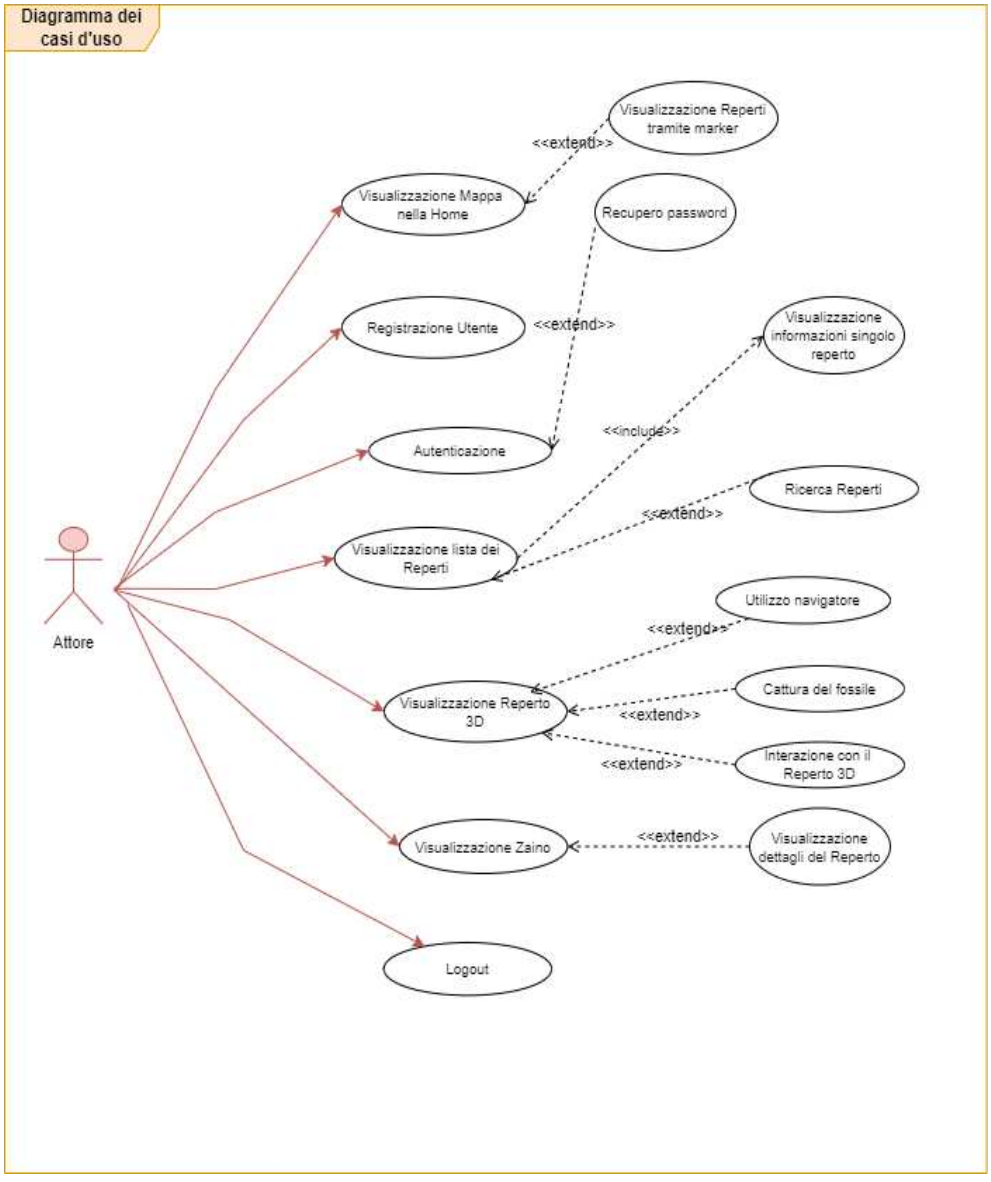


Figura 2.1

2.1.1 *Descrizione dei casi d'uso*

Caso d'uso: ***Registrazione Utente***

Attori: Utente.

Questo caso d'uso si verifica quando l'utente vuole registrarsi a sistema.

Pre-condizioni: l'utente non esiste a sistema.

Post-condizioni: l'utente viene registrato a sistema (o non è stato possibile registrarlo).

Sequenza degli eventi principale:

1. Il caso d'uso ha inizio quando l'utente decide di effettuare la registrazione e creare un account.
2. L'utente avvia l'applicazione e avvia la procedura di accesso.
3. Il sistema visualizza a schermo la schermata di login.
4. L'utente richiede di accedere alla schermata di registrazione.
5. Il sistema visualizza la schermata di registrazione.
6. L'utente inserisce le informazioni necessarie.
7. L'utente avvia la procedura di registrazione.
8. Il sistema registra il nuovo utente a sistema.

Sequenza degli eventi alternativa:

La sequenza alternativa inizia dal punto 7.

1. Le informazioni inserite dall'utente sono incomplete.
2. La procedura di registrazione non va a buon fine.

Caso d'uso: ***Visualizzazione lista dei reperti***

Attori: Utente.

Il caso d'uso si verifica quando l'utente decide di visualizzare la lista dei reperti presenti nel database.

Pre-condizioni: nessuna.

Post-condizioni: nessuna.

Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente vuole visualizzare i reperti presenti nel database.
2. L'utente accede alla schermata per la visualizzazione della lista.
3. Il sistema visualizza a schermo i reperti.

Sequenza degli eventi alternativa: nessuna

Caso d'uso: ***Ricerca reperto***

Attori: Utente.

Il caso d'uso inizia quando l'utente decide di effettuare una ricerca tra i reperti presenti nel database.

Pre-condizioni: il reperto esiste a sistema.

Post-condizioni: nessuna.

Sequenza degli eventi principale:

1. Il caso d'uso ha inizio quando l'utente intende effettuare una ricerca tra i reperti presenti.
2. Il sistema visualizza la barra di ricerca e il filtro disponibile.
3. L'utente avvia la ricerca con il filtro scelto.
4. Il sistema legge il filtro applicato dall'utente per la ricerca (nome).
5. Il sistema visualizza a schermo i reperti filtrati.

Sequenza degli eventi alternativa: nessuna.

Caso d'uso: ***Visualizzazione zaino***

Attori: Utente.

Il caso d'uso si verifica quando l'utente decide di visualizzare il contenuto dello zaino.

Pre-condizioni: lo zaino esiste a sistema.

Post-condizioni: nessuna.

Sequenza degli eventi principale:

1. Il caso d'uso ha inizio quando l'utente vuole visualizzare lo zaino.
2. L'utente avvia la procedura di visualizzazione dello zaino.
3. Il sistema visualizza a schermo il contenuto dello zaino.

Caso d'uso: ***Recupero Password***

Attori: Utente.

Il caso d'uso ha inizio qualora l'utente abbia bisogno di recuperare la propria password.

Pre-condizioni: l'utente è registrato nel sistema.

Post-condizioni: l'utente riceve una password di recupero con cui effettuare l'autenticazione con successo.

Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente vuole recuperare la propria password di accesso.
2. L'utente accede alla schermata di login.
3. L'utente accede alla schermata di recupero password.
5. L'utente inserisce la propria e-mail.
6. Il sistema invia nell'e-mail indicata la password con la quale effettuare l'autenticazione.

Caso d'uso: *Logout*

Attori: Utente, Dipendente.

Il caso d'uso si verifica quando l'utente o il dipendente vogliono effettuare il logout.

Pre-condizioni: utente ha effettuato il login in precedenza.

Post-condizioni: l'utente viene disconnesso dall'applicazione.

Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente decide di effettuare il logout.
2. L'utente accede alla schermata di Impostazioni.
3. L'utente accede alla schermata di visualizzazione dell'account.
4. L'utente avvia la procedura di logout.
5. Il sistema disconnette l'utente dall'applicazione.

Sequenza degli eventi alternativa: nessuna.

Caso d'uso: *Autenticazione*

Attori: Utente, Dipendente.

Questo caso d'uso si verifica quando l'utente o il dipendente effettuano il login per la prima volta o hanno effettuato un logout in precedenza.

Pre-condizioni: l'utente ha le credenziali di accesso.

Post-condizioni: l'utente accede all'applicazione (o non è stato possibile permettere l'accesso).

Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente vuole effettuare l'autenticazione per utilizzare l'applicazione.
2. L'utente accede alla schermata di login.
3. L'utente inserisce le proprie credenziali di accesso (e-mail e password).
4. L'utente avvia la procedura di autenticazione.
5. Il sistema visualizza a schermo la home page dell'utente.

Sequenza degli eventi alternativa:

La sequenza alternativa inizia dal punto 4.

1. Le credenziali inserite non sono valide.
2. La procedura di autenticazione non va a buon fine.

Caso d'uso: ***Visualizzazione dettagli reperto***

Attori: Utente.

Il caso d'uso ha inizio quando l'utente decide di visualizzare i dettagli relativi ad un reperto.

Pre-condizioni: il reperto esiste a sistema.

Post-condizioni: nessuna.

Sequenza degli eventi principale:

2. Il caso d'uso ha inizio quando l'utente vuole visualizzare le informazioni relative ad un singolo reperto.
2. L'utente accede alla lista e seleziona un reperto.
3. Il sistema legge le informazioni del reperto scelto.
4. Il sistema visualizza a schermo i dettagli del reperto scelto dall'utente inoltre permette la visualizzazione del modello 3D.

Sequenza degli eventi alternativa: nessuna.

Caso d'uso: ***Visualizzazione reperto 3D***

Attori: Utente.

Il caso d'uso ha inizio quando l'utente si trova vicino alla posizione geografica del fossile(distanza di 5 metri).

Pre-condizioni: il reperto esiste a sistema.

Post-condizioni: nessuna.

Sequenza degli eventi principale:

1. Il caso d'uso ha inizio quando l'utente è molto vicino all'ammonite.
2. Il sistema legge le informazioni del reperto scelto.
3. Il sistema visualizza a schermo i dettagli del reperto scelto dall'utente inoltre permette la visualizzazione del modello 3D.
4. L'utente interagisce con il modello 3D del fossile.

Sequenza degli eventi alternativa: nessuna.

Caso d'uso: ***Utilizzo navigatore***

Attori: Utente.

Questo caso d'uso si verifica quando l'utente preme il pulsante del navigatore.

Pre-condizioni: l'utente esiste a sistema.

Post-condizioni: nessuna.

Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente preme il bottone.
2. Il sistema visualizza a schermo la strada da percorrere per raggiungere il reperto.

Sequenza degli eventi alternativa: nessuna.

Caso d'uso: ***Visualizzazione Mappa nella home***

Attori: Utente.

Questo caso d'uso si verifica quando l'utente decide di visualizzare la mappa che contiene i marker che identificano la presenza di reperti.

Pre-condizioni: l'utente esiste a sistema.

Post-condizioni: nessuna.

Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente effettua il login.
2. Il sistema visualizza a schermo l'home page contenente la mappa.

Sequenza degli eventi alternativa: nessuna.

Caso d'uso: ***Cattura del fossile***

Attori: Utente.

Questo caso d'uso si verifica quando l'utente decide di catturare l'ammonite 3D.

Pre-condizioni: l'utente esiste a sistema.

Post-condizioni: nessuna.

Sequenza degli eventi principale:

1. Il caso d'uso inizia quando l'utente si trova vicino alle coordinate dell'ammonite.
2. Il sistema permette la visualizzazione del modello 3D del reperto.
3. L'utente premendo il bottone cattura può catturare il modello dell'ammonite.

Sequenza degli eventi alternativa: nessuna.

3. Strumenti utilizzati

In ingegneria del software è fondamentale conoscere e saper utilizzare gli strumenti che abbiamo a disposizione per realizzare i software.
In questo capitolo sono spiegate le caratteristiche generali dei programmi, delle varie librerie utilizzate e delle risorse utilizzate nelle varie fasi di sviluppo.

3.1 *Ambiente di sviluppo*

Android Studio [4] è l'ambiente di sviluppo integrato (IDE) ufficiale per lo sviluppo di app Android. È sviluppato da Google ed è ampiamente utilizzato dagli sviluppatori Android per creare, testare ed eseguire il debug di applicazioni Android. Android Studio offre un set completo di strumenti e funzionalità che semplificano il processo di sviluppo delle app.
Ha un'interfaccia molto semplice (Figura 3.1) e facilmente comprensibile.

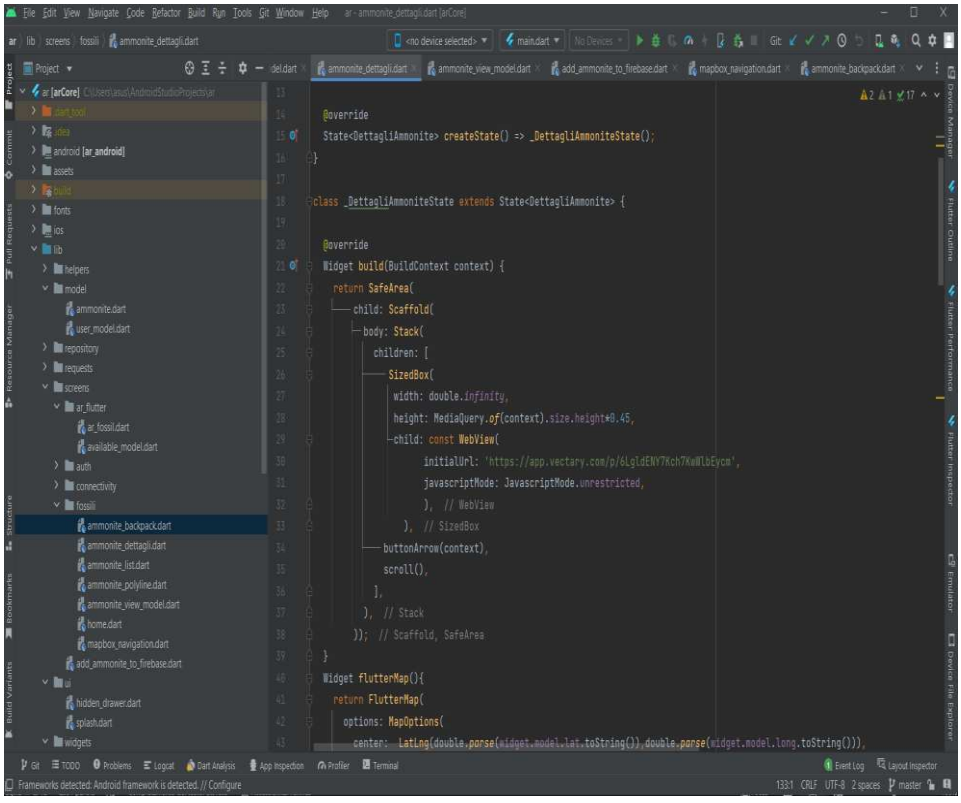


Figura 3.1

Ecco alcuni aspetti e caratteristiche chiave di Android Studio:

- **Interfaccia utente:** Android Studio ha un'interfaccia intuitiva con vari pannelli e finestre per la codifica, la progettazione di layout e l'anteprima delle schermate delle app.

- **Editor di codice:** offre un potente editor di codice con funzionalità come il completamento del codice, la navigazione del codice e l'evidenziazione della sintassi. Gli sviluppatori possono scrivere codice in Java, Kotlin [6] o altri linguaggi supportati.

- **Editor di layout:** Android Studio include un editor di layout visivo che consente agli sviluppatori di progettare interfacce di app utilizzando un generatore di interfacce drag-and-drop. È possibile visualizzare un'anteprima dell'aspetto dell' app su dispositivi e dimensioni dello schermo diversi.

- **Emulatore:** Android Studio fornisce un emulatore integrato che consente di testare l'app su diverse configurazioni di dispositivi Android, tra cui varie versioni di Android e dimensioni dello schermo.

- **Strumenti di debug:** offre strumenti di debug affidabili per identificare e risolvere i problemi nel codice. È possibile impostare punti di interruzione, esaminare le variabili e scorrere il codice per individuare i bug.

- **Profilazione delle prestazioni:** Android Studio include strumenti per la profilazione delle prestazioni dell'app, aiutando l'utente a identificare i colli di bottiglia e ottimizzare il codice per migliorare le prestazioni.

- **Integrazione del controllo della versione:** supporta l'integrazione con i sistemi di controllo della versione più diffusi come Git [3],semplificando la gestione del codice sorgente del progetto.

- **Gradle Build System:** Android Studio utilizza il sistema di compilazione Gradle, che consente di definire e personalizzare il processo di compilazione e le dipendenze dell'app.

- **Firma delle app:** Android Studio semplifica il processo di firma dell'app per la distribuzione sul Google Play Store e su altri app store.

- **Modelli e procedure guidate:** fornisce modelli e procedure guidate per iniziare le attività comuni di sviluppo di app, come la creazione di una nuova attività, la configurazione della navigazione o l'aggiunta di un database.

- **Android Jetpack:** Android Studio si integra con Android Jetpack, un insieme di librerie, strumenti e best practice per aiutare gli sviluppatori a creare più facilmente app Android di alta qualità.

- **Integrazione con Google Play Console:** permette la pubblicazione delle app su Google Play Store da Android Studio e monitorare le prestazioni della tua app utilizzando Google Play Console.

3.2 *Il framework Flutter e Dart*

Flutter [5] è un toolkit di sviluppo software open source UI (User Interface) creato da Google. Viene utilizzato per creare applicazioni compilate in modo nativo per dispositivi mobili, Web e desktop da un'unica code base. Flutter è stato rilasciato per la prima volta nel 2017 e ha guadagnato popolarità tra gli sviluppatori per la sua facilità d'uso, lo sviluppo rapido e la capacità di creare app di alta qualità e visivamente accattivanti.

Le caratteristiche e gli aspetti principali di Flutter includono:

- **Base di codice singola**: con Flutter, una singola base di codice che viene eseguita su più piattaforme, tra cui Android, iOS, Web e desktop. Ciò si ottiene utilizzando gli stessi widget e framework per ogni piattaforma, riducendo la necessità di codice specifico della piattaforma.

- **Widget:** Flutter fornisce un ricco set di widget personalizzabili che consentono di creare interfacce utente complesse e belle. Questi widget sono progettati per apparire e sentirsi nativi su ogni piattaforma.

- **Hot Reload:** Una delle caratteristiche distintive di Flutter è la sua capacità "Hot Reload", che consente agli sviluppatori di vedere gli effetti delle modifiche al codice quasi istantaneamente senza dover ricostruire l'intera app. Ciò accelera lo sviluppo e il debug.

- **Prestazioni:** le app Flutter sono note per le loro elevate prestazioni grazie al fatto che sono compilate in codice ARM nativo, con conseguenti animazioni fluide e tempi di caricamento rapidi.

- **Supporto Web e desktop:** mentre Flutter era inizialmente focalizzato sullo sviluppo di app mobili, si è espanso per supportare applicazioni web e desktop, rendendolo una scelta versatile per lo sviluppo multipiattaforma.

- **Integrazione della piattaforma:** Flutter consente di accedere a funzionalità e API specifiche della piattaforma utilizzando plug-in. Ciò significa che è possibile integrare perfettamente funzionalità come l'accesso alla telecamera, i servizi di localizzazione e i sensori dei dispositivi.

- **Linguaggio di programmazione Dart**: Flutter utilizza il linguaggio di programmazione Dart, anch'esso sviluppato da Google. Dart è facile da imparare e offre funzionalità come la compilazione Just-in-Time (JIT) e Ahead-of-Time (AOT).

3.3 *Firebase*

Firebase [6] è una piattaforma completa per lo sviluppo di applicazioni mobili e web sviluppata da Google. Fornisce un'ampia gamma di strumenti e servizi per aiutare gli sviluppatori a creare applicazioni di alta qualità e ricche di funzionalità in modo rapido ed efficiente. I servizi Firebase sono progettati per gestire vari aspetti dello sviluppo di app, dall'autenticazione e dai database in tempo reale all'archiviazione cloud e all'apprendimento automatico.
L'interfaccia di Firebase è rappresentata in Figura 3.2.

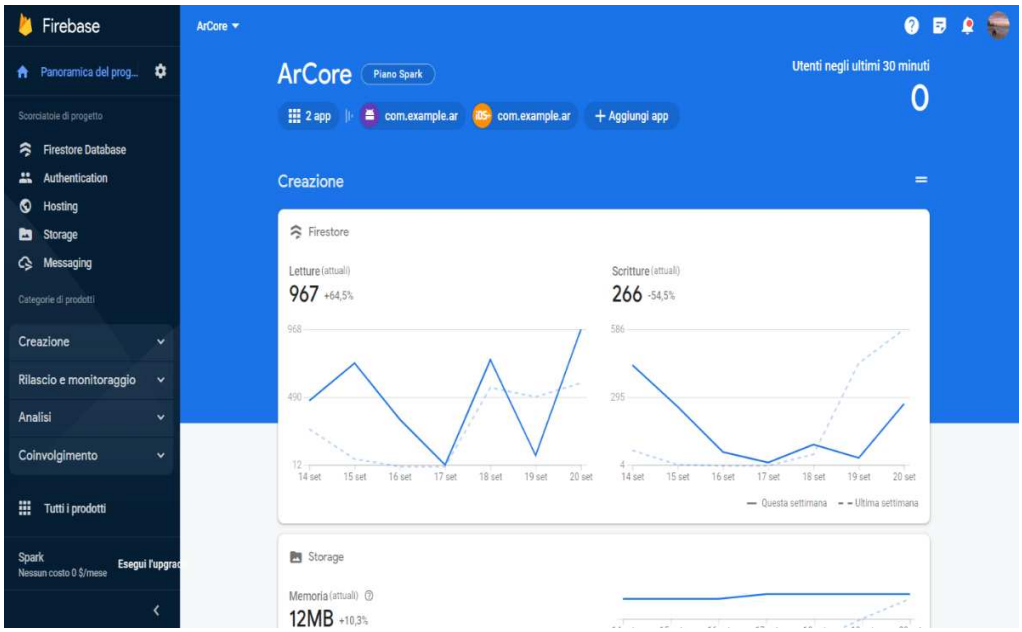


Figura 3.2

Ecco alcuni componenti che sono stati utilizzati nell'app *'AmmoniteExplorer*:

- **Autenticazione:** Firebase offre solidi servizi di autenticazione, tra cui autenticazione e-mail/password, autenticazione social (ad esempio, Google, Facebook, Twitter) e autenticazione del numero di telefono. Ciò semplifica la gestione degli utenti e consente di implementare flussi di accesso e registrazione sicuri degli utenti nell'app.
  È stato integrato nell'applicazione con la funzione di autentificare e registrare i vari utenti**.**

- **Cloud Storage:** Firebase Cloud Storage fornisce un cloud storage sicuro e scalabile per contenuti generati dagli utenti come immagini, video e file. Si integra perfettamente con Firebase Authentication per il controllo degli accessi.
  Questo cloud è stato utilizzato per lo storage delle immagini delle ammoniti (in formato gif, jpg).

- **Firestore:** Firestore è il database NoSQL più avanzato di Firebase che offre funzionalità aggiuntive come scalabilità, potenti funzionalità di query e sincronizzazione dei dati offline. È adatto per una vasta gamma di applicazioni, dalle app mobili alle app Web.
  Tale servizio è stato utilizzato per memorizzare i dati relativi:

  1. Cloud Anchor Api
  2. Utenti
  3. Ammoniti
  4. Modelli glb [15]

3.4 *Package e Plugin*

Nel contesto dello sviluppo di Flutter, sia "package" [9] che "plugin" si riferiscono a librerie o moduli esterni che estendono le funzionalità dell'applicazione Flutter. Tuttavia, ci sono alcune differenze nel modo in cui vengono utilizzati questi termini:

Package Flutter:

- **Descrizione:** i pacchetti Flutter sono librerie o moduli Dart che è possibile includere nel progetto Flutter per aggiungere caratteristiche o funzionalità specifiche. Questi pacchetti sono in genere scritti in Dart e sono ospitati nel Dart Package Repository (pub.dev).

- **Utilizzo:** per utilizzare un pacchetto Flutter nel progetto, specificarlo come dipendenza nel file pubspec.yaml [8] del progetto, nella sezione dipendenze.

Plugin Flutter:

- **Descrizione:** I plug-in Flutter sono moduli esterni che consentono di accedere alle funzionalità native della piattaforma (Android e iOS) dall'app Flutter.
  Questi plugin sono in genere scritti in linguaggi specifici della piattaforma come Kotlin [6] (per Android) e Swift [7] (per iOS).

- **Utilizzo:** Per utilizzare un plug-in Flutter, lo si aggiunge come dipendenza nel file pubspec.yaml del progetto, proprio come con i package Flutter.

In sintesi, si usano i pacchetti Flutter per le funzionalità basate su Dart e i plug-in Flutter per accedere alle funzionalità e alle API native della piattaforma nell' app Flutter. Entrambi vengono gestiti utilizzando il file pubspec.yaml e lo strumento da riga di comando flutter pub.

3.4.1 Pubspec.yaml

Il file pubspec.yaml è un file di configurazione utilizzato nei progetti Dart e Flutter. Svolge un ruolo centrale nella definizione delle dipendenze, dei metadati e di altre impostazioni per il progetto basato su Dart.

Ecco una panoramica del contenuto del file pubspec.yaml e del suo significato:

- **Metadati del progetto:** Include metadati sul progetto, ad esempio nome, descrizione, versione e informazioni sull'autore (Figura 3.3).

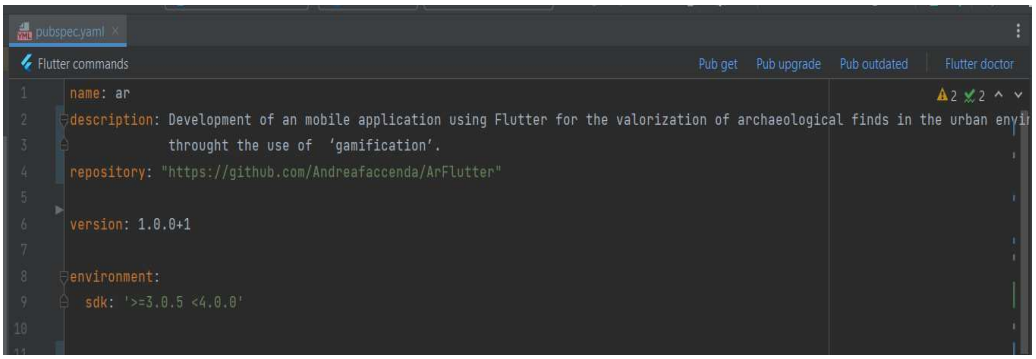


Figura 3.3

- **Dipendenze di sviluppo:** oltre alle dipendenze regolari, è anche possibile specificare dipendenze di solo sviluppo nella sezione dev_dependencies (Figura 3.4).

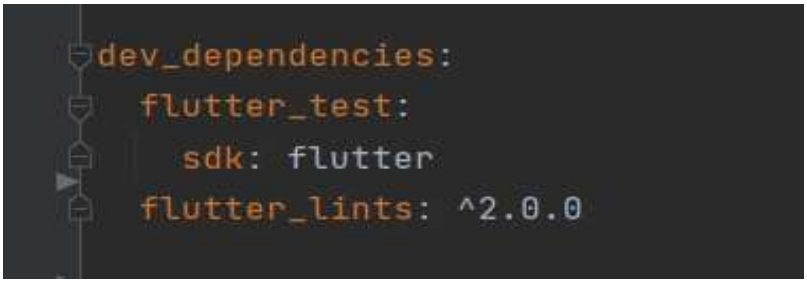


Figura3.4

- **Gestione delle risorse:** puoi specificare risorse come immagini, font e file di configurazione che devono essere raggruppati con la tua app nella sezione flutter (Figura 3.5).

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

  assets:
    - assets/icon/
    - assets/image/
    - assets/config/

  fonts:
    - family: Oswald
      fonts:
        - asset: fonts/Oswald-VariableFont_wght.ttf

    - family: PlayfairDisplay
      fonts:
        - asset: fonts/PlayfairDisplay-Italic-VariableFont_wght.ttf
        - asset: fonts/PlayfairDisplay-VariableFont_wght.ttf
```

Figura 3.5

- **Dipendenze di sviluppo:** uno degli scopi principali del file pubspec. yaml consiste nell'elencare le dipendenze su cui si basa il progetto. Queste dipendenze possono essere pacchetti Dart o Flutter e sono specificate nella sezione dipendenze**.**

3.4.2 Dipendenze dichiarate nel pubspec. yaml

Visto che abbiamo citato le dipendenze di sviluppo nell'applicazione *'Ammonite Explorer'* sono stati utilizzati i seguenti package e plugin [9]:

Dipendenze Firebase(Figura 3.6)

```yaml
#Firebase dependencies
google_sign_in: ^6.1.4
cloud_firestore: ^3.5.1
firebase_core: ^1.24.0
firebase_image: ^1.1.1
firebase_auth: ^3.11.2
firebase_storage: ^10.3.11
```

Figura 3.6

- Firebase Core for Flutter: un plugin per usare le API Firebase Core, che attiva la connessione a molteplici servizi di Firebase.
- Cloud Firestore Plugin for Flutter: un plugin Flutter per usare il database Cloud Firestore.
- Cloud Storage for Flutter: un plugin che consente di utilizzare lo Storage di Firebase in un progetto Flutter.

Dipendenze per la geolocalizzazione e mappe(Figura 3.7)

```
# Mapbox dependencies
flutter_map: any
flutter_map_location_marker: any
geolocator: ^9.0.2
geocoding: ^2.1.0
latlong2: ^0.8.0
location: ^5.0.3
flutter_mapbox_navigation:
  git: https://github.com/Andreafaccenda/flutter_mapbox_navigator
```

Figura 3.7

- **Flutter Map for Flutter**: Permette la creazione di un app basata su mappe in Flutter comporta l'integrazione di una libreria o di un servizio di mappatura, come Google Maps o Mapbox[10] ,e la creazione di interfacce utente per visualizzare e interagire con la mappa.

- **Geolocator**: Il pacchetto geolocator in Flutter è un pacchetto comunemente usato per accedere alla posizione del dispositivo (latitudine e longitudine) in un'app Flutter. Fornisce un modo semplice e diretto per recuperare la posizione dell'utente e monitorare le modifiche in tempo reale.

- **Geocoding**: Il pacchetto geocoding è comunemente usato come processo di conversione di indirizzi in coordinate geografiche o viceversa, utile soprattutto per posizionare indicatori su una mappa*(*definiti come marker) o per varie altre attività basate sulla posizione nelle applicazioni.

- **LatLong2:** Il pacchetto latlong fornisce utilità per lavorare con le coordinate geografiche (latitudine e longitudine).

- **Location:** Il pacchetto Location consente di accedere al GPS del dispositivo e ottenere coordinate di latitudine e longitudine, nonché altre informazioni relative alla posizione. (come Geolocator)

- **Flutter mapbox navigation:** Utile per implementare la navigazione Mapbox in un'app Flutter. Questo plugin consente di integrare la navigazione turn-by-turn con Mapbox nell'applicazione Flutter.

Dipendenze per il plugin Ar(Figura 3.8)

Figura 3.8

- **Ar flutter plugin:** Si tratta di un plug-in che consente di integrare funzionalità di realtà aumentata nelle applicazioni Flutter. Questi plugin in genere sfruttano i framework e le librerie AR [11] per abilitare esperienze di realtà aumentata su dispositivi Android [6] e iOS [7] (verrà spiegato in maniera dettagliata nel prossimo paragrafo*).*

- **Geoflutterfire:** È una libreria Flutter che fornisce funzionalità geospaziali per Firestore, che è il servizio di database cloud in tempo reale di Firebase. Consente di archiviare e interrogare i dati basati sulla posizione in Firestore, rendendolo utile per le applicazioni che richiedono funzionalità di geolocalizzazione, come il rilevamento della posizione, i recinti virtuali e le notifiche basate sulla posizione.

Altre Dipendenze(Figura 3.9)

```
#Other dependecies
cupertino_icons: ^1.0.2
provider: ^6.0.5
flutter_svg: ^2.0.7
hidden_drawer_menu: ^3.0.1
http: any
dio: ^4.0.4
webview_flutter: ^2.8.0
get: ^4.6.5
shared_preferences: ^2.2.0
flutter_dotenv: ^5.1.0
flutter_spinkit: ^5.1.0
simple_animation_progress_bar: ^1.6.0
google_fonts: ^4.0.4
connectivity_plus: ^4.0.2
```

Figura 3.9

- **Provider:** Si tratta di un pacchetto di gestione dello stato per Flutter che consente di gestire e condividere i dati tra i widget in modo efficiente.

- **Dio:** Libreria client HTTP per Dart e Flutter che semplifica l'esecuzione di richieste HTTP e l'utilizzo di API REST. Fornisce un modo comodo ed efficiente per eseguire operazioni HTTP come GET, POST, PUT, DELETE e altro. Dio supporta anche funzionalità come l'intercettazione di richieste / risposte, la cancellazione delle richieste e l'invio di dati in vari formati come JSON, FormData e altro.

- **Get:** Pacchetto di gestione dello stato e navigazione per Flutter che semplifica lo sviluppo di app fornendo una soluzione ad alte prestazioni per la gestione dello stato, delle rotte e delle dipendenze dell'app.

- **Shared Preferences**: Pacchetto Flutter che consente di archiviare e recuperare facilmente tipi di dati semplici, ad esempio numeri interi, stringhe, valori booleani e float, in modo persistente sul dispositivo. In 'AmmoniteExplorer' viene comunemente utilizzata per archiviare le informazioni dell'utente, le impostazioni e piccole quantità di dati che devono persistere tra l'avvio dell'app come la distanza dalla posizione corrente dell'utente dal reperto.

- **Flutter dotenv:** Pacchetto per Flutter che consente di caricare facilmente i valori di configurazione (come chiavi API, impostazioni specifiche dell'ambiente o altre informazioni riservate) da un file. env nell'applicazione Flutter.

3.5 Libreria AR Flutter Plugin

Il plugin Ar Flutter Plugin è stato già citato nel paragrafo precedente, ma poiché lo scopo centrale dell'applicazione mirava allo sviluppo di una app dove l'utente potesse interagire con i modelli 3D delle varie ammoniti presenti nel capoluogo marchigiano, tutto questo è stato possibile con l'ausilio di tale libreria.
Vediamola ora in maniera più dettagliata.

3.5.1 Panoramica generale

Ar Flutter Plugin [13] supporta ARKit [12] per iOS e ARCore [11] per dispositivi Android.
Si tratta della piattaforma di Google per creare esperienze di realtà aumentata, consentendo al tuo telefono di rilevare l'ambiente circostante, comprendere il mondo e interagire con le informazioni. Alcune API sono disponibili su Android e iOS per consentire esperienze AR condivise.
Utilizza tre funzionalità chiave per integrare i contenuti virtuali nel mondo reale visto attraverso la fotocamera del tuo telefono:

- Il monitoraggio del movimento consente al telefono di comprendere e monitorare la sua posizione rispetto al mondo.
- La comprensione ambientale consente al telefono di rilevare le dimensioni e la posizione di tutti i tipi di superfici: superfici orizzontali, verticali e inclinate, come il pavimento, un tavolino o le pareti.
- La stima della luce consente al telefono di stimare le condizioni attuali di illuminazione dell'ambiente.

Nel Capitolo 4 si parlerà più approfonditamente di Ar Flutter Plugin.

3.5.1 Architettura generale

L'architettura della libreria (Figura 3.7) prevede una combinazione di algoritmi software, componenti hardware e API per consentire esperienze AR coinvolgenti.

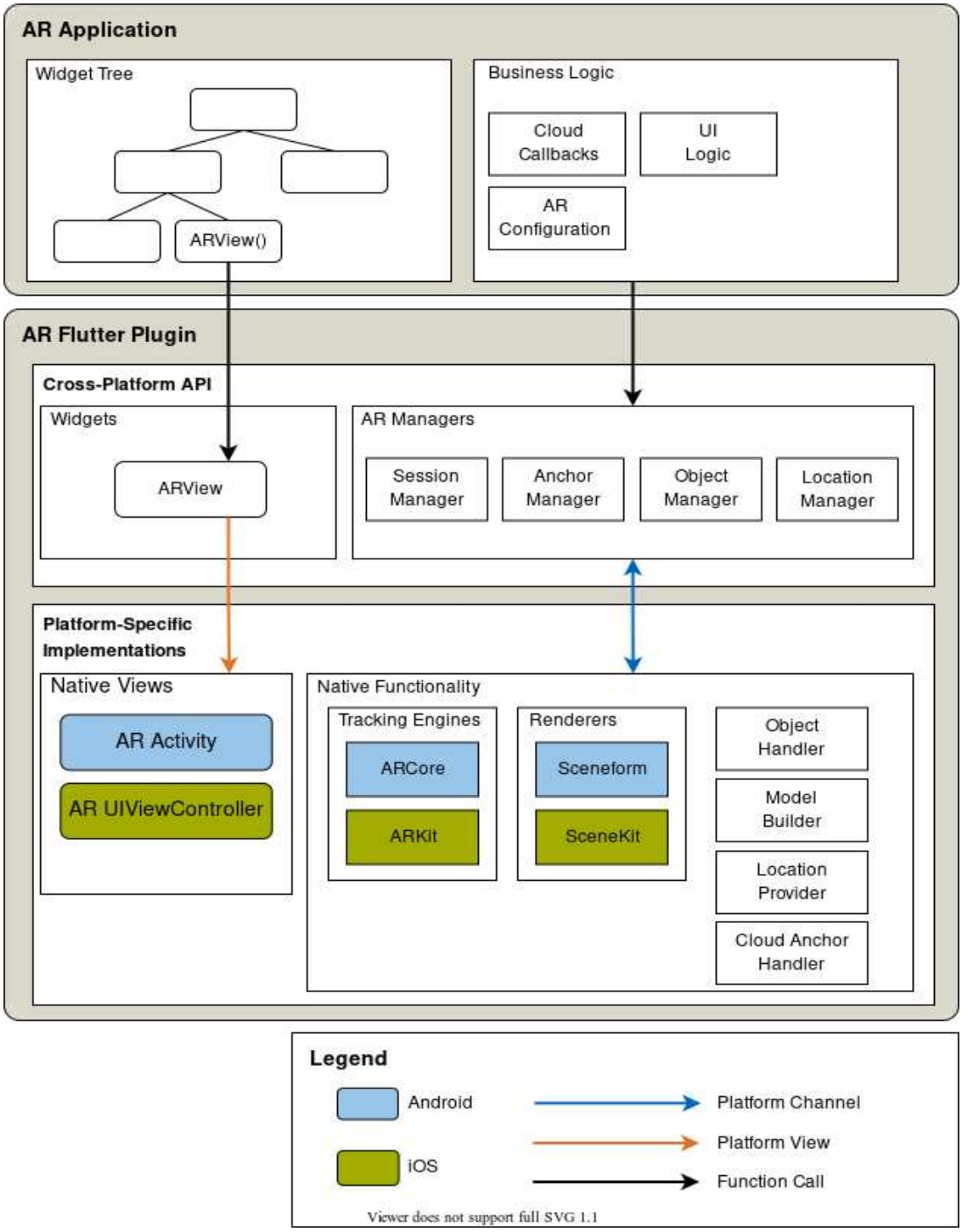


Figura 3.7

4. Sviluppo del software

In ingegneria del software è fondamentale conoscere e saper utilizzare gli strumenti che abbiamo a disposizione per realizzare i software.
In questo capitolo sono spiegate le caratteristiche generali dei programmi, delle varie librerie utilizzate e delle risorse utilizzate nelle varie fasi di sviluppo.
La progettazione del software è una delle fasi cruciali nello sviluppo del software ed è responsabile della definizione dell'architettura, della struttura e delle specifiche dettagliate del sistema software che verrà sviluppato.

4.1 Architettura

Architettura Flutter

Durante questa fase, viene creata un'architettura ad alto livello dell'applicazione. Si identificano i principali componenti del sistema, le interazioni tra di essi e la struttura generale dell'applicazione.
Come primo luogo però è di notevole importanza andare a considerare l'architettura nativa di Flutter [14] .
Flutter è progettato come un sistema estensibile e stratificato. Esiste come una serie di librerie indipendenti che dipendono ciascuna dal livello sottostante. Nessun livello ha accesso privilegiato al livello sottostante e ogni parte del livello del framework è progettata per essere facoltativa e sostituibile.
Il vantaggio di questo framework deriva poi dalla compatibilità con i vari sistemi operativi che è resa possibile grazie alla presenza appunto di questi strati software definiti come 'layer'.

Analizziamo opportunamente ogni singolo strato:

- L'embedder è scritto in un linguaggio appropriato per la piattaforma: attualmente Java e C ++ per Android, Objective-C / Objective-C ++ per iOS e macOS e C ++ per Windows e Linux. Utilizzando l'embedder, il codice Flutter può essere integrato in un'applicazione esistente come modulo oppure il codice può essere l'intero contenuto dell'applicazione. Flutter include un certo numero di embedder per piattaforme di destinazione comuni, ma esistono anche altri embedder.

- Flutter engine, è per lo più scritto in C ++ e supporta le primitive necessarie per supportare tutte le applicazioni Flutter. Il motore è responsabile della rasterizzazione delle scene composte ogni volta che è necessario dipingere un nuovo fotogramma. Fornisce l'implementazione di basso livello dell'API principale di Flutter, tra cui grafica (tramite Impeller su iOS e in arrivo su Android e Skia su altre piattaforme), layout del testo, file e I/O di rete, supporto per l'accessibilità.

- Lo strato del framework è la parte in cui la maggior parte degli sviluppatori può interagire con Flutter. Il Flutter engine viene esposto al framework Flutter tramite dart:ui, che esegue il wrapping del codice C++ sottostante nelle classi Dart. Questa libreria espone le primitive di livello più basso, ad esempio le classi per la guida di sottosistemi di input, grafica e rendering di testo.

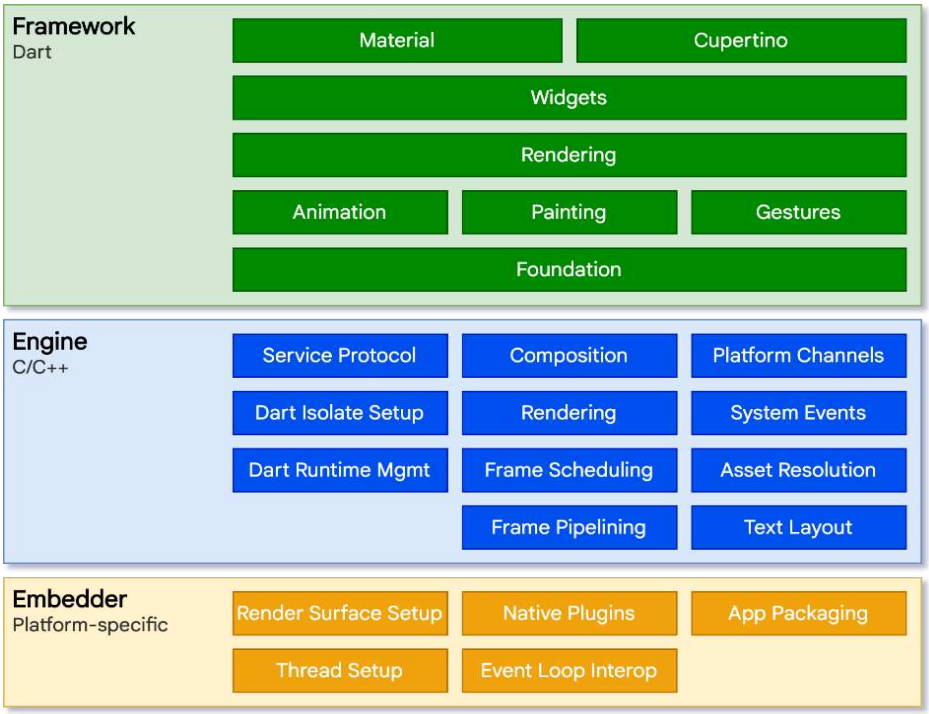


Figura 4.1

Per quanto riguarda l'architettura dell'app Ammoniti Explorer (Figura 4.2), 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.

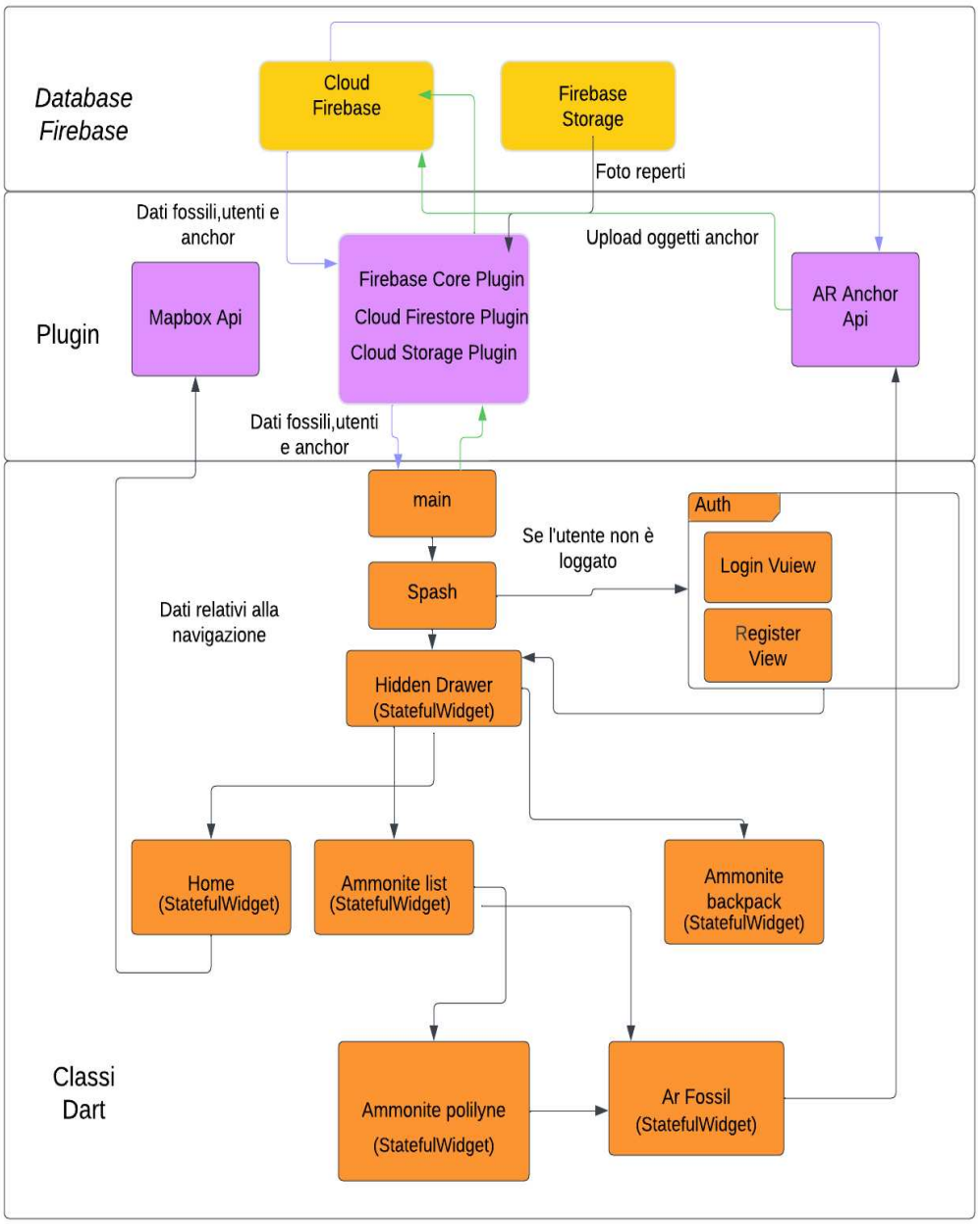


Figura 4.2

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.

Più nello specifico, queste sono le classi facenti parte del progetto Flutter:

- Splash(Figura 4.4): classe che si occupa di gestire il login automatico dell'utente nel caso in cui è già registrato a sistema e gestisce il collegamento con l'API di Mapbox prelevando i dati necessari.
  I dati fanno riferimento ai campi di distanza, durata e route di ogni singolo ammonite con la posizione attuale dell'utente.
  Il download dei dati viene effettuato una sola volta e vengono salvati in locale nelle shared preferences in modo da averli subito disponibili quando devono essere utilizzati.

- Login view(Figura 4.5): classe che gestisce l'operazione di autenticazione dell'utente,tramite Firebase Authentication che è un servizio fornito dalla piattaforma Firebase di Google. Offre un modo sicuro e semplice per gestire l'autenticazione degli utenti, incluse funzionalità come l'autenticazione e-mail / password, l'integrazione dell'accesso ai social media e altro ancora.

- Register view(Figura 4.7): classe che gestisce l'operazione di registrazione dell'utente, in Firebase Authentication.

- Hidden drawer view: classe che gestisce l'operazione di navigazione fra tutte le view principali dell'applicazione.
  Fornisce un menu di navigazione o opzioni a cui è possibile accedere scorrendo o toccando dal lato dello schermo.

- Home(Figura 4.8): definisce la schermata principale dell'applicazione. Questa classe utilizza il concetto di stato per modificare ciò che viene visualizzato come la posizione corrente dell'utente ogni volta che quest'ultima cambia.
  Questa classe contiene la logica per la rappresentazione di una mappa con tutti i vari marker delle ammoniti.

- Ammonite list(Figura 4.9): classe che contiene la logica per popolare una lista con tutte le ammoniti e le operazioni di filtraggio della lista.

- Ammonite dettagli(Figura 4.10): definisce la schermata in cui vengono rappresentate le informazioni e la foto di ogni singolo reperto. Questa schermata viene visualizzata in seguito al tocco dell'utente su un elemento della lista.

- Ammonite polyline(Figura 4.11):classe che contiene gestisce tutti i dati di navigazione salvati nelle shared preferences e li inserisce nei vari elementi di layout.
  Questa classe elabora tali informazioni per redigere sulla mappa una route che colleghi la posizione corrente dell'utente e quella del reperto.

- Ar flutter(Figura 4.12):classe che contiene tutta la logica relativa alla realtà aumentata.
  Questa classe è spiegata in maniera dettagliata nel paragrafo successivo.

- Ammonite backpack(Figura 4.17):classe che gestisce le operazioni di prelievo dei dati dal cloud rispettivamente la lista dei fossili catturati dell'utente momentaneamente loggato.
  Tali dati vengono poi inseriti all'interno di una lista che rappresenta lo zaino dell'utente.

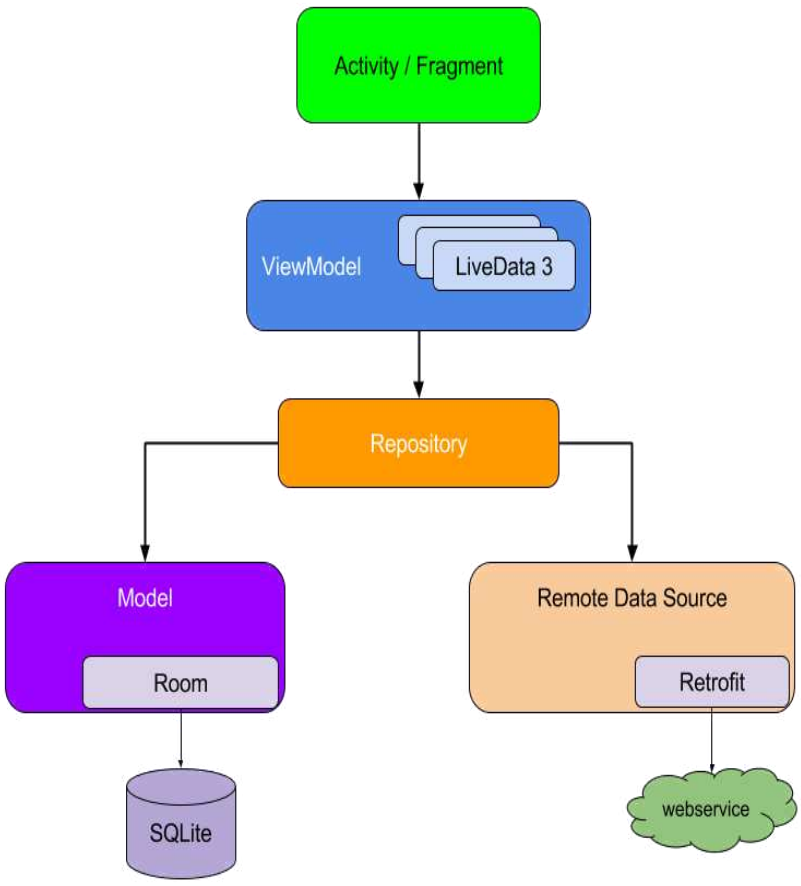
L'applicazione utilizza poi due fonti persistenti di dati:

• Cloud Firestore: è un database remoto messo a disposizione da Firebase, in cui è possibile salvare dati in documenti raccolti in delle collection (collezioni). Nel caso dell'applicazione Ammoniti di strada, il database è costituito da una sola collection in cui per ogni reperto da considerare è presente un documento.

• 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 ammoniti presenti in Firestore.

• API Mapbox e Cloud Anchor Api sono plugin che verranno discussi approfonditamente nei prossimi paragrafi.

Nell'applicazione è stato poi utilizzato il patter architetturale MVVM per avere una maggiore scalabilità.
MVVM [16] (Model-View-ViewModel) è un pattern architetturale per la separation of concerns, ovvero il principio di progettazione modulare secondo cui le diverse sezioni di un programma devono essere progettati come componenti separate.
Il Model rappresenta le classi in cui vengono gestiti i dati.
La View definisce il layout e gestisce il modo in cui le informazioni sono mostrate all'utente.
Il ViewModel gestisce gli eventi che arrivano dalla View e aggiorna lo stato di quest'ultima attraverso un LiveData che viene osservato dalla View. È inoltre disaccoppiato dal lifecycle della View.
Il repository pattern è un design pattern che separa il Data Layer, lo strato che gestisc
e i dati dell'applicazione, dallo UI Layer, composto da View e ViewModel.
Del Data Layer fa parte il repository, che viene usato dagli altri strati (in particolare il ViewModel) come tramite con il database. Più nello specifico, esso contiene la logica con cui vengono forniti i dati, ne gestisce i cambiamenti e i conflitti che possono generare.



Schema del database

Questa sezione è dedicata a descrivere il modello utilizzato per la rappresentazione dei dati relativi ai reperti. Come detto in precedenza, l'applicazione *'Ammoniti Explorer'* 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 database Firestore è stata creata una collection (raccolta), di nome 'ammonite', in cui sono raccolti tutti i documenti che riguardano i reperti. Per ogni ammonite infatti è presente un documento, che viene identificato dal nome del fossile, in modo da poterli distinguere facilmente. Tutti i documenti hanno la stessa struttura, ovvero sono caratterizzati dagli stessi attributi; un documento infatti è una serie di coppie chiave-valore, e le chiavi rimangono le stesse da un documento all'altro, mentre cambiano ovviamente i valori in base al reperto considerato. Un esempio di come è strutturato un documento è visibile nella Figura 4.3.

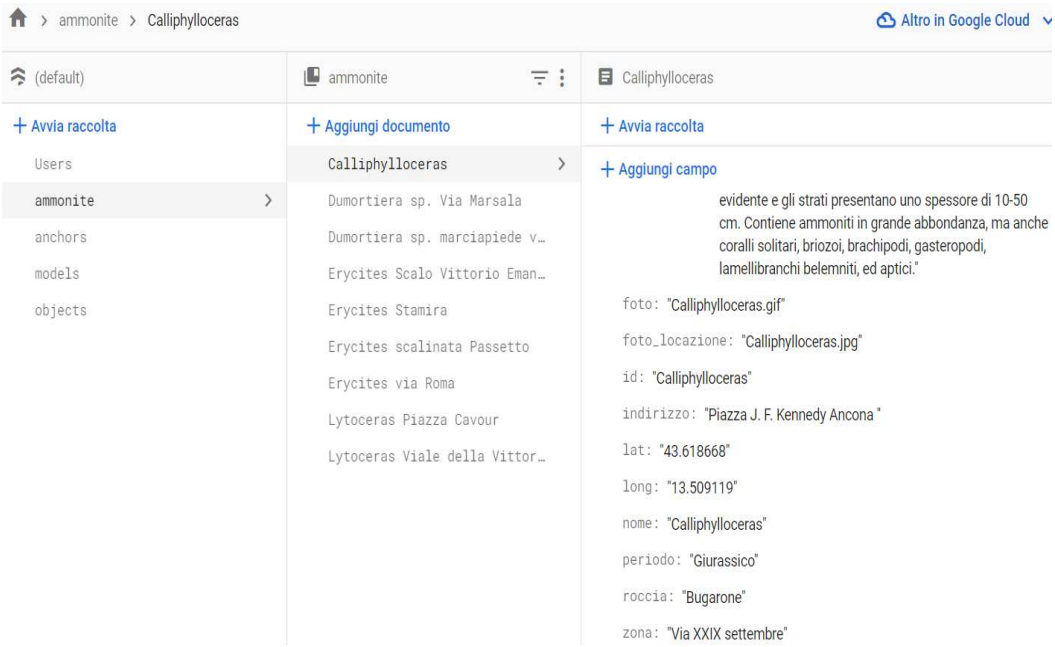


Figura 4.3

Ogni documento della collection ammonite è costituito da dodici campi:

• descrAmmonite (stringa): la descrizione dell'ammonite.
• foto_locazione (stringa): il nome del file che corrisponde alla foto di quel reperto, che servirà per poter risalire alla foto salvata nello Storage.
• lat (stringa): la latitudine del luogo in cui si trova il fossile.
• long (stringa): la longitudine del luogo in cui si trova il fossile.
• indirizzo (stringa): indirizzo dove è localizzato il reperto.
• zona (stringa): il nome del luogo in cui si trova il reperto (via, piazza, palazzo…).
• roccia (stringa): la tipologia di roccia dell'ammonite.
• descrRoccia (stringa): la descrizione della roccia.
• foto (stringa): il nome del file .gif che corrisponde alla foto di quel reperto, che servirà per poter risalire alla foto salvata nello Storage.
• id (stringa): identificativo del reperto.
• periodo (stringa): periodo storico dell'ammonite.

La collection 'Users' rappresentata dalla Figura 4.3 alla lista di utenti registrati all'applicazione.
Ogni documenti di questa collection è formato da 5 campi:
• e-mail (stringa): l'e-mail di accesso dell'utente.
• lista_fossili (array): array di id di ogni reperto che l'utente ha catturato.
• nome (stringa): nome dell'utente.
• password (stringa): password di accesso dell'utente.
• userId (stringa): identificativo di ogni utente.

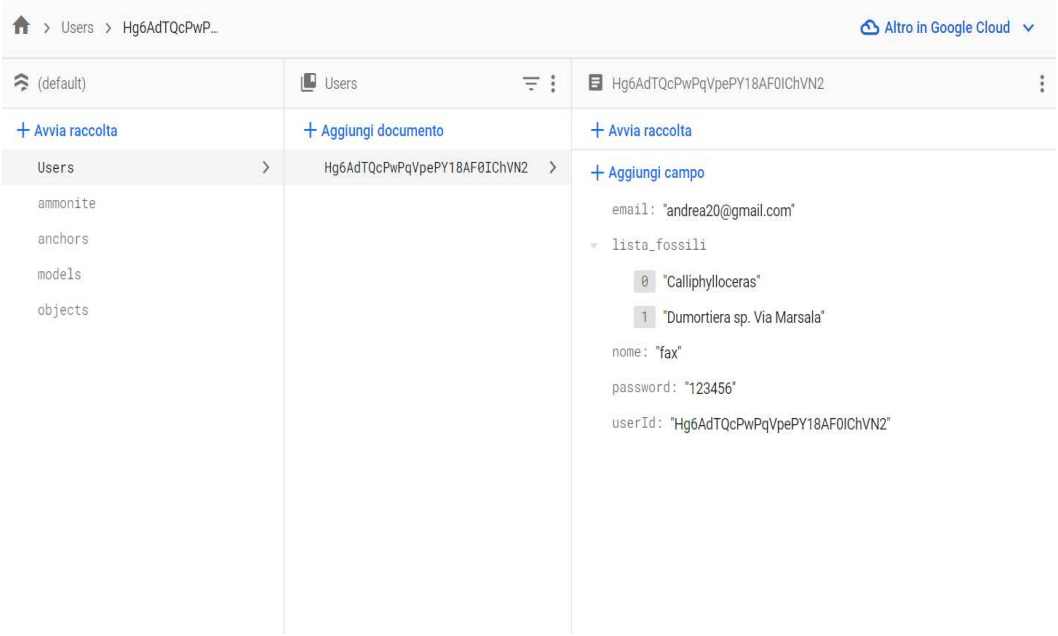


Figura 4.3

Le collection anchors, objects e model fanno riferimento all'API Cloud Anchor e verranno esaminate dettagliatamente nel paragrafo relativo alla AR.

4.2 *Pagine dell'app*

Ognuna delle immagini riportate di seguito è stata ottenuta catturando uno schermo di risoluzione 1080 x 2340 px.

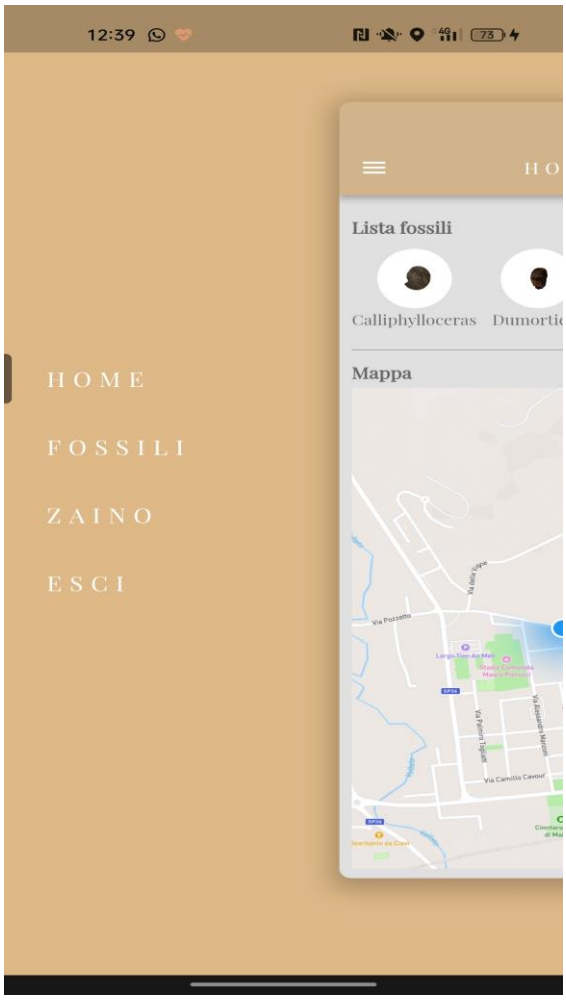
Questa è la prima interfaccia dell'applicazione di *'AmmoniteExplorer' (*Figura 4.2*)*. Se è già stato effettuato un accesso in precedenza e non è stato effettuato un log-out, verrà effettuato un log-in automatico senza che si debba inserire alcun dato.

Figura 4.4

La navigazione all'interno delle pagine è resa possibile grazie a un Hidden Drawer Menu, a cui è possibile accedere cliccando l'icona del menu posta in alto ad ogni view.

All'interno sono disposti i seguenti tasti:

- **Home** *(*Figura 4.6*)*
- **Fossili** *(*Figura 4.7*)*
- **Zaino** *(*Figura 4.15*)*
- **Esci** => Permette all'utente di uscire dall'applicazione effettuando il log out.



Se il software non riuscirà a recuperare un accesso precedente, avremo la schermata di login (Figura 4.5).
Da questa schermata è possibile, inoltre, registrarsi (Figura 4.6) o recuperare la propria password.

Figura 4.6

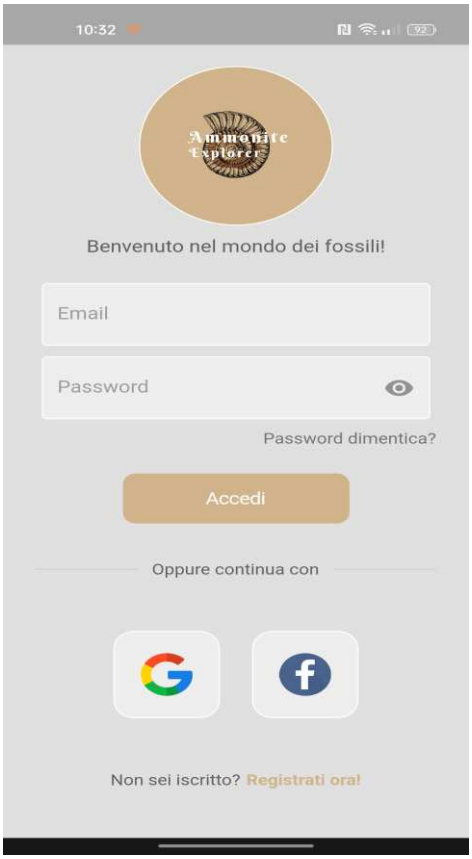


Figura 4.5

Se è la prima volta su *'AmmoniteExplorer'*, è possibile creare un account all'interno dell'app compilando i seguenti campi(Figura 4.7).
È sufficiente premere, nella schermata di Login *(*Figura 4.5*),* il bottone "Registrati ora", situato sotto il tasto "Accedi".

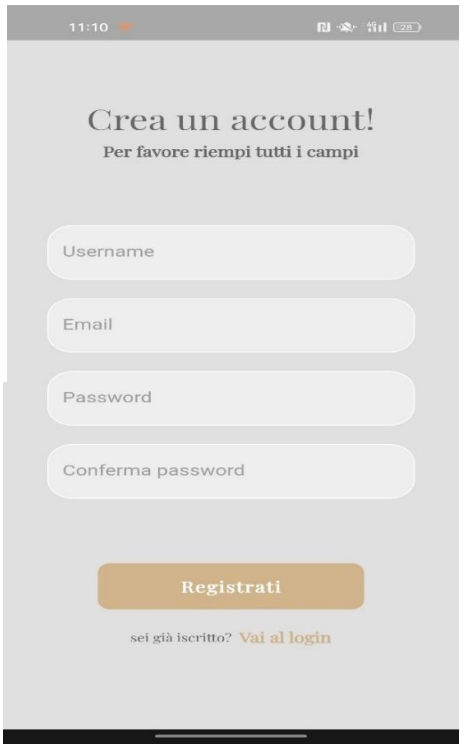


Figura 4.7

Questa è la home page dell'applicazione (Figura 4.6).
In alto è situato un'icona del menu (Hidden Menu Drawer), cliccandola è possibile accedere alle altre view della nostra app.
Nella sezione "Lista fossili" troviamo le varie miniature delle ammoniti e premendo è possibile vedere sulla mappa dove sono geolocalizzate.
In conclusione, un Floating Button rappresentato con l'icona del GPS ci consentirà di ritornare, qualora l'utente si sia spostato all'interno della mappa, alla posizione attuale.

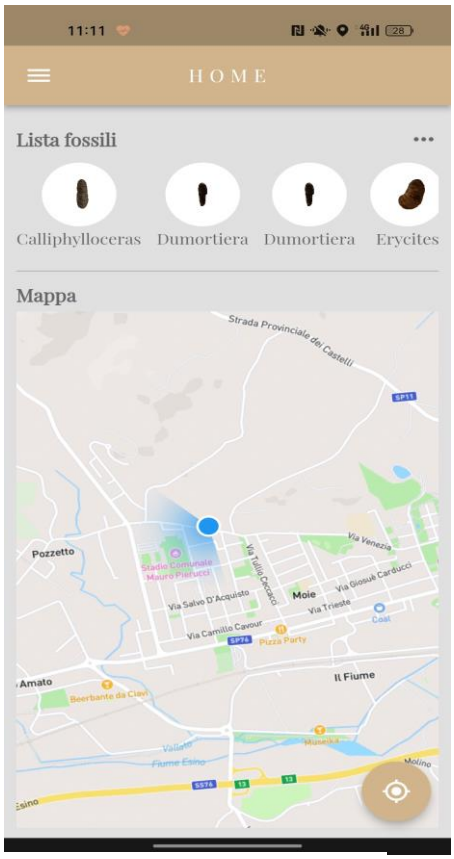


Figura 4.8

Questa è la schermata di catalogo delle ammoniti (Figura 4.9).
Da qua è possibile ricercare un particolare ammonite e inoltre scorrendo dal basso verso l'alto è possibile dare un'occhiata generale alla collezione.
Cliccando un'ammonite si procede con la view relativa ai dettagli dell'ammonite *(*Figura 4.10), mentre l'icona del piccone porta l'utente alla view di interazione con il modello 3D del fossile (Figura 4.12*)* e per ultima l'icona della mappa conduce ad una view di dettaglio della navigazione (Figura 4.11*)*.

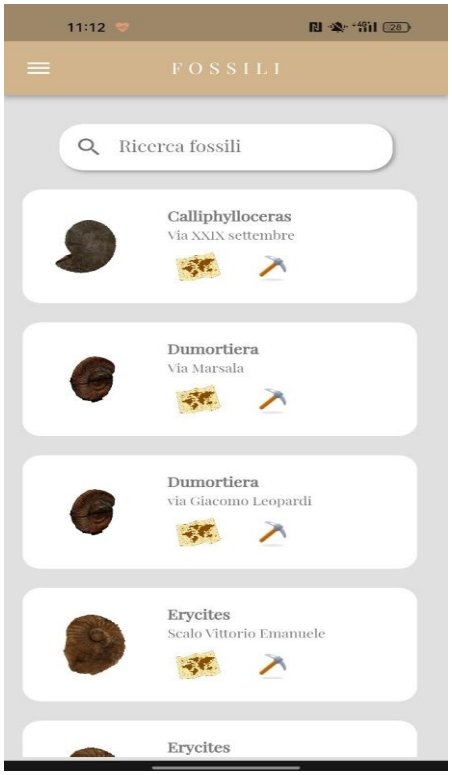


Figura 4.9

Questa è la schermata relativa ai dettagli di ogni singolo ammonite(Figura 4.10).
È possibile interagire con il modello 3D del fossile che è in formato. glb.
All'interno della view sono presenti tutte le specifiche di ogni singolo ammonite e una mappa per la geolocalizzazione del fossile.

Figura 4.10

Premendo sull'icona della mappa (Figura 4.9), si aprirà un'interfaccia utile a dare una
visione più specifica della posizione dell'ammonite.
Da qua verrà visualizzato l'identificativo del fossile, l'indirizzo di locazione, la distanza dalla posizione attuale dell'utente e il tempo di percorrenza di quest'ultima.
Infondo alla view sono presenti due *Floating Button:*

- Icona GPS => posizione attuale
- Icona Piccone => Schermata per la cattura dell'ammonite (Figura 4.12).

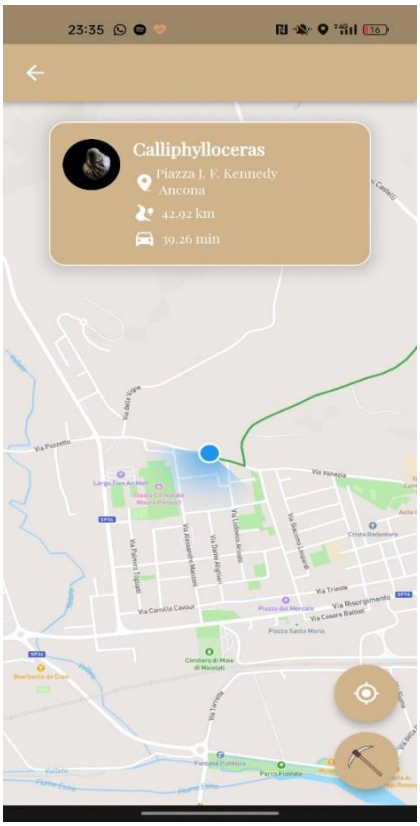


Figura 4.11

Questa è la schermata dedicata all'interazione dell'utente con il modello*(.glb)* dell'ammonite.

Figura 4.12

Figura 4.13

Le icone svolgono le seguenti funzioni:

- **Coordinate** => Il colore rosso dell'icona sta a significare che siamo molto distanti dalla locazione dell'ammonite la cui distanza è segnalata dalla TextBox posizionata sopra.
  Il colore verde implica che siamo molto vicino alla posizione del reperto e ne consegue che lo possiamo vedere come nella Figura 4.13.
- **Cattura** => Questo pulsante ci permette di catturare l'ammonite una volta che siamo molto vicini.
- **Navigatore** =>Permette la navigazione fino all'ammonite come si vede dalla Figura 4.14.
- **Cestino** = >Pulsante per rimuovere tutto
- **Locazione reperto** = > Foto del posto dove è localizzata l'ammonite. Figura 4.15
- **Scelta modello** => Permette all'amministratore la scelta del modello 3D da inserire nel cloud. Figura 4.16

Figura 4.14

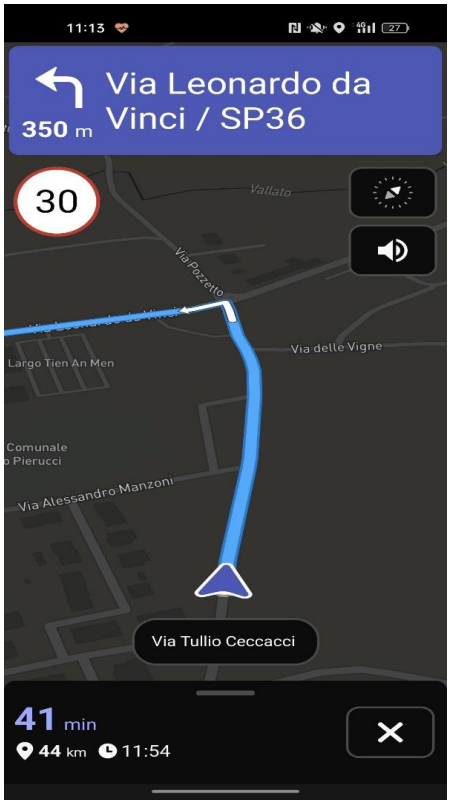


Figura 4.15

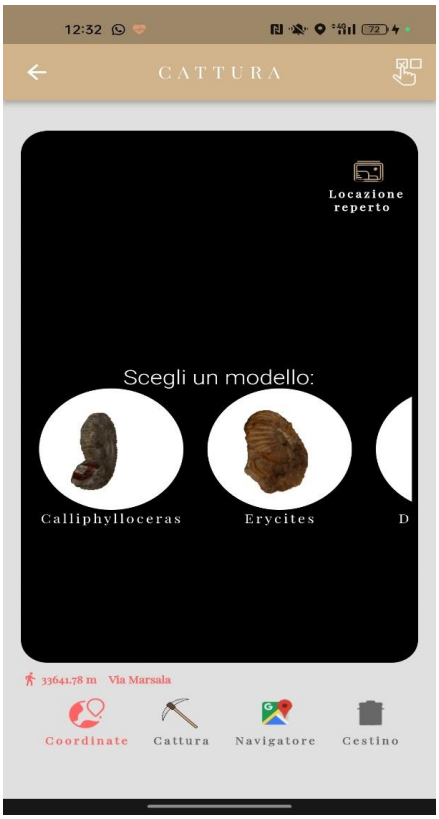


Figura 4.16

Questa è la schermata dello zaino dell'utente.
Da qua è possibile ricontrollare le ammoniti catturate (Figura 4.17), con i relativi dettagli come la roccia, la zona in cui è localizzata inoltre premendo l'icona della freccia è possibile andare alla schermata di dettaglio dell'ammonite *(*Figura 4.10*)*.
Nel momento in cui la lista dei reperti catturati dell'utente è vuota verrà visualizzata la schermata descritta dalla Figura 4.18.

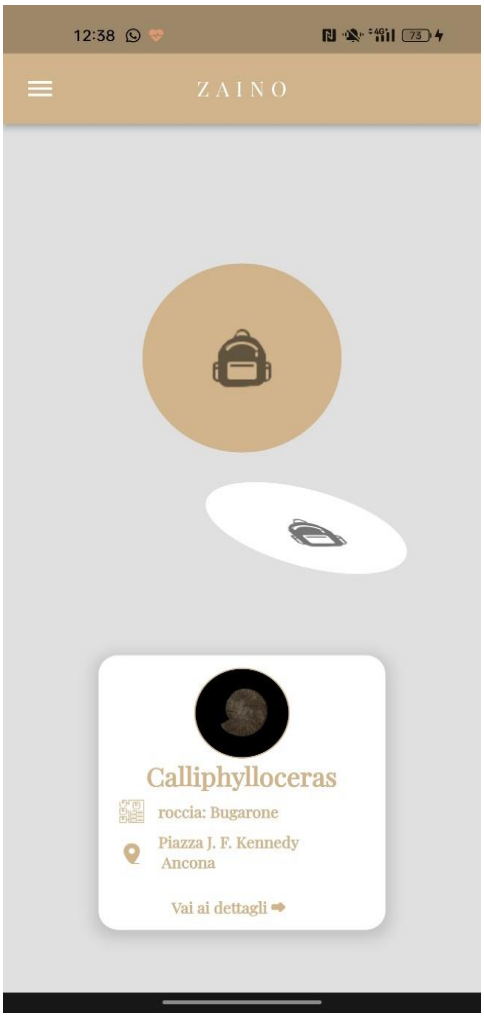


Figura 4.17

Figura 4.18

4.3 Implementazione in Flutter

4.3.1 Struttura del progetto

Questo progetto è stato sviluppato utilizzando Flutter che usa come linguaggio di programmazione Dart, che è sviluppato anch'esso da Google. Dart è un linguaggio fortemente tipizzato e orientato agli oggetti.
L'applicazione è stata sviluppata per essere eseguita su ambiente:

- Android versione 11 o superiori.
- iOS versione iOS 12.

Ogni progetto Flutter ha una struttura di base, la quale evolve a seconda dell'applicazione che si sta progettando.
In Figura 4.19 è riportata la directory di progetto dell'app che è stata discussa nei capitoli precedenti.
I vari package che compongono il progetto sono stati suddivisi nel seguente modo:

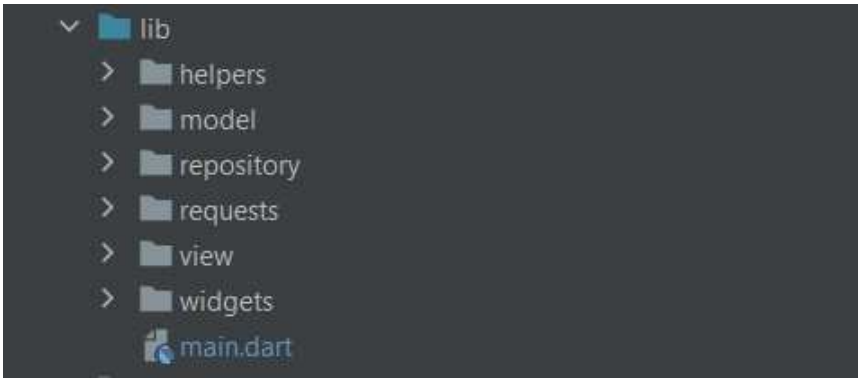


Figura 4.19

- Nel package model sono presenti le classi degli oggetti utilizzati per gestire i dati.
- Nel package repository sono presenti i repository utilizzati per comunicare con le sorgenti dati.
- Il package view contiene le user interface dove verranno visualizzate le informazioni.
- Il package helper contiene la classe per risolvere le dipendenze relative a AuthViewModel e AmmoniteViewModel inoltre sono presenti anche le classi per il fetching dei dati per la navigazione dell'API Mapbox.
- Nel package widget sono presenti i metodi per la personalizzazione dei Widget stessi.

4.4 Logica dell'applicazione

Tale sezione tratta della logica di implementazione dell'app, definendo per ogni singolo package tutta la logica presente all'interno di ogni classe.

4.4.1 La funzione main

```dart
late SharedPreferences sharedPreferences;
late List<Ammonite> ammoniti;
final viewmodel = AmmoniteViewModel();


void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
  sharedPreferences = await SharedPreferences.getInstance();
  await dotenv.load(fileName: "assets/config/.env");
  ammoniti = viewmodel.ammonite;
  DependencyInjection.init();
  runApp(const initializeFirebase());
}
```

Figura 4.20

Il file main.dart(Figura 4.20) in Flutter è il punto di ingresso principale dell'applicazione. È il file che viene eseguito quando si avvia l'app Flutter ed è responsabile di inizializzare l'applicazione tramite il metodo *runApp ()* e di configurare l'elemento radice, noto come Widget, che rappresenta l'intera UI dell'app.
In questa funzione si vanno ad inizializzare i servizi Firebase, le sharedPreferences, un modo di memorizzare piccole quantità di dati key-value in modo persistente, e inoltre di caricare la lista delle ammoniti.
Il codice *dotenv. load ('')* fa riferimento al caricamento di variabili di ambiente da un file. env nell'applicazione. Questa è una pratica comune nello sviluppo di software per gestire la configurazione e i dati sensibili in modo più sicuro e strutturato e ci consente rendere visibile a tutta l'applicazione l'API key di Mapbox.
Tramite il codice *DependencyInjection.init()* si inizializza un controller che monitora costantemente lo stato di connessione dell'applicazione.

4.4.2 Il package model

Gli oggetti presenti all'interno del sistema sono i seguenti:

• **UserModel**: rappresenta chiunque usufruisca dei servizi forniti dall'applicazione(Figura 4.21).

```
class UserModel {
  String? userId, nome, email, password;
  List<dynamic>? lista_fossili;

  UserModel({required this.userId, required this.nome, required this.email, required this.password,required this.lista_fossili});

  UserModel.fromJson(Map<dynamic, dynamic> map) {
    if (map == null) {
      return;
    }
    userId = map['userId'];
    nome = map['nome'];
    email = map['email'];
    password=map['password'];
    lista_fossili=map['lista_fossili'];
  }

  toJson() {
    return {
      'userId': userId,
      'nome': nome,
      'email': email,
      'password': password,
      'lista_fossili':lista_fossili,
    };
  }
}
```

Figura 4.21

Ad ogni user viene assegnato un id al momento della registrazione per identificarlo. Gli attributi e-mail e password rappresentano insieme le credenziali di accesso da utilizzare al momento del login.
L'attributo lista_fossili è una MutableList di oggetti Ammonite contenente gli ammoniti catturati dall'utente durante il suo utilizzo dell'app.

- **Ammonite**: rappresenta un singolo ammonite presente nel database(Figura 4.22).

```
class Ammonite {
  String? id,nome,foto_locazione,descrAmmonite,foto,roccia,descrRoccia,zona,lat,long,periodo,indirizzo;

  Ammonite({required this.id,required this.nome,required foto_locazione,required this.descrAmmonite,required this.foto,
            required this.roccia,required this.descrRoccia,required this.zona,
            required this.lat,required this.long,required this.periodo,required this.indirizzo});

  Ammonite.fromJson(Map<dynamic, dynamic> map) {
    if (map == null) {
      return;
    }
    id = map['id'];
    nome = map['nome'];
    foto_locazione = map['foto_locazione'];
    descrAmmonite = map['descrAmmonite'];
    foto = map['foto'];
    roccia = map['roccia'];
    descrRoccia = map['descrRoccia'];
    zona = map['zona'];
    lat = map['lat'];
    long = map['long'];
    periodo = map['periodo'];
    indirizzo = map['indirizzo'];
  }

  toJson() {
    return {
      'id' : id,
      'nome': nome,
```

Figura 4.22

Per questo oggetto, l'attributo id rappresenta l'identificativo dell'ammonite.
Gli attributi:

- Nome => rappresenta il nome del reperto.
- Foto_locazione => rappresenta l'immagine del posto in cui è geolocalizzato l'ammonite.
- descrAmmonite => si tratta di una descrizione dettagliata del reperto.
- foto => immagine dell'ammonite.
- roccia => tipo di roccia in cui è fossilizzata.
- descrRoccia => analisi del materiale
- zona => quartiere dove è geolocalizzato
- lat => latitudine
- long => longitudine
- periodo => era geologica di appartenenza
- indirizzo=> la via e la città di dove è posizionato il fossile.

4.4.3 Realizzazione pattern

Per la realizzazione del pattern repository nella nostra applicazione abbiamo utilizzato un Firestore Database come sorgente remota di dati.
Sono stati creati un repository e un ViewModel per ogni tipo di dato: userModel (utenti) e ammonite(Figura 4.23).
Il repository relativo agli utenti ottiene la sorgente dati dal database remoto, principalmente per l'autenticazione degli utenti che effettuano il login, mentre per il recupero delle impostazioni utente vengono utilizzate le shared Preferences.
Per questo scopo è stato creato un package repository contenente i repository relativi all'autenticazione degli utenti, e alle ammoniti.

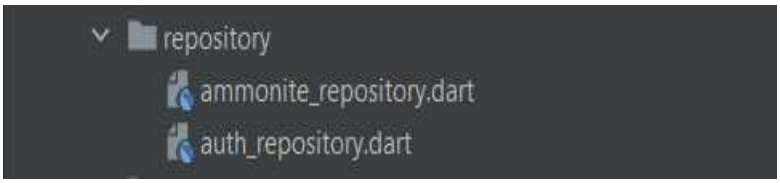


Figura 4.23

Nei repository sono quindi implementate le funzioni per interagire con il database remoto Firestore Database.
Queste implementazioni verranno poi utilizzate dai vari ViewModel per risolvere le richieste provenienti dal UI Layer.
Per lo scambio di dati con la sorgente dati ogni oggetto del package model contiene un metodo *fromJson*, per deserializzare i dati provenienti dal database, e un metodo *toJson*, per preparare i data ad essere elaborati dal database.
Per realizzare il dependency injection abbiamo utilizzato i metodi del micro-framework GetX, che permette di evitare la risoluzione manuale delle dipendenze dei vari componenti (*auth_view_model e ammoniti_view_model*).

```dart
import 'package:ar/helpers/ammonite_view_model.dart';
import 'package:get/get.dart';
import 'auth_view_model.dart';

class Binding extends Bindings {
  @override
  void dependencies() {
    Get.lazyPut(() => AuthViewModel());
    Get.lazyPut(() => AmmoniteViewModel());
  }
}
```

Figura 4.24

4.4.4 Gesture Detector

Il GestureDetector è un widget che abbiamo utilizzato per rilevare un'interazione dell'utente con l'interfaccia grafica. Fornisce il metodo *onTap* che permette di definire il comportamento dell'applicazione quando l'utente seleziona un elemento dell'interfaccia.

```
return GestureDetector(
  onTap: () {
    Get.to(() => DettagliAmmonite(model: lista[index]));
  },
```

Figura 4.25

GetX fornisce anche metodi di navigazione facili da usare e iniezione di dipendenze. È possibile utilizzare *Get.to (), Get.off ()* e metodi simili per la navigazione e *Get.put* () per l'inserimento delle dipendenze (Figura 4.25).

4.4.5 Shared Preferences

Tramite il plugin shared_preferences abbiamo memorizzato la sessione dell'utente che sta utilizzando l'applicazione e tutte le risposte API delle indicazioni stradali *(*route, duration, distance) per ogni ammonite.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="flutter.password">123456</string>
    <string name="flutter.fossile--8">{"geometry":{"coordinates":[[13.137809,43.504966],[13.137949,43.504986],[13.137
    <string name="flutter.fossile--7">{"geometry":{"coordinates":[[13.137809,43.504966],[13.137949,43.504986],[13.137
    <string name="flutter.name">fax</string>
    <string name="flutter.latitude">VGhpcyBpcyB0aGUgcHJlZml4IGZvciBEb3VibGUu43.5050174</string>
    <string name="flutter.longitude">VGhpcyBpcyB0aGUgcHJlZml4IGZvciBEb3VibGUu13.1377944</string>
    <string name="flutter.userId">Hg6AdTQcPwPqVpePY18AF0IChVN2</string>
    <string name="flutter.fossile--4">{"geometry":{"coordinates":[[13.137809,43.504966],[13.137949,43.504986],[13.137
    <string name="flutter.fossile--3">{"geometry":{"coordinates":[[13.137809,43.504966],[13.137949,43.504986],[13.137
    <string name="flutter.email">andrea20@gmail.com</string>
    <string name="flutter.fossile--6">{"geometry":{"coordinates":[[13.137809,43.504966],[13.137949,43.504986],[13.137
    <string name="flutter.fossile--5">{"geometry":{"coordinates":[[13.137809,43.504966],[13.137949,43.504986],[13.137
    <string name="flutter.fossile--0">{"geometry":{"coordinates":[[13.137809,43.504966],[13.137949,43.504986],[13.137
    <string name="flutter.fossile--2">{"geometry":{"coordinates":[[13.137809,43.504966],[13.137949,43.504986],[13.137
    <string name="flutter.fossile--1">{"geometry":{"coordinates":[[13.137809,43.504966],[13.137949,43.504986],[13.137
</map>
```

Figura 4.26

Tramite questo plugin vengono salvate le informazioni dell'utente su disco al momento del login, tramite il metodo *storeSession* (), e rimosse dopo aver effettuato il logout, tramite il metodo *removeSession* (). In particolare, queste informazioni,

se presenti, vengono utilizzate per effettuare il auto login al momento dell'avvio dell'applicazione. Mentre per il salvataggio delle informazioni di navigazione si è usufruito il metodo *initializeLocationAndSave ()* presente nella view Splash ().

4.4.6 Controllo connessione

Nel main, come abbiamo visto, è presente una funzione *DependencyInjection.init()* che inizializza un network controller(**_connectivity**) che periodicamente effettua un check sulla connessione(Figura 4.27).

```
@override
void onInit() {
  super.onInit();
  _connectivity.onConnectivityChanged.listen(_updateConnectionStatus);
}

void _updateConnectionStatus(ConnectivityResult connectivityResult) {

  if (connectivityResult == ConnectivityResult.none) {
    Get.rawSnackbar(
        messageText: const Text(
            'PER FAVORE CONNETTITI AD INTERNET',
            style: TextStyle(
                color: Colors.white,
                fontSize: 14,
              fontFamily: 'PlayfairDisplay'
            )  // TextStyle
        ),  // Text
        isDismissible: false,
        duration: const Duration(days: 1),
        backgroundColor: Colors.red[400]!,
        icon : const Icon(Icons.wifi_off, color: Colors.white, size: 35,),
        margin: EdgeInsets.zero,
        snackStyle: SnackStyle.GROUNDED
    );
  } else {
    if (Get.isSnackbarOpen) {
      Get.closeCurrentSnackbar();
    }
  }
```

Figura 4.27

Se la funzione rileva un problema sulla connessione internet verrà mostrato uno dialog che informerà l'utente del problema riscontrato.
In caso di assenza di connessione internet non saranno utilizzabili le interazioni con il database remoto e verranno quindi utilizzati solamente i dati in locale.

4.4.7 API Mapbox

Nella nostra applicazione si è deciso l'implementazione di una libreria alternativa a Google Maps che offrisse i medesimi servizi e prestazione.
Il plugin in questione è Mapbox una piattaforma popolare per la creazione di mappe personalizzate e applicazioni basate sulla posizione. Forniscono una gamma di servizi e strumenti per gli sviluppatori per integrare mappe e funzionalità geospaziali nelle loro applicazioni. Uno dei componenti chiave di Mapbox è la sua API, che consente agli sviluppatori di accedere e interagire con i loro servizi di mappatura e geospaziali in modo programmatico.
Ecco alcune delle principali API e servizi Mapbox che sono stati implementati nell'app:

- **Mapbox Maps API:** questa API consente di incorporare le mappe Mapbox nelle applicazioni Web o mobili. È possibile personalizzare l'aspetto delle mappe, aggiungere marcatori, etichette e vari.
- **Mapbox Geocoding API**: questa API fornisce servizi di geocodifica e geocodifica inversa, consentendo di convertire indirizzi o nomi di luoghi in coordinate geografiche (latitudine e longitudine) e viceversa. È utile per la ricerca e la navigazione basate sulla posizione.
- **Mapbox Directions API**: questa API offre servizi di routing e indicazioni stradali. Viene usato per calcolare le indicazioni stradali, pedonali o ciclabili tra due o più località. Fornisce istruzioni dettagliate e può essere integrato nelle applicazioni di navigazione.

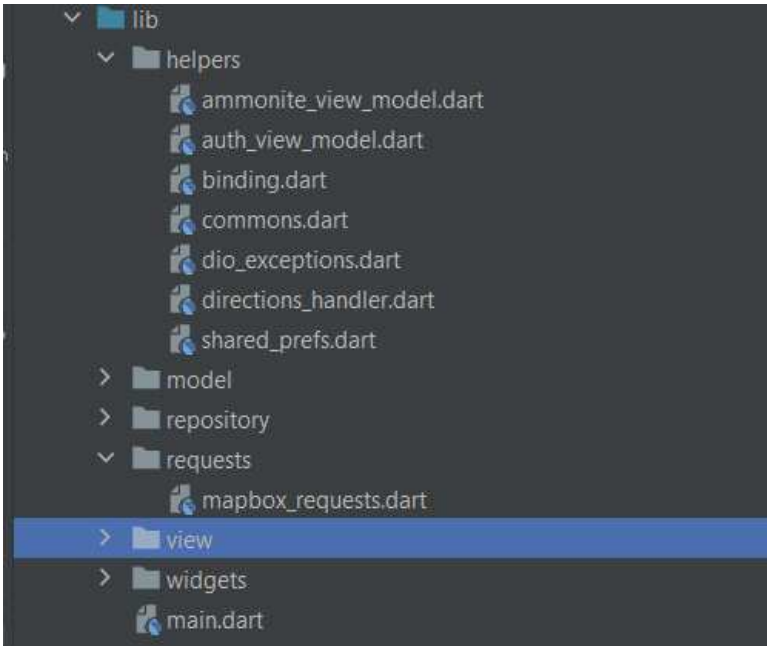


Figura 4.28

Nella Figura 4.28 si possono osservare i file. Dart che gestiscono le varie chiamate all'API di Mapbox.

La classe mapbox_request

Nello specifico la classe mapbox_request contiene il metodo per la chiamata all'API tramite il metodo *getDrivingRouteUsingMapbox* che richiede due parametri di tipo LatLong relativamente destinazione e partenza.
Per il recupero dei dati dalle API si ricorre all'utilizzazione della libreria Dio tramite il metodo "*jsonContentType*" il quale si riferisce in genere all'intestazione "Content-Type", che viene utilizzata per indicare il tipo o il formato dei dati inviati o ricevuti nel corpo del messaggio HTTP .
La figura sottostante fa riferimento alla classe mapbox_request.

```dart
import 'package:dio/dio.dart';
import 'package:flutter/cupertino.dart';
import 'package:flutter_dotenv/flutter_dotenv.dart';
import 'package:latlong2/latlong.dart';

import '../helpers/dio_exceptions.dart';

String baseUrl = 'https://api.mapbox.com/directions/v5/mapbox';
String accessToken = dotenv.env['MAPBOX_ACCESS_TOKEN']!;
String navType = 'driving';
String navTypew = 'walking';

Dio _dio = Dio();

Future getDrivingRouteUsingMapbox(LatLng source, LatLng destination) async {
  String url =
      '$baseUrl/$navType/${source.longitude},${source.latitude};${destination.longitude},${destination.latitude}'
      '?alternatives=true&continue_straight=true&geometries=geojson&language=en&overview=full&steps=true&access_token='
      '$accessToken';
  try {
    _dio.options.contentType = Headers.jsonContentType;
    final responseData = await _dio.get(url);
    return responseData.data;
  } catch (e) {
    final errorMessage = DioExceptions.fromDioError(e as DioError).toString();
    debugPrint(errorMessage);
  }
}
```

Per la gestione delle eccezioni che si possono verificare durante la chiamata all'API si ricorre alla classe DioExceptions, presente nella directory Helpers.

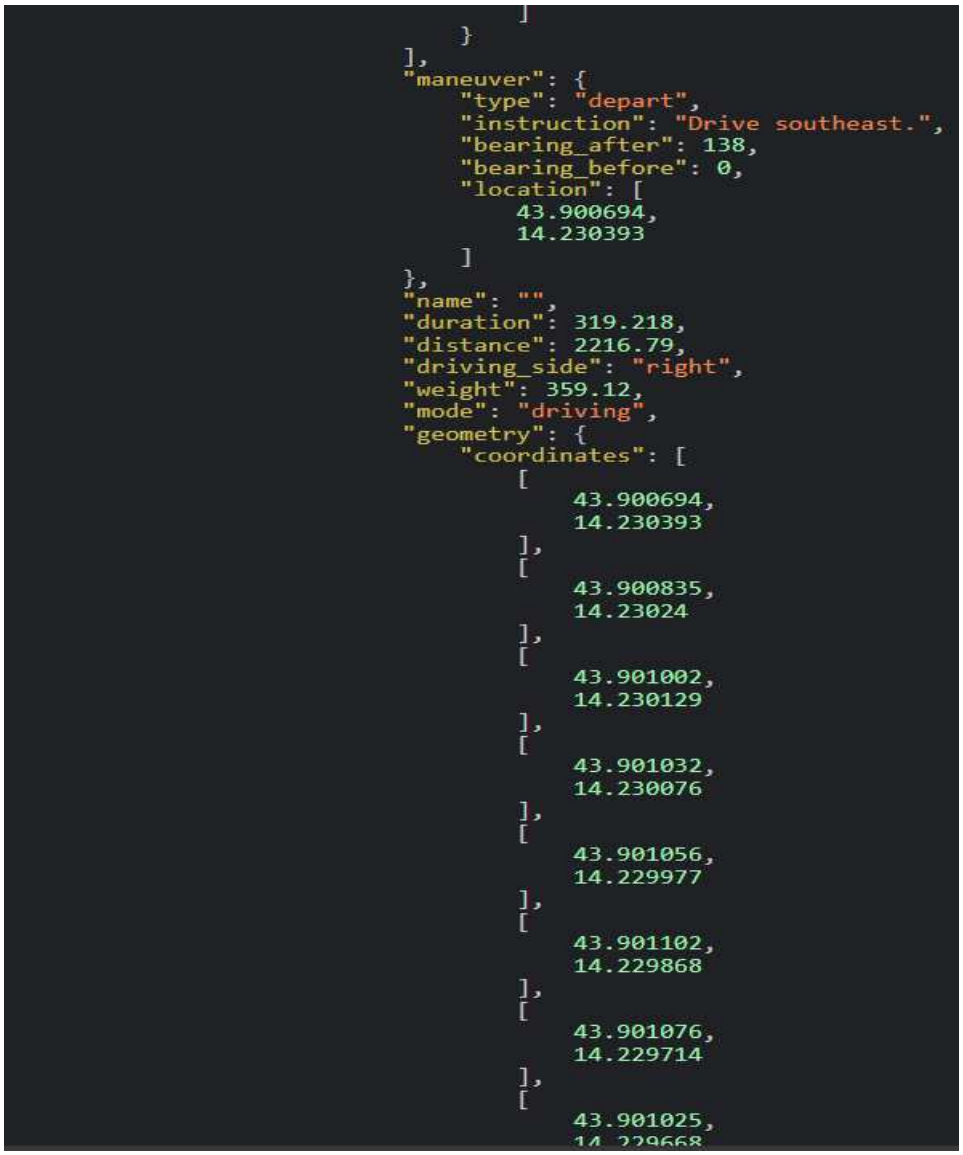
```dart
import 'package:dio/dio.dart';

class DioExceptions implements Exception {
  DioExceptions.fromDioError(DioError dioError) {
    switch (dioError.type) {
      case DioErrorType.cancel:
        message = "Request to API server was cancelled";
        break;
      case DioErrorType.connectTimeout:
        message = "Connection timeout with API server";
        break;
      case DioErrorType.other:
        message = "Connection to API server failed due to internet connection";
        break;
      case DioErrorType.receiveTimeout:
        message = "Receive timeout in connection with API server";
        break;
      case DioErrorType.response:
        message = _handleError(
            dioError.response!.statusCode!, dioError.response!.data);
        break;
      case DioErrorType.sendTimeout:
        message = "Send timeout in connection with API server";
        break;
      default:
        message = "Something went wrong";
        break;
    }
  }
```

Progettazione dei dati

I dati recuperati dalla chiamata all'API sono memorizzati in un file JSON(Figura 4.29 e Figura 4.30).

```
{
    "routes": [
        {
            "weight_name": "auto",
            "weight": 12905.281,
            "duration": 12580.136,
            "distance": 106675.125,
            "legs": [
                {
                    "via_waypoints": [],
                    "admins": [
                        {
                            "iso_3166_1_alpha3": "YEM",
                            "iso_3166_1": "YE"
                        },
                        {
                            "iso_3166_1_alpha3": "YEM",
                            "iso_3166_1": "YE"
                        },
                        {
                            "iso_3166_1_alpha3": "YEM",
                            "iso_3166_1": "YE"
                        },
                        {
                            "iso_3166_1_alpha3": "YEM",
                            "iso_3166_1": "YE"
                        }
                    ],
                    "weight": 12905.281,
                    "duration": 12580.136,
                    "steps": [
                        {
                            "intersections": [
                                {
                                    "bearings": [
                                        138
                                    ],
                                    "entry": [
                                        true
                                    ],
                                    "mapbox_streets_v8": {
                                        "class": "street"
                                    },
                                    "is_urban": false,
                                    "admin_index": 0,
                                    "out": 0,
                                    "geometry_index": 0,
                                    "location": [
                                        43.900694,
                                        14.230393
                                    ]
                                }
```


Figura 4.29



```
            ]
        }
    ],
    "maneuver": {
        "type": "depart",
        "instruction": "Drive southeast.",
        "bearing_after": 138,
        "bearing_before": 0,
        "location": [
            43.900694,
            14.230393
        ]
    },
    "name": "",
    "duration": 319.218,
    "distance": 2216.79,
    "driving_side": "right",
    "weight": 359.12,
    "mode": "driving",
    "geometry": {
        "coordinates": [
            [
                43.900694,
                14.230393
            ],
            [
                43.900835,
                14.23024
            ],
            [
                43.901002,
                14.230129
            ],
            [
                43.901032,
                14.230076
            ],
            [
                43.901056,
                14.229977
            ],
            [
                43.901102,
                14.229868
            ],
            [
                43.901076,
                14.229714
            ],
            [
                43.901025,
                14.229668
```


Figura 4.30

Si tratta di un array di routes, dove ogni oggetto steps, rappresenta un passo della navigazione e descritto da diversi attributi:

- ‘duration’ => durata in secondi della navigazione.
- ‘distance’ => distanza espressa in metri tra le due località.
- ‘mode’ => la modalità di navigazione.
- ‘coordinates’ => oggetto contenente a sua volta due attributi nei quali vengono memorizzate rispettivamente latitudine e longitudine della località. Tale array è utilizzato per creare la route di navigazione tra le due locazioni.
- ‘instruction’ => le istruzioni per la navigazione.

Parsing

Si occupa della richiesta http la funzione *getDirectionAPIRespone (*Figura 4.13*)*

```
Future<Map> getDirectionsAPIResponse(LatLng currentLatLng, int index) async {
  final response = await getDrivingRouteUsingMapbox(
      currentLatLng,
      LatLng(double.parse(ammoniti[index].lat.toString()),
          double.parse(ammoniti[index].long.toString())));
  Map geometry = response['routes'][0]['geometry'];
  num duration = response['routes'][0]['duration'];
  num distance = response['routes'][0]['distance'];
  Map modifiedResponse = {
    "geometry": geometry,
    "duration": duration,
    "distance": distance,
    "id": ammoniti[index].id,
  };
  return modifiedResponse;
}
```

Figura 4.31

Dopo aver specificato l'url da consultare, si può far partire la procedura HTTP grazie alla chiamata alla funzione *getDrivingRouteUsingMapbox* della classe mapbox_request. In questo modo viene estrapolato il testo contenuto nel JSON e assegnato alla variabile **response**. A questo punto **modifiedResponse** sarà una Map contenente le coppie key-value relative a tutte le informazioni per la navigazione come la geometria la distanza e la durata.
Il metodo *getDirectionAPIRespone* (Figura 4.31) viene chiamato per ogni singolo ammonite al momento della creazione dell'applicazione e i dati vengono poi salvati nelle shared preferences con il relativo indice di lista delle ammoniti e come possiamo vedere in Figura 4.26 tramite i metodi *getFromSharedPreferences* () i dati vengono prelevati e salvati negli opportuni attributi della classe.

```
calculateRoute() {
  for (int index = 0; index < ammoniti.length; index++) {
    String id = getIdFromSharedPrefs(index);
    if (widget.model.id == id) {
      distance = getDistanceFromSharedPrefs(index) / 1000;
      duration = getDurationFromSharedPrefs(index) / 60;
      Map geometry = getGeometryFromSharedPrefs((index));
      for (int i = 0; i < geometry['coordinates'].length; i++) {
        var coordinate = geometry['coordinates'][i];
        point.add(LatLng(double.parse(coordinate[1].toString()),
            double.parse(coordinate[0].toString())));
      }
    }
  }
}
```

Figura 4.32

4.4.8 AR flutter plugin

Nel capitolo 2 avevamo citato brevemente la libreria Ar descrivendone le funzionalità l'architettura e i servizi che venivano messi a disposizione.
La creazione di un'applicazione completa di realtà aumentata (AR) Flutter comporta diversi passaggi e può essere piuttosto estesa.
La combinazione di AR con Flutter consente agli sviluppatori di creare applicazioni AR multipiattaforma che sfruttano la facilità d'uso di Flutter e le capacità coinvolgenti dell'AR.
L'integrazione dell'AR nelle applicazioni Flutter offre il vantaggio dello sviluppo multipiattaforma, in particolare Flutter consente di sviluppare app AR che funzionano su dispositivi Android e iOS, riducendo i tempi e gli sforzi di sviluppo.
In questo capitolo verrà discusso in maniera approfondita l'implementazione della realtà aumentata nell'applicazione.

Inizializzazione e configurazione
Iniziamo col definire i vari oggetti AR che compongono la nostra applicazione:

- **ARView**: crea una vista telecamera dipendente dalla piattaforma utilizzando PlatformARView. Si tratta di un widget personalizzato creato per gestire il rendering AR e le interazioni all'interno dell'app.
  Per creare un'istanza di un ARView, il widget chiamante deve passare la funzione di callback *onARViewCreated(*Figura 4.33*)* a cui la funzione *createManagers* restituisce manager quali ARSessionManager e ARObjectManager.planeDetectionConfig viene passato al costruttore per determinare quali tipi di piani devono essere monitorati dai framework AR sottostanti (il valore predefinito è nessuno).

```
child: ARView(
    onARViewCreated: onARViewCreated,
    planeDetectionConfig: PlaneDetectionConfig.horizontal,),
```

Una volta che la ArView è istanziata viene chiamato il metodo onArViewCreated dove viene eseguita qualsiasi attività di inizializzazione e configurazione correlata alla realtà aumentata, come la configurazione dell'ambiente AR, la configurazione della telecamera o l'aggiunta di contenuto AR iniziale.

```
void onARViewCreated(
    ARSessionManager arSessionManager,
    ARObjectManager arObjectManager,
    ARAnchorManager arAnchorManager,
    ARLocationManager arLocationManager) {
  this.arSessionManager = arSessionManager;
  this.arObjectManager = arObjectManager;
  this.arAnchorManager = arAnchorManager;
  this.arLocationManager = arLocationManager;

  this.arSessionManager!.onInitialize(
    showFeaturePoints: false,
    showPlanes: true,
    customPlaneTexturePath: "assets/image/triangle.png",
    showWorldOrigin: false,
    showAnimatedGuide: false,
  );
  this.arObjectManager!.onInitialize();
  this.arAnchorManager!.initGoogleCloudAnchorMode();

  this.arSessionManager!.onPlaneOrPointTap = onPlaneOrPointTapped;
  this.arObjectManager!.onNodeTap = onNodeTapped;
  this.arAnchorManager!.onAnchorUploaded = onAnchorUploaded;
  this.arAnchorManager!.onAnchorDownloaded = onAnchorDownloaded;
```

Figura 4.33

- **ARSessionManager**: gestisce la configurazione della sessione, i parametri e gli eventi di ARView.
  Il metodo *onInitialize* viene utilizzato per configurare vari aspetti della visualizzazione e del comportamento della sessione AR.
  Viene fornita di seguito na spiegazione di ciascun parametro:

  - **showFeaturePoints**: false: questo parametro controlla se vengono visualizzati i feature point rilevati dal sistema AR. I punti caratteristici sono punti distintivi nell'ambiente del mondo reale che i sistemi AR utilizzano per il tracciamento. In questo caso, sono impostati per non essere visualizzati (false).

  - **showPlanes**: questo parametro controlla se vengono visualizzati i piani rilevati (superfici piane) nel mondo reale. È impostato su true, indicando che devono essere visualizzati i piani rilevati.

  - **customPlaneTexturePath**: "assets/image/triangle.png": questo parametro consente di specificare una texture personalizzata (immagine) da applicare ai piani rilevati.
  - **showWorldOrigin**: questo parametro determina se viene visualizzata l'origine del mondo il punto (0,0,0) (sistema di coordinate).

- **showAnimatedGuide**: false: questo parametro controlla se una guida animata viene visualizzata durante l'inizializzazione AR.

- **onPlaneOrPointTap**: si tratta di un campo del gestore di sessioni AR utilizzato per gestire gli eventi tap sui piani rilevati o sui punti di funzionalità nell'ambiente AR (Figura 4.34*)*.

- **ARObjectManager:** gestisce tutte le azioni relative ai nodi di un ARView.

  - **onNodeTapped:** si tratta di una funzione di callback che si sta assegnando alla proprietà onNodeTap di arObjectManager. Questa funzione viene chiamata quando un utente tocca o interagisce con un oggetto o nodo AR nella vista AR.

- **ARLocationManager:** consente di ottenere e aggiornare la posizione corrente del dispositivo.

- **ARAnchorManager**: Gestisce le funzionalità di ancoraggio come il gestore di download e il gestore di upload. Piu avanti nel capitolo affronteremo meglio il discorso relativo ad ARAnchorManager.

La funzione onPlaneOrPointTap

La funzione *onPlaneOrPointTapped* riceve un elenco di oggetti ARHitTestResult come parametro, che sono i risultati dei test hit eseguiti quando l'utente tocca la vista AR.

Cerca il primo risultato del test hit corrispondente a un piano rilevato (ARHitTestResultType.plane). Se viene trovato un tale risultato, crea un ARPlaneAnchor basato sulla trasformazione del risultato del test hit e imposta un valore di time-to-live (ttl) pari a 2 (che indica che l'ancoraggio deve esistere per 2 secondi).

Tenta di aggiungere l'ancoraggio alla sessione AR usando *arAnchorManager!. addAnchor* e controlla se l'aggiunta dell'ancoraggio è stata eseguita correttamente.

Crea quindi un ARNode che rappresenta un modello 3D (NodeType.webGLB) con determinate proprietà quali URI, scala, posizione e rotazione.

Tenta di aggiungere il nodo all'ancoraggio usando *arObjectManager! addNode (newNode, planeAnchor: newAnchor)* e controlla se l'aggiunta del nodo è stata eseguita correttamente.

Tale funzione è messa a disposizione solamente dall'amministratore di progetto perché è utilizzata per caricare nel database l'oggetto 3D.

```
Future<void> onPlaneOrPointTapped(
    List<ARHitTestResult> hitTestResults) async {
  var singleHitTestResult = hitTestResults.firstWhere(
        (hitTestResult) => hitTestResult.type == ARHitTestResultType.plane);
  if (singleHitTestResult != null) {
    var newAnchor = ARPlaneAnchor(
        transformation: singleHitTestResult.worldTransform, ttl: 2);
    bool? didAddAnchor = await this.arAnchorManager!.addAnchor(newAnchor);
    if (didAddAnchor!) {
      this.anchors.add(newAnchor);
      // Add note to anchor
      var newNode = ARNode(
          type: NodeType.webGLB,
          uri: this.selectedModel.uri,
          scale: VectorMath.Vector3(0.2, 0.2, 0.2),
          position: VectorMath.Vector3(0.0, 0.0, 0.0),
          rotation: VectorMath.Vector4(1.0, 0.0, 0.0, 0.0),
          data: {"onTapText": "I am a " + this.selectedModel.name});  // ARNode
      bool? didAddNodeToAnchor =
      await this.arObjectManager!.addNode(newNode, planeAnchor: newAnchor);
      if (didAddNodeToAnchor!) {
        this.nodes.add(newNode);
```

Figura 4.34

La funzione onNodeTapped

Il codice fornito in Figura 4.35 recupera un nodo AR specifico (un modello 3D) da un elenco di nodi in base al suo nome e quindi visualizza un messaggio utilizzando *arSessionManager! onError* () indicandone il nome del modello con cui l'utente ha interagito.

```
Future<void> onNodeTapped(List<String> nodeNames) async {
  var foregroundNode = nodes.firstWhere((element) => element.name == nodeNames.first);
  this.arSessionManager!.onError(foregroundNode.data!["onTapText"]);
}
```

Figura 4.35

Cloud anchor API

Cloud Anchor API consente lo sviluppo di applicazioni AR multiutente ancorando oggetti virtuali a posizioni specifiche nel mondo reale sincronizzate tra più dispositivi e utenti.
È stato fondamentale l'utilizzazione di questa API per avere ogni modello 3D di ammonite archiviato nel cloud, consentendo a più utenti di accedere e interagire con lo stesso contenuto AR ancorato alla stessa posizione del mondo reale nel tempo.
Gli ancoraggi Cloud consentono a un utente di posizionare un oggetto AR nell'ambiente fisico e a un altro per visualizzare lo stesso oggetto in un secondo momento. Nella nostra applicazione, è stato utilizzato Cloud Anchor per creare modelli 3D delle ammoniti posizionate nelle rispettive coordinate geografiche.

Come funzionano gli ancoraggi di Cloud
AR si connette all'endpoint cloud dell'API AR per ospitare e risolvere gli ancoraggi di Cloud, abilitando quindi queste esperienze condivise. Questa operazione richiede una connessione a Internet funzionante.
Ecco un quadro generale di come funzionano l'hosting e la risoluzione dei problemi:

1. L'amministratore crea un ancoraggio locale nel proprio ambiente tramite la funzione *onPlaneOrPointTap* (Figura 4.34).
2. L'ancoraggio è ospitato: AR carica i dati dell'ancoraggio locale nell'endpoint cloud dell'API AR che ne restituisce l'ID univoco tramite il metodo *onAnchorUploaded* che verrà spiegato in maniera più dettagliata nelle pagine successive.
3. L'app distribuisce questo ID univoco ad altri utenti.
4. L'ancoraggio è risolto: gli utenti i cui dispositivi hanno l'ID univoco possono ricreare lo stesso ancoraggio utilizzando l'API Cloud Anchor tramite la funzione *onAnchorDownloaded* () che parte in automatico quando l'utente si trova a una distanza di 5 metri dalla posizione dell'ammonite.

Come autorizzare l' app a chiamare l'API Anchor Cloud
Autorizzare l'app a chiamare l'API per ospitare e risolvere Cloud Anchors. Le app che ospitano e risolvono Cloud Anchor con un TTL (time-to-live) superiore a 1 giorno devono utilizzare l'autorizzazione senza chiave.

1. Abilitare Anchor Cloud API per un progetto Google Cloud Platform nuovo o esistente.
2. Crea un ID client OAuth per l' app Android in Google Cloud Console, utilizzando l'ID applicazione dell'app e firmando l'impronta SHA-1 del certificato. In questo modo l'app viene associata al progetto Google Cloud Platform.
3. Includere **com.google.android.gms: play-services-auth:16+** nelle dipendenze dell'app.

Gestione delle dipendenze

Dopo aver configurato l'API nel progetto per le operazioni di download e upload è richiesta l'installazione di GeoflutterFire nel pubspec.yaml.
Questa libreria è specificamente progettata per funzionare con Firebase, consente di tenere traccia della posizione in tempo reale di utenti o dispositivi e aggiornare le loro posizioni in Firebase Firestore.
Relativamente al codice che è stato scritto per le due funzionalità è supportata una versione del plugin 3.0.1,la quale fa riferimento come upper bound ad una versione di cloud_firebase 3.5.1(ottobre 2022).
Questo inizialmente comportava problematiche di conflitto di libreria poiché tutta l'applicazione faceva riferimento ad una versione del cloud che era molto più recente,cioè la versione 4.5.1.
Per ovviare a tale conflitto è stata adottata la soluzione di effettuare un dependency_overrides per indicare versioni specifiche dei pacchetti da utilizzare, ignorando le versioni definite nelle dipendenze del progetto. Ciò può essere utile quando si desidera garantire la compatibilità con determinate versioni di altri pacchetti o è necessario utilizzare temporaneamente una versione specifica per motivi di test o compatibilità.

```
dependency_overrides:
  firebase_core_platform_interface: 4.5.1
```

firebase_core_platform_interface: 4.5.1:
Questa riga specifica che il pacchetto firebase_core_platform_interface deve essere sovrascritto per utilizzare la versione 4.5.1. Ciò significa che, anche se le dipendenze del progetto specificano una versione diversa, Flutter utilizzerà la versione 4.5.1 per questo progetto specifico.

Parallelamente a questa problematica sorgeva anche questo errore:

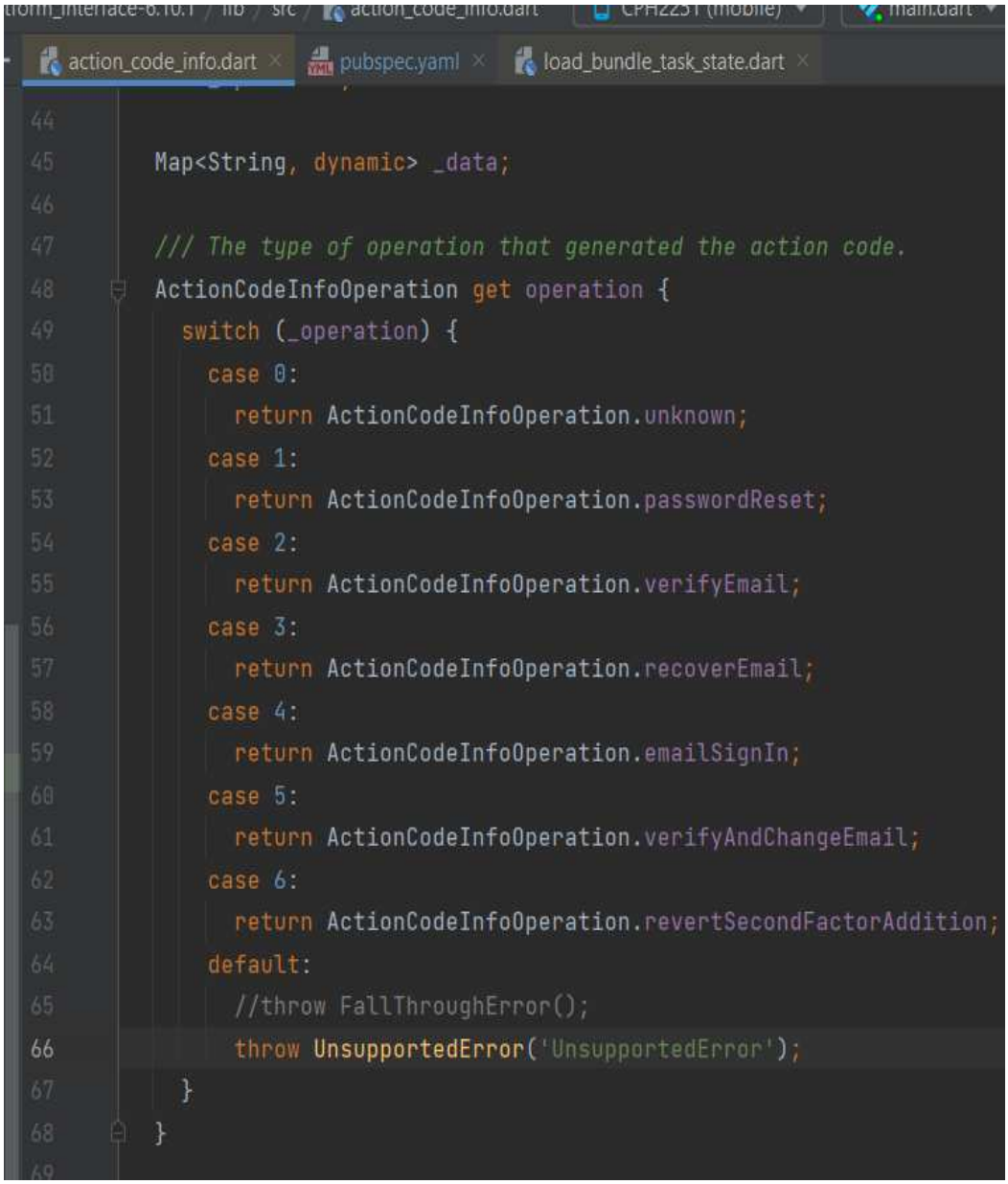
load_bundle_task_state.dart:13:13:
Error: Method not found: 'FallThroughError'.
throw FallThroughError();

Se si verifica questo errore, significa che si dispone di codice che sta tentando di utilizzare il comportamento fall-through in un'istruzione switch, ma l'analizzatore Dart o il runtime lo rileva come un errore perché ritiene che il fall-through non sia il comportamento previsto. Per risolvere questo errore, è necessario rimuovere il comportamento di caduta o indicare esplicitamente l'intenzione di utilizzarlo.

Il problema è stato risolto cancellando nella classe action_code_info il throw FallThroughError e inserendo il throw “UnsupportedError”, eccezione che viene utilizzata per segnalare che una particolare operazione o comportamento non è implementato o non può essere eseguito per un oggetto o una situazione specifici.

Errore generato: **Path action_code_info: firebase_auth_platform_interface-6.11.3/lib/src/action_code_info.dart**

La figura sotto fa vedere come è stato risolto il problema citato sopra.



Upload dei dati nel Cloud

Dopo aver attivato la funzionalità Cloud Anchor nell'app, l'inizializzazione o configurazione del sistema AR per utilizzare Google Cloud Anchors viene effettuata nel onARViewCreated tramite il metodo *initGoogleCloudAnchorMode* .

```
this.arAnchorManager!.initGoogleCloudAnchorMode();
```

Dopo che l'amministratore crea un ancoraggio locale (ArNode) nel proprio ambiente utilizzando la funzione *onPlaneOrPointTap* questo attiva ed esegue una funzione di callback *onAnchorUploaded ()* visibile in Figura 4.36.

```
onAnchorUploaded(ARAnchor anchor) {
  // Upload anchor information to firebase
  firebaseManager.uploadAnchor(anchor,
      currentLocation: this.arLocationManager!.currentLocation);
  // Upload child nodes to firebase
  if (anchor is ARPlaneAnchor) {
    anchor.childNodes.forEach((nodeName) => firebaseManager.uploadObject(
        nodes.firstWhere((element) => element.name == nodeName)));
  }
  setState(() {
    readyToDownload = true;
    readyToUpload = false;
  });
  this.arSessionManager!.onError("Upload avvenuto con successo");
}
```

Figura 4.36

Questa funzione carica le informazioni di ancoraggio su Firebase. Chiama un metodo chiamato *uploadAnchor* (Figura 4.37*)* su un oggetto firebaseManager, passando un ARAnchor (l'anchor appena caricato) e la posizione corrente ottenuta da un arLocationManager.
Carica tramite il metodo *uploadObject* (Figura 4.38), i nodi figlio su Firebase (se si tratta di un ancoraggio aereo) e successivamente, il codice controlla se l'ancoraggio è un'istanza di ARPlaneAnchor. Se lo è, entra in un ciclo per caricare i nodi figlio associati a questo ancoraggio su Firebase. Questo viene fatto iterando attraverso i childNodes dell'ancoraggio del piano e caricando oggetti (che rappresentano oggetti virtuali) su Firebase utilizzando firebaseManager. Il codice utilizza l'elenco dei nodi per trovare l'oggetto corrispondente in base al nome.

Di seguito viene riportata una panoramica di ciò che fa *uploadAnchor*:

- Controlla Firestore: controlla innanzitutto se l'oggetto firestore non è null. Questa è una buona pratica per garantire che Firestore sia inizializzato correttamente prima di tentare di utilizzarlo. Se Firestore è null, restituisce semplicemente, indicando che la funzione non procederà.
- Serializza Anchor: serializza l'oggetto ARAnchor in un formato JSON utilizzando il metodo *anchor. toJson* . Ciò consente ai dati di ancoraggio di essere archiviati come documento JSON in Firestore.
- Calcola ora di scadenza: calcola il tempo di scadenza per l'ancoraggio. Questo viene fatto prendendo il tempo corrente in millisecondi dall'epoca e aggiungendo il "tempo di vita" (TTL) dell'ancora in secondi convertiti in giorni.Il risultato viene archiviato nel campo expirationTime dell'ancoraggio serializzato.

```
void uploadAnchor(ARAnchor anchor, {Position? currentLocation}) {
  if (firestore == null) return;

  var serializedAnchor = anchor.toJson();
  var expirationTime = DateTime.now().millisecondsSinceEpoch / 1000 +
      serializedAnchor["ttl"] * 24 * 60 * 60;
  serializedAnchor["expirationTime"] = expirationTime;
  // Add location
  if (currentLocation != null) {
    GeoFirePoint myLocation = geo!.point(
        latitude: currentLocation.latitude,
        longitude: currentLocation.longitude);
    serializedAnchor["position"] = myLocation.data;
  }

  anchorCollection!
      .add(serializedAnchor)
      .then((value) =>
      print("Successfully added anchor: " + serializedAnchor["name"]))
      .catchError((error) => print("Failed to add anchor: $error"));
}
```

Figura 4.37

- Aggiungi posizione: se viene fornito un currentLocation (un oggetto Position contenente latitudine e longitudine), crea un GeoFirePoint utilizzando l'oggetto geo e aggiunge le informazioni geospaziali all'ancoraggio serializzato in un campo denominato "position". Ciò consente di associare l'ancoraggio a una posizione geografica specifica.

- Aggiungi Anchor a Firestore: aggiunge quindi il documento di ancoraggio serializzato alla raccolta Firestore (*anchorCollection Figura 4.38)* utilizzando il metodo *add*.

Di seguito viene riportata una panoramica di ciò che fa *uploadObject*:

- Serializza Object: serializza l'oggetto ARNode in un formato che può essere memorizzato in Firestore. Tramite il metodo *node.toMap* essenzialmente converte l'oggetto in una struttura simile a JSON.

- Aggiunge l'oggetto a Firestore: aggiunge quindi l'oggetto serializzato (la rappresentazione della mappa) a una raccolta Firestore (*objectCollection* Figura 4.39) utilizzando il metodo *add*.

```
void uploadObject(ARNode node) {
  if (firestore == null) return;

  var serializedNode = node.toMap();

  objectCollection!
      .add(serializedNode)
      .then((value) =>
      print("Successfully added object: " + serializedNode["name"]))
      .catchError((error) => print("Failed to add object: $error"));
}
```

Figura 4.38

I dati relativi ad Anchor Cloud API vengono salvati su Firebase Cloud in questo modo:

Figura 4.39

Figura 4.40

Download dei dati dal Cloud

Questa sezione fa riferimento alla descrizione delle modalità del download delle ammoniti virtuali in formato glb.
Nell' *initState ()* della view tramite uno streamPosition l'applicazione monitora ogni due secondi il flusso di aggiornamenti delle posizioni. Ogni volta che la posizione dell'utente cambia, viene eseguito il codice all'interno della funzione di callback che calcola la distanza tra la posizione dell'ammonite e la locazione dell'utente.

```
_getPositionSubscription  = Geolocator.getPositionStream(locationSettings: locationSettings)
    .listen((Position? position) async {
  setState(() {
    currentPosition=position!;
    distanceInMeters = Geolocator.distanceBetween(position!.latitude, position.longitude,
                                                  double.parse(widget.model.lat.toString()),
                                                  double.parse(widget.model.long.toString()));
  });
```

Dopo il calcolo della distanza viene effettuato il controllo se la variabile *distanceInMeters* è minore o uguale a 5 metri. Se questa condizione è vera, si dispone di un ciclo while che esegue ripetutamente una funzione di download dell'ammonite virtuale finché readyToDownload è true.

```
      if (distanceInMeters <= 5.00) {
      setState(() {
        vicino = true;
        while(readyToDownload){
          onDownloadButtonPressed();
        }
      });
    }else{
      setState(() {
        vicino = false;
      });
```

Il codice della funzione *onDownloadButtonPressed* gestisce il download degli ancoraggi in base alla posizione corrente e imposta il flag readyToDownload di conseguenza.
Il blocco condizionale controlla se la posizione corrente, ottenuta da arLocationManager, non è null. Se la posizione corrente è disponibile, procede con il download delle ancore tramite la funzione *downloadAnchorsByLocation* dove il parametro snapshot contiene informazioni sull'ancoraggio, ad esempio il suo ID cloud.
I dati di ancoraggio recuperati da Firebase vengono salvati in una mappa chiamata anchorsInDownloadProgress utilizzando l'ID di ancoraggio cloud come chiave.
Viene utilizzato arAnchorManager per scaricare l'ancoraggio associato all'ID di ancoraggio cloud ottenuto da Firebase.
Dopo aver avviato il download degli ancoraggi, viene impostato il flag readyToDownload su false. Questo indica che l'app è attualmente in fase di download degli ancoraggi.
Quando la posizione corrente non è disponibile (this. arLocationManager! currentLocation == null), gestisce il caso segnalando un errore.

```
Future<void> onDownloadButtonPressed() async {


  // Get anchors within a radius of 100m of the current device's location
  if (this.arLocationManager!.currentLocation != null) {
    firebaseManager.downloadAnchorsByLocation((snapshot) {
      final cloudAnchorId = snapshot.get("cloudanchorid");
      anchorsInDownloadProgress[cloudAnchorId] = snapshot.data() as Map<String, dynamic>;
      arAnchorManager!.downloadAnchor(cloudAnchorId);
    }, this.arLocationManager!.currentLocation, 0.1);
    setState(() {
      readyToDownload = false;
    });
  } else {
    this
        .arSessionManager!
        .onError("Location updates not running, can't download anchors");
  }
}
```

La *downloadAnchorsByLocation* è responsabile del recupero dei dati di ancoraggio da un database Firebase Firestore in base a una posizione geografica e a un raggio specificati inoltre sfrutta la libreria GeoFire per eseguire query geospaziali.
La funzione accetta un listener come parametro. Questo listener è una funzione di callback che verrà eseguita per ogni documento recuperato dalla query Firestore.
Il parametro location rappresenta il centro dell'area geografica per la query (specificato come oggetto Position Figura 4.20) e radius, che definisce il raggio all'interno del quale i documenti devono essere recuperati.
Viene creato un oggetto GeoFirePoint denominato center utilizzando l'oggetto geo. Converte la latitudine e la longitudine dal parametro di posizione in un GeoFirePoint, che è un punto sulla superficie terrestre che può essere utilizzato per query geospaziali.
Questo codice inizializza un flusso denominato stream che rappresenta un flusso di documenti dalla raccolta Firestore specificata da anchorCollection(*anchors*). Utilizza il metodo *within* per recuperare i documenti entro un raggio specificato (determinato dal raggio) dal punto centrale.

```
void downloadAnchorsByLocation(FirebaseDocumentStreamListener listener,
    Position location, double radius) {
  GeoFirePoint center =
  geo!.point(latitude: location.latitude, longitude: location.longitude);

  Stream<List<DocumentSnapshot>> stream = geo!
      .collection(collectionRef: anchorCollection!)
      .within(center: center, radius: radius, field: 'position');

  stream.listen((List<DocumentSnapshot> documentList) {
    documentList.forEach((element) {
      listener(element);
    });
  });
}
```

Conclusione

In questa tesi, abbiamo esplorato l'interessante campo dello sviluppo di applicazioni mobile, concentrandoci sull'analisi, la progettazione e l'implementazione di un'app specifica *AmmonitiExplorer*.
È stato esaminato in dettaglio l'integrazione della realtà aumentata (AR) in un'applicazione mobile, analizzando gli aspetti tecnologici, le sfide di progettazione e le potenziali applicazioni.
La creazione di *AmmonitiExplorer* è stata un processo complesso che ha coinvolto varie fasi, dalla pianificazione iniziale alla progettazione dell'interfaccia utente, dalla scelta delle tecnologie di sviluppo all'implementazione delle funzionalità chiave.
Uno dei risultati più rilevanti è stato l'identificazione di scenari d'uso specifici in cui la RA può apportare il massimo valore. L'implementazione di funzionalità di RA in un'app può migliorare l'esperienza dell'utente, aumentare l'interazione e fornire nuove opportunità per l'innovazione.
Tuttavia, questa tesi non rappresenta la fine del percorso per AmmoniteExplorer. I il mondo delle applicazioni mobile è in continua evoluzione, con nuove tecnologie, tendenze e esigenze degli utenti che emergono costantemente. Pertanto, le prospettive future per AmmoniteExplorer includono ulteriori sviluppi, aggiornamenti e adattamenti per rimanere rilevante e soddisfare le esigenze mutevoli degli utenti.
Un aspetto che potrebbe essere incluso nel progetto futuro potrà essere l'inserimento di elementi tipici dei giochi come obbiettivi,regole,ricompense, feedback e competizione con il fine ultimo di migliorare l'engagement degli utenti.
In conclusione, questa tesi ha rappresentato un passo significativo nel processo di sviluppo di AmmoniteExplorer e ha aperto nuove opportunità per migliorare e ottimizzare ulteriormente l'applicazione. L'esperienza acquisita in questo progetto può essere utilizzata come base per futuri progetti di sviluppo di applicazioni mobile.

Bibliografia

[1] Enciclopedia Treccani, le ammoniti - https://www.treccani.it/enciclopedia/

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

[3] Git - https://git-scm.com/doc

[4] Android Studio, features overview - https://developer.android.com/studio/features

[5] Dart, features overview - https://dart.dev

[6] JetBrains. "Kotlin". - https://kotlinlang.org

[7] Swift - Apple Developer. -https://developer.apple.com/swift/

[8] The pubspec file | Dart. -https://dart.dev/tools/pub/pubspec

[9] List of all Flutter packages - https://pub.dev/packages

[10] Mapbox | Maps, Navigation, Search, and Data.- https://www.mapbox.com/

[11] ARCore supported devices | Google for Developers. - https://developers.google.com/ar/devices?hl=it

[12] ARKit 6 - Augmented Reality - Apple Developer - https://developer.apple.com/augmented-reality/arkit/

[13] CariusLars/ar_flutter_plugin - GitHub - https://github.com/CariusLars/ar_flutter_plugin

[14] Flutter, architectural overview - https://docs.flutter.dev/resources/architectural-overview

[15] What's a GLB file?.-https://visao.ca/what-is-glb-file/

[16] Flutter,MVVM architecture.-https://medium.com/@ravipatel84184/mvc-vs-mvvm-in-flutter-choosing-the-right-architecture-for-your-app-d8a09e2e254c