UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea triennale in Ingegneria Informatica e dell'Automazione

Progettazione e sviluppo di una web app partecipativa per la scoperta di reperti fossili con realtà aumentata

A partecipative web app design and development to discover fossils through augmented reality

Relatore:

Prof. Emanuele Storti

Laureando:

Emanuele Biccheri

ANNO ACCADEMICO 2021/2022

Indice

Indice delle figure

Introduzione

L'obiettivo principale a cui ambisce questo elaborato è presentare la progettazione e l'implementazione di una web app, risultato di un progetto eseguito ai fini del tirocinio curricolare.
Lo sviluppo della web app discussa, in realtà, rappresenta il completamento di un lavoro molto più ampio, riguardante l'applicazione "Ammoniti di strada", per il quale erano state già create le versioni Android e iOS. L'applicazione in questione ha lo scopo di raccogliere le informazioni delle ammoniti sparse per la città di Ancona, rendendole disponibili agli utenti per una conoscenza più approfondita.
La finalità di questo lavoro era sviluppare una versione integrativa che, mediante l'introduzione e l'utilizzo della realtà aumentata, rendesse maggiormente partecipativa la scoperta dei reperti fossili, risultando coinvolgente e somigliante ad un gioco interattivo.
Il progetto mirava allo sviluppo di un'applicazione dedicata ai dispositivi mobile, per cui la scelta preliminare e necessaria ha riguardato la tipologia di applicazione da implementare: *nativa*, *web* o *ibrida*. Le *mobile applications*, infatti, possono essere suddivise in queste tre categorie. Le app native, in particolare, sono pensate e sviluppate per un determinato sistema operativo, come per esempio Android o iOS; riescono ad interfacciarsi totalmente con l'hardware del dispositivo, sfruttando ed ottimizzando tutte le sue potenzialità. Lo sviluppo della versione nati-

va per “Ammoniti di strada” ha riguardato la prima parte del progetto. Volendo quindi implementare un’applicazione in grado di funzionare, tramite il browser, in tutti dispositivi, si è optato per una versione web.
Nel primo capitolo si presenterà, dunque, l’applicazione già esistente. Si valuteranno inoltre i requisiti che la web app dovrà avere e si illustrerà una breve ma generale panoramica della realtà aumentata.
Nel secondo capitolo si discuterà della fase progettuale del lavoro, cioè quella in cui si valuta come procedere all’implementazione. Verranno dunque presentati gli strumenti utilizzati e saranno mostrate le strutture sia dei dati presenti nel database, sia del progetto della web app.
Infine, nell’ultimo capitolo, si analizzerà l’implementazione vera e propria, entrando nel dettaglio di come sono state sviluppate le funzionalità. Verranno inoltre elencate le problematiche riscontrate durante lo sviluppo e come queste siano state risolte. Saranno presentati poi alcuni possibili miglioramenti, riguardanti versioni future della web app.
La visualizzazione della web app nella sua versione finale sarà mostrata in Appendice, facendo riferimento ad un reale test di utilizzo eseguito post implementazione.

Capitolo 1

Analisi

In questo capitolo sarà mostrata un'analisi preliminare allo sviluppo vero e proprio della nuova web app. In particolare, si discuterà della situazione di partenza del progetto, ovvero l'applicazione già esistente, descrivendo sia l'interfaccia sia il database utilizzato. Saranno poi analizzati gli obiettivi funzionali e i requisiti tecnici che la web app dovrà avere. Infine sarà presentato un breve approfondimento sulla realtà aumentata, soffermandosi su riferimenti storici, classificazione generale e applicazioni.

1.1 Punto di partenza

Il primo passo dell'analisi, preliminare allo sviluppo della web app, riguarda la presentazione del progetto in cui questa si inserisce e la discussione dell'applicazione già esistente: "Ammoniti di strada".
Lo scopo dell'applicazione già sviluppata è raccogliere le informazioni relative alle ammoniti presenti per le strade della città di Ancona, in modo da permettere agli utenti di conoscerle e ritrovarle. Oltre a poter consultare le informazioni sui fossili, infatti, si può essere guidati verso la loro posizione.
La struttura dell'applicazione si sintetizza in cinque schermate principa-

li, tra cui è possibile navigare: *Home*, *Nelle vicinanze*, *Tutti i reperti*, *Segnalazioni* e *Login/Registrazione*.
La *Home* è composta da una mappa, centrata inizialmente sulla posizione dell'utente, in cui sono inseriti dei marker corrispondenti alle zone della città dove sono presenti delle ammoniti. L'utente può consultare la mappa e interagire con essa per scoprire le posizioni dei fossili. Cliccando su un marker si può accedere all'elenco delle ammoniti nella zona corrispondente; selezionando ogni ammonite singolarmente, si presentano informazioni specifiche come la descrizione, la foto, il nome e la posizione.
Attraverso la barra di navigazione posizionata in basso, si può passare alla schermata *Nelle vicinanze*. Qui viene mostrato un elenco contenente tutte le ammoniti presenti in un raggio di 1km rispetto alla posizione dell'utente. Come per il caso precedente, l'utente può cliccare su un reperto per aprire la scheda con tutte le informazioni.
La terza schermata *Tutti i reperti*, anch'essa accedibile dalla barra di navigazione, riporta l'elenco integrale delle ammoniti presenti all'interno dell'applicazione.
L'ultima schermata accedibile dalla barra di navigazione, *Segnalazioni*, permette di segnalare nuove ammoniti scoperte dall'utente, oltre a mostrare un riepilogo delle segnalazioni già fatte.
In alto a destra è possibile accedere all'area riservata, dove effettuare il login o una nuova registrazione dell'utente.

L'applicazione sfrutta un database creato in collaborazione con il Dipartimento di Scienze della Vita e dell'Ambiente dell'Università Politecnica delle Marche, che si è occupato dello studio e della raccolta dei dati.
La piattaforma su cui è implementato il database è Firebase, sviluppata da Google. L'organizzazione consiste in raccolte, ognuna delle quali

comprende uno o più documenti.
Tra le principali si cita *users*, in cui sono contenuti tutti gli utenti registrati all'applicazione, idenificati da un codice univoco: di ognuno si forniscono informazioni personali come nome, cognome ed email.
Un'altra raccolta, chiamata *dettaglio*, contiene invece tutte le ammoniti, rappresentate dal proprio nome scientifico, e le rispettive informazioni.
Per gestire le informazioni condivise da più fossili sono state create raccolte di supporto come *zone*, che si riferisce a zone della città, oppure *materiali*, che contiene le informazioni dei materiali dove le ammoniti si sono fossilizzate. Laddove sia necessario usufruire delle informazioni contenute in queste raccolte, verrà inserito il riferimento corrispondente.
La struttura in dettaglio verrà poi analizzata nella Sezione 2.2, dove saranno presentate anche le raccolte aggiunte successivamente, per sviluppare la nuova web app.

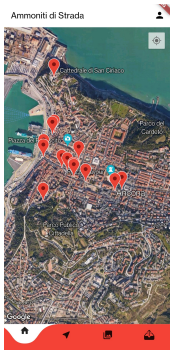

Figura 1.1: Homepage dell'applicazione "Ammoniti di strada".

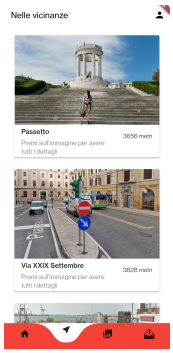

Figura 1.2: Schermata *Nelle vicinanze* di "Ammoniti di strada".

L'applicazione di partenza è già completa di tutte le informazioni sulle ammoniti sparse per la città, riesce a guidare l'utente alla loro scoperta grazie alla mappatura, offre la possibilità di consultare le descrizioni dettagliate dei fossili. Da queste sue caratteristiche nasce quindi l'idea di implementare una web app che renda ancora più interattiva la ricerca e

la scoperta delle ammoniti. Grazie alla realtà aumentata infatti, l'utente riuscirà ad interagire direttamente con i reperti fossili.

1.2 Requisiti

Dopo aver presentato l'applicazione di partenza, i dati a disposizione e come sono stati strutturati, il secondo step dell'analisi consiste nel valutare ed identificare i requisiti fondamentali che la web app dovrà soddisfare. In questo modo sarà più chiaro come procedere alla progettazione e allo sviluppo e capire quando il lavoro può considerarsi completato.
L'analisi dei requisiti consiste nel creare una sintesi di cosa andrà implementato e le caratteristiche tecniche che il software dovrà avere, riassumendo in modo sintetico il prodotto finale concettualizzato.
Si suddividerà lo studio dei requisiti in due categorie: requisiti funzionali e requisiti non funzionali. I primi rappresentano gli obiettivi veri e propri riguardanti il funzionamento della web app, cioè le funzioni a disposizione dell'utente finale. In sostanza, sono i requisiti "visibili" e necessari al corretto funzionamento. I requisiti non funzionali, invece, sono le caratteristiche tecniche dello sviluppo della web app, ovvero come le funzionalità devono essere sviluppate.

1.2.1 Requisiti funzionali

Tra i requisiti funzionali individuati per questa web app si inseriscono le possibilità di navigazione a disposizione dell'utente. Nello specifico, si dovranno garantire le seguenti funzionalità:

- *Guardarsi intorno:* all'apertura della web app, se tutti i permessi sono stati concessi, l'utente dovrà visualizzare sullo schermo le im-

magini riprese dalla fotocamera; si potrà così orientare e muovere nello spazio circostante, per cercare le ammoniti intorno a lui.

- *Visualizzazione delle ammoniti:* quando l'utente si troverà in prossimità di un'ammonite dovrà visualizzarla sullo schermo, come se fosse ripresa dalla fotocamera nello spazio reale; la posizione dovrà corrispondere alle coordinate assegnate nella mappa.

- *Muoversi con il dispositivo:* una volta trovata e visualizzata un'ammonite, l'utente potrà muoversi in tutte le direzioni continuando a vedere il fossile ancorato allo stesso punto del mondo reale.

- *Visualizzare le informazioni di un'ammonite:* quando l'utente cliccherà l'ammonite, se questa è sufficientemente vicina, verrà mostrata sullo schermo una piccola finestra riassuntiva del fossile selezionato, formata da foto, nome e la descrizione.

- *Aggiungere un'ammonite alla raccolta:* l'utente possiederà una raccolta personale di tutte le ammoniti trovate. Per collezionarne una, dovrà essere sufficientemente vicino e cliccarla sullo schermo. A quel punto, se nuova nella sua raccolta, verrà aggiunta; altrimenti, verrà segnalata come già presente.

- *Visualizzare la raccolta personale:* attraverso un pulsante sempre presente sullo schermo, l'utente potrà aprire e consultare una schermata di riepilogo con tutte le ammoniti già raccolte.

- *Visualizzare messaggi di avviso:* vi saranno delle situazioni in cui andranno comunicati dei messaggi all'utente, nello specifico: scarso segnale GPS, aggiunta di una nuova ammonite alla raccolta o la sua presenza, avviso se l'ammonite selezionata risulta troppo distante (in questo caso sarà comunicata anche la distanza).

1.2.2 Requisiti non funzionali

In questa sezione saranno presentati i requisiti non funzionali, cioè quelli di cui tener conto in fase di sviluppo, che riguardano l'implementazione del software e quali criteri dovrà seguire.

- *Velocità di esecuzione:* la web app dovrà risultare veloce nel suo funzionamento, in modo da garantire un buon utilizzo.
- *Geolocalizzazione:* la web app dovrà essere in grado di accedere alle informazioni sulla geolocalizzazione del dispositivo, previa concessione dei permessi, per calcolare la sua posizione e quella dei fossili. E' un fattore obbligatorio e necessario, senza il quale nulla potrà funzionare.
- *Fotocamera:* la web app dovrà poter accedere ai contenuti della fotocamera, anch'essi dopo aver ottenuto i permessi da parte dell'utente, per poterli mostrare sullo schermo. Anche questo, come la geolocalizzazione, è obbligatorio.
- *Interfacciamento con DBMS:* la web app si interfaccerà ad un DBMS per poter immagazzinare tutte le informazioni che saranno richieste. In questo caso il database è già esistente, dunque dovrà interagire con esso ed essere in grado di sfruttare tutti i dati già memorizzati.
- *Cross platform:* sarà importante che la web app sia disponibile all'utilizzo in tutte le piattaforme, ovvero che non necessiti di requisiti specifici del dispositivo.
- *Adattabilità:* lo sviluppo avverrà tenendo in considerazione dei possibili cambiamenti futuri, sia sulla struttura dei dati e del suo contenuto sia dell'applicazione stessa.

- *Espandibilità:* si terrà conto anche di possibili sviluppi futuri della web app in termini di funzionalità dunque si cercherà di sviluppare il software in modo tale da essere facilmente espandibile più avanti.
- *Interfaccia semplice:* l'interfaccia dovrà essere semplice, con pochi elementi sullo schermo in modo da incentrare l'esperienza sulle funzionalità di realtà aumentata.

1.3 La realtà aumentata

La web app presentata è incentrata sull'utilizzo di tecnologie che forniscono un'esperienza di realtà aumentata: sarà questo l'argomento trattato nella sezione corrente. Nello specifico, sarà discusso di che cosa si tratta, una classificazione generale delle varie tipologie ed infine i possibili utilizzi e potenzialità.

1.3.1 Definizione e cenni storici

La realtà aumentata consiste nell'interazione di informazioni digitali con il mondo reale. A differenza della realtà virtuale, che crea un mondo interamente immaginario, la realtà aumentata sfrutta il mondo tangibile inserendo al suo interno elementi digitali, sovrapponendoli di fatto all'ambiente circostante. Questi possono essere di vario tipo, ma tra i più comuni si citano: immagini, video, modelli 3D, animazioni.
Un primo esempio di realtà aumentata si può individuare negli anni '60 del secolo scorso. Ivan Sutherland, ricercatore e informatico statunitense, inventa e progetta in quel periodo un prototipo di occhiali per la realtà mediata, da cui discendono sostanzialmente i concetti di realtà virtuale e realtà aumentata: Head-mounted Display, o HMD [1].
Il termine *realtà aumentata* nasce ufficialmente solo nel 1990, coniato da

Thomas Caudell per descrivere gli HMD utilizzati alla Boeing Computer Services Research, impiegati come supporto nell'assemblaggio dei cavi all'interno degli aeromobili.
Nei primi anni duemila si diffondono poi applicazioni ludiche e biomediche, oltre che quelle industriali già ampiamente presenti.

1.3.2 Tipologie

La tecnologia a cui si fa riferimento con il termine "realtà aumentata" comprende un mondo vasto, caratterizzato da moltissimi sviluppi differenti e dai più disparati campi di applicazione. Parlare solo di inserimento di oggetti virtuali all'interno del mondo reale risulta alquanto riduttivo: per questo, nella sezione corrente, saranno schematizzate le diverse modalità con cui la realtà aumentata può interagire con l'ambiente circostante. Si individuano principalmente due categorie per il funzionamento della realtà aumentata: *Marker-Based AR* e *Markerless AR* [2]. Come facilmente deducibile dai nomi, la differenza consiste nell'utilizzo o meno di marker, cioè elementi riconosciuti attraverso la telecamera del dispositivo utilizzato. Infine si entrerà nel dettaglio della seconda categoria citata, la Markerless AR: *Projection*, *Location-Based*, *Overlay* e *Contour-Based* [3].

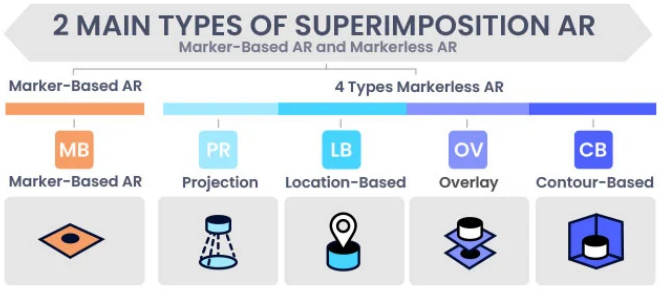

Figura 1.3: Schema riassuntivo delle tipologie di realtà aumentata.

Marker-Based

La prima tipologia presentata è quella basata sui marker. In questo caso, le funzioni di realtà aumentata vengono attivate dalla scannerizzazione di elementi prestabiliti, denominati *marker*. Questi consistono in elementi 2D facilmente riconoscibili dalla fotocamera, come QR Code o codici a barre; in alternativa possono essere immagini, purchè riconoscibili. Quando i marker vengono centrati, si genera sul display del dispositivo utilizzato il contenuto desiderato; questo rimarrà ancorato alla posizione del suo marker fino a quando inquadrato della fotocamera. È infatti possibile muoversi intorno al contenuto virtuale generato e visionarlo da qualsiasi suo lato.

Projection

Un primo esempio di realtà aumentata che non utilizza i marker è denominato *Projection*. Questo sfrutta fasci di luce per generare una proiezione del contenuto digitale direttamente su superfici, possibilmente piane, come un tavolo o un muro. In questo caso l'utente non è vincolato all'utilizzo di un dispositivo per la fruizione della realtà aumentata, in quanto le immagini sono visualizzate direttamente nel mondo reale e non attraverso uno schermo. L'interazione con il contenuto digitale, infatti, avviene mediante tocchi o gesti che il soggetto effettua direttamente sull'immagine proiettata. Generalmente per sfruttare questa tecnologia sono necessari un proiettore, che genera il contenuto, ed una o più fotocamere, in grado di identificare gli eventi che si verificano nell'ambiente dell'utente.
Alcuni tra gli esempi più comuni di realtà aumentata basata sulla proiezione sono la generazione di ologrammi o di tastiere per dispositivi

portatili.

Location-Based

Un altro tipo di realtà aumentata "markerless" è basata sulla posizione. È una tecnica che sfrutta i dati geografici per creare una mappatura e posizionare in essa i contenuti da generare. In questo caso si sfruttano sensori del dispositivo, come il GPS, la fotocamera, la bussola o l'accelerometro, per inserire elementi all'interno della mappatura virtuale. Infine si collocano nella corretta posizione tutti gli oggetti digitali, in riferimento alla posizione dell'utente.
Una prima soluzione di implementazione prevede unicamente l'utilizzo della geolocalizzazione, basando la posizione del dispositivo e degli altri oggetti sull'acquisizione dei dati del GPS; in alternativa si rilevano le superfici dell'ambiente circostante, creando un reticolo digitale in cui posizionare gli oggetti, sfruttando i sensori del dispositivo.
La web app presentata in questo elaborato utilizzerà la realtà aumentata basata sulla posizione che sfrutta il segnale GPS.

Overlay

Questa tipologia consiste nel sostituire interamente o parzialmente un oggetto reale con una sua versione digitale. Nello specifico, può essere utilizzata per rimpiazzare un oggetto con una sua versione più evoluta oppure modificata. In alternativa, si possono visualizzare informazioni corrispondenti all'oggetto inquadrato dal dispositivo.
È molto importante in questo caso che il dispositivo possa identificare l'oggetto nel mondo reale, da sostituire poi totalmente o parzialmente.
Un possibile esempio è rappresentato dall'animazione delle fasi di mon-

taggio e smontaggio di un macchinario, inquadrato dall'utente con la fotocamera del proprio dispositivo.

Contour-Based

L'ultima tecnologia presentata si basa sull'identificazione dei contorni di oggetti reali ed è spesso impiegata con la finalità di aiutare l'utente in azioni quotidiane.
Tra le applicazioni più frequenti e utilizzate, si ricordano: la misurazione delle dimensioni di un oggetto, inquadrato dal dispositivo per mezzo della fotocamera, e il riconoscimento dei bordi di una carreggiata, tramite sensori integrati alla vettura, come supporto dell'automobilista in situazioni di scarsa visibilità, distrazione o disattenzione.

1.3.3 Applicazioni

Risulta evidente, da quanto appena presentato, che i possibili utilizzi delle tecniche di realtà aumentata sono molteplici e sembra possibile trovarne ancora di nuovi. Seppur considerata già una tecnologia all'avanguardia, il suo sviluppo può essere ancora considerevole e i campi di applicazione sono davvero moltissimi e in prospettiva di ampliamento.
Come già accennato nella Sezione 1.3.1, le prime applicazioni avvennero nel campo industriale come supporto ai lavoratori nelle loro mansioni. Con il tempo, il suo utilizzo in questo settore è andato così in crescendo da arrivare a parlare addirittura di "Industria 4.0". Sono moltissimi i benefici infatti che il suo impiego comporta: gli operai possono essere supportati da strumenti in grado di guidarli nel loro lavoro, anche segnalando in tempo reale eventuali anomalie nelle procedure, per esempio. Questo riduce gli errori, talvolta evitando conseguenze come infortuni, ed aumenta di gran lunga l'efficienza della mansione.

La realtà aumentata sta vedendo ultimamente un grande sviluppo anche in campo medico. Un enorme contributo è rappresentato dalla possibilità, prima di un intervento, che un medico ha di sfruttare queste tecnologie per esplorare gli organi da operare, visualizzare dei modelli dei casi chirurgici pù complessi oppure simulare operazioni analoghe a quella che andrà a svolgere.
Anche nel campo della formazione si possono trarre grandi benefici. Sempre in riferimento al campo medico per esempio, la realtà aumentata può fornire un grande aiuto nel consultare elementi anatomici del corpo umano in fase di apprendimento scolastico. Più in generale, può essere sfruttata per rendere più partecipative le lezioni ed aiutare gli studenti nell'acquisizione delle nozioni con più consapevolezza.
Non può mancare ovviamente l'utilizzo della realtà aumentata a scopo ludico. Sono infatti tantissimi i videogiochi che sfruttano questa tecnologia per rendere l'esperienza dell'utente più coinvolgente. Un esempio molto calzante è rappresentato dall'applicazione *PokemonGo*, usando la realtà aumentata basata sulla posizione.
Un'ultima applicazione quotidiana da presentare è l'impiego della realtà aumentata nella costruzione degli edifici. In questo caso si può creare una simulazione del prodotto finale, sia dal punto di vista dell'aspetto esterno sia interno. Nello specifico, può essere sfruttata per mostrare il progetto finale di arredamento di una stanza o di una casa intera, aiutando l'acquirente a scegliere e valutare più alternative.

Capitolo 2

Progettazione

Terminata la fase di analisi, si passa alla progettazione: a questo punto, si studia per capire come impostare lo sviluppo della web app.
In questo capitolo, dunque, saranno discusse le fasi iniziali dell'implementazione. Nello specifico, saranno prima presentati tutti gli strumenti che si è deciso di utilizzare durante lo sviluppo, comprendendo tra gli altri: editor, linguaggi, librerie e browser. Verrà poi analizzata nel dettaglio la struttura del database, con particolare riferimento alle informazioni più utilizzate dalla web app e alla loro organizzazione. Infine, si mostrerà la struttura scelta per il progetto, presentando i file e la cartella di supporto creati.

2.1 Strumenti utilizzati

Il primo passo della fase progettuale consiste nell'individuare gli strumenti necessari allo sviluppo.
Come già anticipato nel Capitolo 1, l'applicazione presentata dovrà essere fruibile dai dispositivi mobile: risulta quindi ottimale svilupparla in versione web, garantendo l'accessibilità a tutti gli utenti ed evitando dipendenze da caratteristiche software o hardware degli strumenti

utilizzati. Infatti, sarà sufficiente servirsi di un browser che sfrutti la connessione ad internet, per far funzionare l'applicativo.
Durante la fase di sviluppo, saranno utilizzati due diversi dispositivi per testare le varie fasi dell'implementazione, compreso il test finale: in uno è installato il sistema operativo Android, mentre nell'altro iOS. La scelta di impiegarli entrambi, testando il corretto funzionamento sulle diverse le piattaforme, è finalizzata a garantire la fruizione della web app a tutti gli utenti.
Implementare l'applicazione in versione web comporta automaticamente l'utilizzo di tre linguaggi: *HTML* [4], *CSS* [5] e *JavaScript* [6]. Sono questi, infatti, i linguaggi principali con i quali si generano le pagine web.
È poi necessario scegliere un ambiente di sviluppo nel quale programmare la web app: non dovendo usufruire delle funzionalità messe a disposizione da un generico IDE, la scelta è ricaduta sull'editor di testo *Visual Studio Code* [7].
Per includere le funzionalità di realtà aumentata, si devono integrare nel codice dell'applicazione alcune librerie che le implementino. È stata effettuata, dunque, una ricerca preliminare, al fine di trovare quella più opportuna. Nel caso in questione, è necessario implementare l'utilizzo della geolocalizzazione, per determinare il posizionamento dei contenuti. Per questo, dopo aver valutato diverse alternative, la libreria scelta e che più soddisfaceva i requisiti richiesti è *AR.js* [8].
Un altro strumento impiegato, e già discusso in precedenza, è *Firebase* [9]: questo permette principalmente di memorizzare le informazioni e consente l'autenticazione degli utenti. Nello specifico, si utilizzano i prodotti: *Firestore Database* per contenere i dati nel database, *Storage* per memorizzare i file necessari e *Authentication* per gestire l'autenticazione degli utenti.

È necessario per lo sviluppo anche il supporto di un server locale, che permetta di testare la web app. In questo caso è stato scelto *Google Chrome* [10], sia per il server locale sia come browser sul dispositivo.
Un ultimo strumento impiegato in fase di sviluppo è *GitHub* [11]: un servizio di hosting per progetti, nel quale si possono caricare diverse versioni del lavoro in corso d'opera.
Nelle prossime sezioni saranno discussi più nel dettaglio tutti gli strumenti sopra presentati.

2.1.1 Visual Studio Code

Visual Studio Code è un editor di codice sorgente distribuito da Microsoft, disponibile per Windows, macOS e Linux.
Permette di programmare in svariati linguaggi di programmazione, tra cui tutta la famiglia C, Java, Python, oltre che i linguaggi web. In base al linguaggo utilizzato, è in grado di mettere a disposizione funzionalità dedicate, dare suggerimenti o segnalare errori nel corso della scrittura del codice.
Offre un supporto grafico attraverso l'interpretazione del codice: utilizzando colori diversi, aiuta il programmatore a orientarsi meglio e a mantenere un chiaro schema visivo.
Per lavorare con l'editor si sfrutta la finestra *Explorer*, nella quale si apre la cartella di lavoro memorizzata nel computer o si scrivono nuovi file.
Visual Studio Code fornisce anche la possibilità di eseguire il codice in modalità di debug tramite la funzione *Run and Debug*, configurabile in base all'applicativo che si sta testando.
È infine possibile ampliare le funzionalità, installando delle estensioni presenti nel *Marketplace*. Tali estensioni possono essere sviluppate da Microsoft stessa oppure da terzi, poi messe a disposizione degli utenti.

2.1.2 HTML, CSS e JavaScript

HTML, acronimo di *HyperText Markup Language*, è un linguaggio di markup per pagine web. Per mezzo dell'HTML si descrive la struttura della pagina web, l'impaginazione e il layout del contenuto inserito.
Non è classificabile come un vero e proprio linguaggo di programmazione, bensì come linguaggio di formattazione. Non sono previste, infatti, variabili, strutture dati o funzioni che agiscano da programma.
L'HTML ha il solo scopo di rappresentare i contenuti all'interno della pagina, utilizzando dei tag di formattazione. Ogni tag rappresenta il determinato ruolo che il contenuto associato deve avere; l'insieme di tutti i tag, eventualmente concatenati tra loro, rappresenta lo scheletro della pagina web. Per la successiva visualizzazione, il codice HTML viene letto dal browser che, sulla base dei contenuti e dei tag associati, si occupa di interpretarlo e generare la pagina desiderata.
A livello sintattico, ogni elemento è formato da un tag di apertura e uno di chiusura; il contenuto si trova interposto. Il tag di apertura è formato dal suo codice identificativo, racchiuso tra due prentesi angolari: per esempio * * corrisponderà ad un paragrafo, *<h1>* ad un titolo. Nel tag di chiusura si aggiunge alla stessa scrittura uno slash dopo l'apertura della parentesi angolare: * * o *</h1>*. L'esempio dell'elemento completo risulta quindi: *<h1>titolo da inserire</h1>*.
Esistono anche tag a chiusura implicita, che agiscono nel preciso punto in cui sono inseriti, senza bisogno di tag di chiusura: un esempio è *<img>*, spesso utilizzato per inserire un immagine.
All'interno del tag è possibile settare dei parametri. I più comuni sono l'*id*, cioè un codice che identifica l'elemento, oppure le regole di stile, per definire il layout che l'elemento deve avere.

Figura 2.1: Esempio di creazione contenuti con HTML.

È possibile eventualmente raggruppare le regole di stile in un unico file, separato, in modo da poterle sfruttare per più elementi o semplicemente per snellire il file HTML: questo processo si elabora tramite il *CSS*.

Il CSS, acronimo di *Cascading Style Sheets*, è un linguaggio che permette di definire le regole di formattazione dei file HTML. Nello specifico, si descrivono al suo interno i layouts che gli elementi dell'HTML dovranno avere. Per collegare tali regole agli elementi, si può far riferimento al loro id univoco oppure assegnarle ad una classe, alla quale possono appartenere più elementi.
Nel caso in cui ad un elemento vengano applicate due regole diverse, ad esempio "colore rosso" sul codice e "colore verde" tramite una classe, il CSS prevede una gerarchia di default: la priorità maggiore è data alle regole inline (cioè scritte direttamente sul tag), seguono quelle associate all'id ed infine quelle associate alla classe o al tipo di tag. Il termine *Cascadin Style Sheets* deriva infatti proprio da questa particolarità.

L'HTML e il CSS lavorano insieme per strutturare la pagina web, ma il contenuto che ne deriva è statico, non può più variare. Per questo motivo si utilizza Javascript, un linguaggio di programmazione che può interagire con l'HTML e modificarne i valori in corso d'opera. È in grado infatti di creare nuovi elementi, eliminare e modificare quelli già presenti. In

sostanza, permette di modificare il codice HTML in maniera dinamica, sfruttando tutte le potenzialità intrinseche di un linguaggio di programmazione.
La sintassi di Javascript deriva dal C, con il quale condivide perciò moltissime caratteristiche. Anche se può essere utilizzato in diversi ambiti, il suo impiego principale è sicuramente lavorare alle spalle di una o più pagine HTML.
Le modifiche apportate da JavaScript possono realizzarsi in seguito a eventi specifici, che si verificano sulla pagina web, come un click del mouse. Tali eventi possono essere previsti e programmati appositamente per attivare delle funzioni che modifichino la pagina.

2.1.3 Libreria AR.js

AR.js è una libreria open source che implementa funzionalità di realtà aumentata per il web e per questo motivo realizzata interamente in JavaScript. Le tipologie di utilizzo a disposizione sono tre: *Image tracking*, *Marker tracking* e *Location Based*. Come facilmente intuibile dal nome, le prime due sfruttano, rispettivamente, un'immagine e un marker, sui quali poi attivare le funzionalità di realtà aumentata. L'ultima, invece, ricorre alle informazioni di geolocalizzazione; per questo motivo è la tipologia utilizzata per lo sviluppo della web app presentata nell'elaborato.
AR.js sfrutta a sua volta una libreria, *A-Frame*, che genera contenuti di realtà aumentata e li adatta, in modo da utilizzare le coordinate geografiche per posizionare gli oggetti. In realtà, esiste anche una versione basata interamente sulla libreria *three.js*: questa però non sarà presentata, in quanto non impiegata nello sviluppo della web app.
Per mezzo di A-Frame si possono creare elementi digitali semplicemente tramite un file HTML: questa infatti mette a disposizione tag perso-

nalizzati, ognuno dei quali svolge un preciso ruolo nella produzione del contenuto di realtà aumentata. I principali sono: <*a-scene*>, che crea la mappatura del mondo virtuale, <*a-camera*>, che sfrutta le immagini della fotocamera, e <*a-entity*>, che inserisce un oggetto. Ognuno di loro ha anche dei parametri eventualmente da definire: al tag <*a-entity*>, per esempio, si può specificare la posizione che l'oggetto assumerà in coordinate del mondo virtuale, oppure le dimensioni o ancora il colore.
AR.js aggiunge a sua volta funzionalità, adattando le componenti di A-Frame all'utilizzo del GPS: per esempio, confrontando <*a-entity*> con le coordinate del dispositivo, riesce a posizionare l'oggetto digitale nel mondo reale.
La libreria e le sue funzionalità saranno analizzate ancora più nel dettaglio successivamente, quando si presenterà il codice sviluppato per la web app.

2.1.4 Firebase

Firebase è una piattaforma per la creazione di applicazioni per dispositivi mobile e web. I prodotti che mette a disposizione sono classificabili in quattro categorie principali: *Creazione*, *Rilascio e Monitoraggio*, *Analisi*, *Coinvolgimento* [12]. Per lo sviluppo della web app presentata in questo elaborato, nello specifico, ne sono stati utilizzati tre, tutti appartenenti alla categoria Creazione: *Firestore Database*, *Storage* e *Authenticaton*.
Firestore Database è un database NoSQL, cioè caraterizzato dal fatto di non utilizzare il modello relazionale. Le informazioni nel database sono strutturate principalmente in tre livelli.
Lo strato più esterno è la raccolta: ognuna contiene più documenti, contrassegnati da un id univoco. Ogni documento ha poi al suo interno uno o più campi, che possono essere di formati diversi come numeri, stringhe

o vettori. In alternativa, può essere inserita una nuova raccolta contenente ulteriori documenti. Così facendo, si può ottenere un database con moltissimi livelli e non per forza definito da una struttura rigida e schematica.
Storage è un prodotto utilizzato per memorizzare e consultare qualsiasi tipo di file, organizzando anche attraverso le cartelle.
Infine, Authentication è uno strumento che permette di memorizzare gli utenti registrati e gestirne l'autenticazione, per consentire l'accesso alle informazioni già memorizzate in Firebase.
Nella Figura 2.2 è possibile visualizzare la dashboard del progetto "Ammoniti di strada".

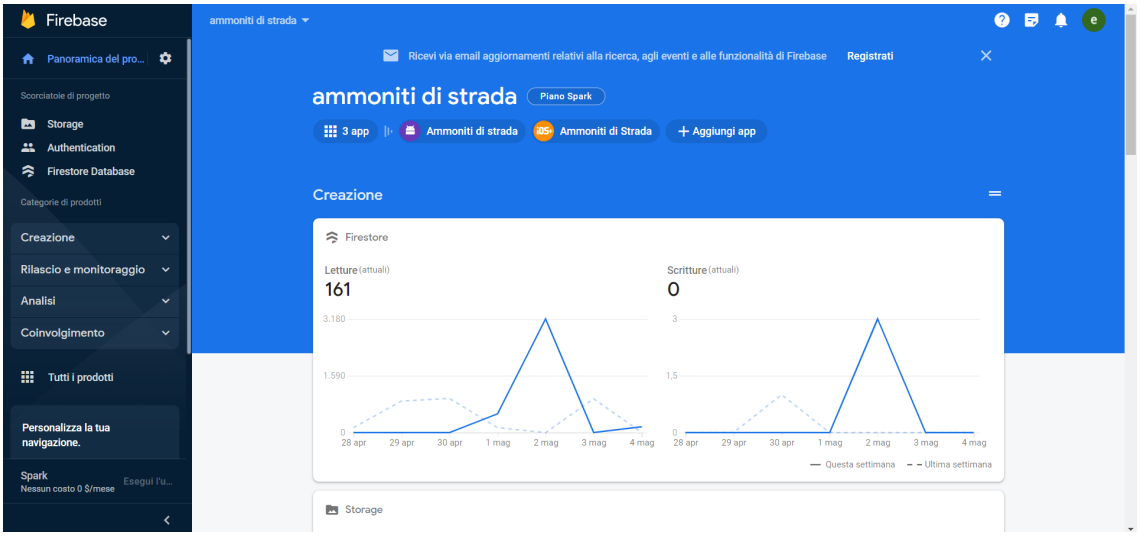

Figura 2.2: Dashboard della Console Firebase.

2.1.5 Google Chrome

Google Chrome è un browser web sviluppato da Google. Nel progetto presentato, in particolare, è stato utilizzato sia come browser sia come

web server locale, per testare l'applicazione durante lo sviluppo. Infatti, rende possibile il funzionamento della web app sul dispositivo mobile, nonostante il contenuto sia memorizzato solo sul computer, sfruttandolo come server. Per fare ciò, è necessario: mantenere collegato il dispositivo al computer tramite un cavo, impostarlo in modalità di debug e poi sfruttare due ulteriori strumenti, messi a disposizione dal browser.
Il primo è un'applicazione, chiamata *Web Server for Chrome*, che consente di visualizzare sul browser del computer il contenuto della cartella di progetto locale, ad un indirizzo impostato a piacimento sull'app.
L'altro, invece, è la pagina di *Ispezione devices*, che permette di fruire dello stesso contenuto della cartella anche sul dispositivo mobile. Da questa pagina è possibile inoltre ispezionare quanto visualizzato nel dispositivo, consultando gli strumenti per sviluppatori messi a disposizione dal browser.

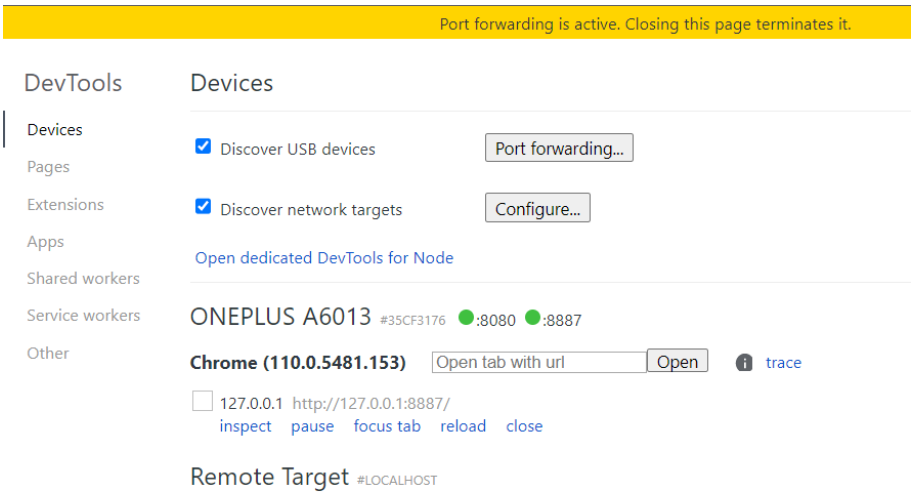

Figura 2.3: Pagina di *Ispezione devices*.

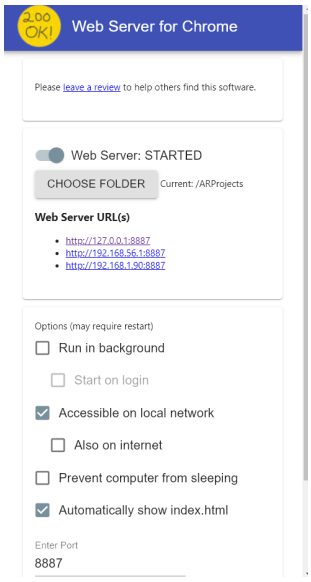

Figura 2.4: Applicazione *Web Server for Chrome*.

2.1.6 GitHub

GitHub è un servizio di hosting per progetti software, che permette agli sviluppatori di archiviare su un Cloud il proprio lavoro e gestire le modifiche apportate successivamente. Un'altra caratteristica molto utile è la possibilità di condivisione del progetto con altri utenti, che quindi possono collaborare.

Le modifiche sono salvate creando delle *commit*, ognuna corrispondente al salvataggio desiderato, e procedendo con una *pull request*, cioè chiedendo alla piattaforma di caricarle sul Cloud.

GitHub offre inoltre il controllo delle versioni, ovvero permette di tornare a versioni precedenti di un file specifico o dell'intero progetto.

Consente inoltre di creare più *branches*, cioè più ramificazioni dello sviluppo del progetto: in sostanza, si può lavorare contemporaneamente e parallelamente a diverse versioni. Quest'ultima funzionalità risulta utile nell'apportare modifiche ad un lavoro, senza rischiare di comprometterlo:

infatti, utilizzando *branch*, si crea una copia e si lavora solo su quella. Alla fine del lavoro si possono unire (*merge*) le due *branches*, aggiornando la versione più vecchia.

2.2 Progettazione dei dati

Come discusso nella Sezione 1.1, i dati utilizzati erano già presenti sul database poichè impiegati nell'applicazione precedente. Per l'implementazione della nuova web app, è stata sfruttata una parte di quei dati ed arricchita con integrazioni ad hoc.
Il database sarà analizzato nel dettaglio nelle sezioni seguenti, soffermandosi sulla struttura già esistente e su ciò che è stato creato di nuovo. Nello specifico, saranno descritti la struttura di Firestore Database ed i file memorizzati nello Storage.

2.2.1 Firestore Database

Il database Firestore è composto in totale da otto raccolte:

- *ammoniti:* contiene le informazioni delle ammoniti e risale alla fase embrionale dell'app, attualmente non utilizzata;
- *dettaglio:* sostituisce la raccolta precedente, *ammoniti*, e contiene infatti le informazioni di tutte le ammoniti;
- *livelli:* è una raccolta di supporto, utilizzata per suddividere in tre livelli una stessa zona della città;
- *materiali:* contiene i materiali, con le rispettive descrizioni, di cui sono fatti i fossili;

- *segnalazioni:* contiene l'elenco di tutte le segnalazioni fatte dagli utenti;
- *users:* comprende tutti gli utenti registrati nell'app e le loro informazioni personali;
- *webApp:* è la raccolta creata appositamente per la web app, dove vengono gestiti i dati finalizzati alla sua implementazione;
- *zone:* contiene tutte le zone in cui è suddivisa la città.

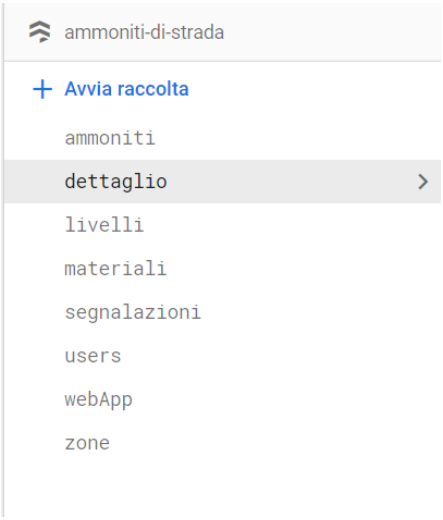

Figura 2.5: Raccolte presenti in Firestore Database.

Saranno ora analizzate più nello specifico le singole raccolte, presentando i campi inseriti all'interno dei documenti. Si discuteranno solamente le raccolte effettivamente utilizzate nella web app: *users*, *zone*, *materiali*, *dettaglio* e *webApp*.

La raccolta *users* è formata da una serie di documenti che rappresentano gli utenti. Ogni documento è identificato univocamente dall'*uid*, assegnato all'utente corrispondente, e contiene al suo interno le seguenti informazioni personali:

- *email* (stringa): email dell'utente;
- *firstName* (stringa): nome dell'utente;
- *secondName* (stringa): cognome dell'utente;
- *uid* (stringa): codice identificativo dell'utente all'interno di Firebase.

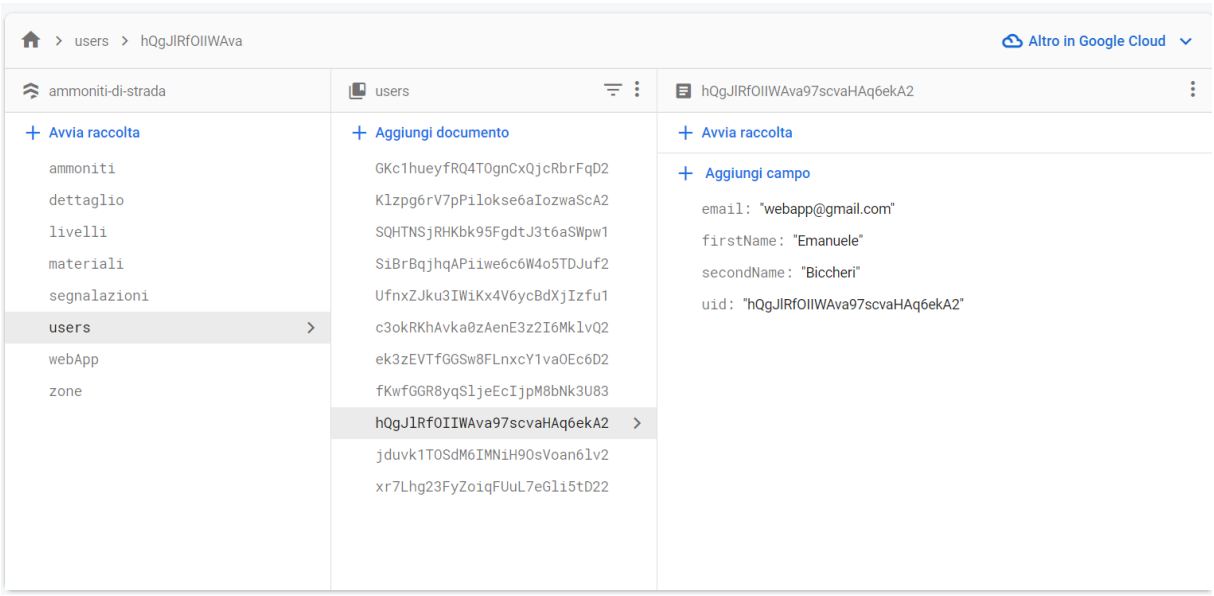

Figura 2.6: Raccolta *users*.

Per quanto riguarda la raccolta *zone*, ogni documento contenuto rappresenta una zona della città di Ancona ed è costituito dai seguenti campi:

- *descr* (stirnga): descrizione della zona;
- *fotozona* (stringa): nome della foto rappresentante la zona;
- *id* (stringa): codice identificativo della zona;
- *lat* (numero): latitudine della zona;

- *livelli* (stringa): codice per collegare i livelli alla zona (se non possiede livelli il valore sarà 0);
- *long* (numero): longitudine della zona;
- *nomezona* (stringa): nome della zona.

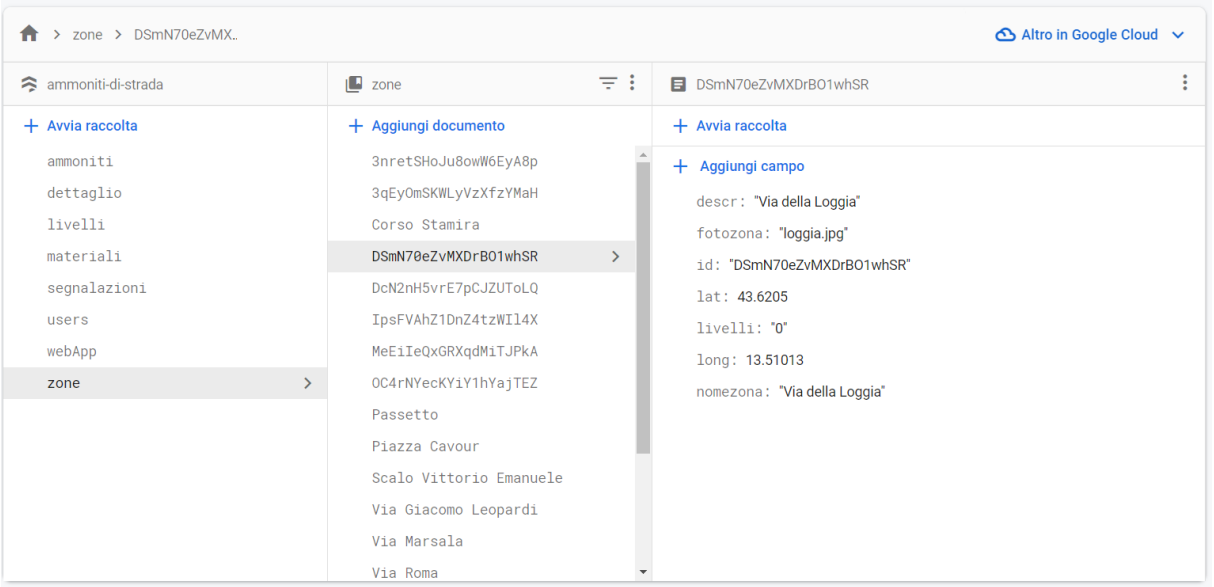

Figura 2.7: Raccolta *zone*.

La raccolta *materiali* è costituita da tutti i materiali in cui le ammoniti si sono fossilizzate. Ciasciun documento comprende:

- *descrizione* (stringa): descrizione del materiale;
- *eta* (stringa): periodo storico del materiale;
- *id* (stringa): codice identificativo del materiale;
- *nome* (stringa): nome del materiale;
- *provenienza* (stringa): provenienza del materiale.

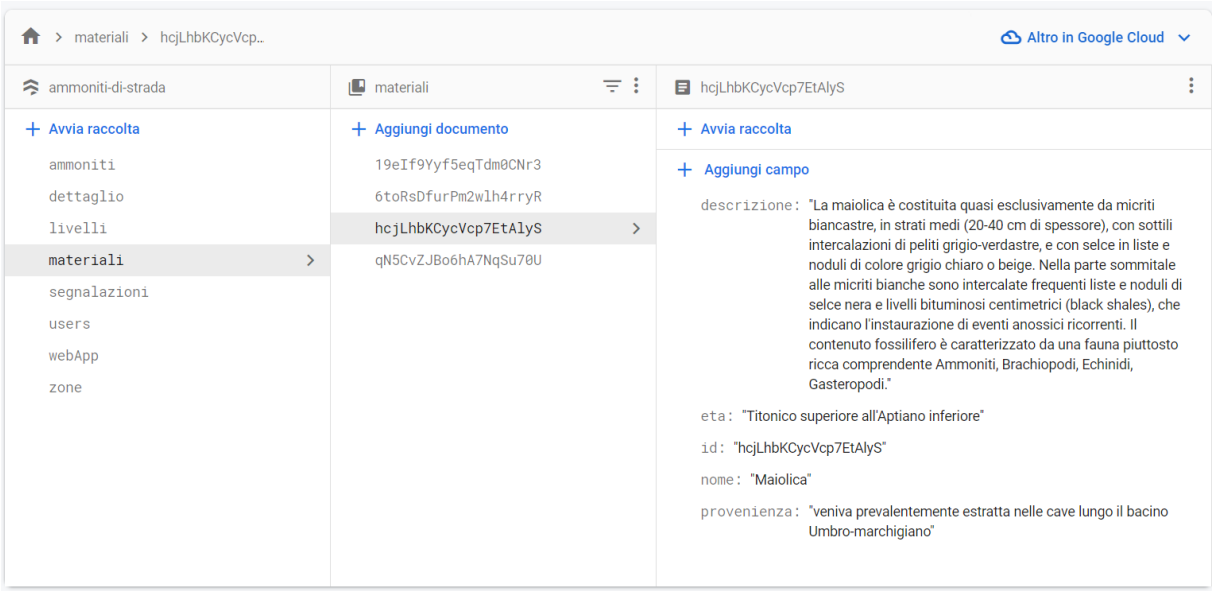

Figura 2.8: Raccolta *materiali*.

La raccolta *dettaglio* è formata da tutti i fossili catalogati. Come codice identificativo è stato assegnato il proprio nome di riferimento. I campi contenuti sono:

- *datetime* (timestamp): data di inserimento del fossile nel database;
- *descrfossile* (stringa): descrizione del fossile;
- *fotofossile* (stringa): nome di riferimento della foto del fossile;
- *materiale* (stringa): codice identificativo del materiale, che coincide con quello nella raccolta *materiali* per collegarli;
- *zona* (stringa): codice identificativo della zona, che coincide con quello nella raccolta *zone*.

Infine, la raccolta *webApp* è stata creata per supportare la web app e fa riferimento alla raccolta personale di ogni utente. Nello specifico, si crea un documento per ogni utente che inizia una raccolta, identificandolo con

lo stesso uid del proprietario in *users*, per essere facilmente collegabili. I documenti all'interno contengono un solo elemento, di tipo array: *ammoniti*, in cui saranno aggiunte tutte le ammoniti che l'utente raccoglierà.

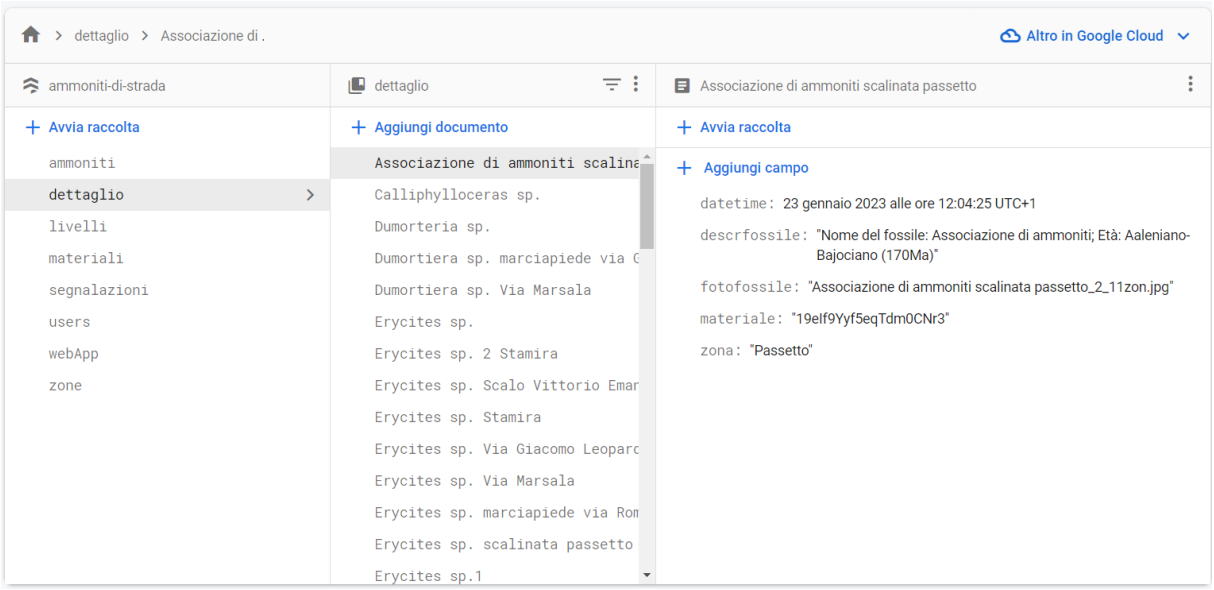

Figura 2.9: Raccolta *dettaglio*.

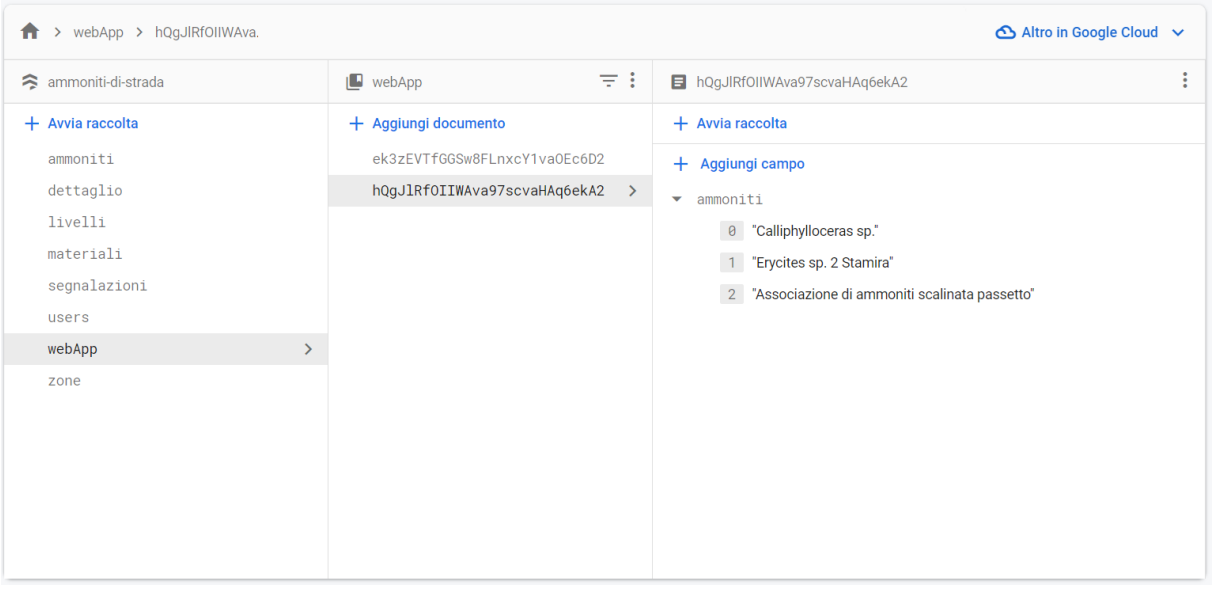

Figura 2.10: Raccolta *webApp*.

2.2.2 Storage

Il prodotto Storage appare più semplice, essendo utilizzato come servizio di archiviazione per tutte le foto presenti nell'applicazione: quelle delle ammoniti, delle zone, dei materiali e delle segnalazioni.
Le cartelle contenenti le immagini sono mostrate in Figura 2.11. Si puntualizza tuttavia che *foto zona* e *foto ammoniti*, seppur presenti, non sono mai state effettivamente utilizzate. Infatti, le cartelle sfruttate dall'applicazione originale sono:

- *foto dettaglio:* foto delle ammoniti presenti nella raccolta *dettaglio*;
- *foto materiali:* foto di tutti i materiali della raccolta *materiali*;
- *foto segnalazioni:* foto delle segnalazioni fatte dagli utenti, presenti nella raccolta *segnalazioni*;
- *foto zona 2:* foto delle zone presenti nella raccolta *zone*.

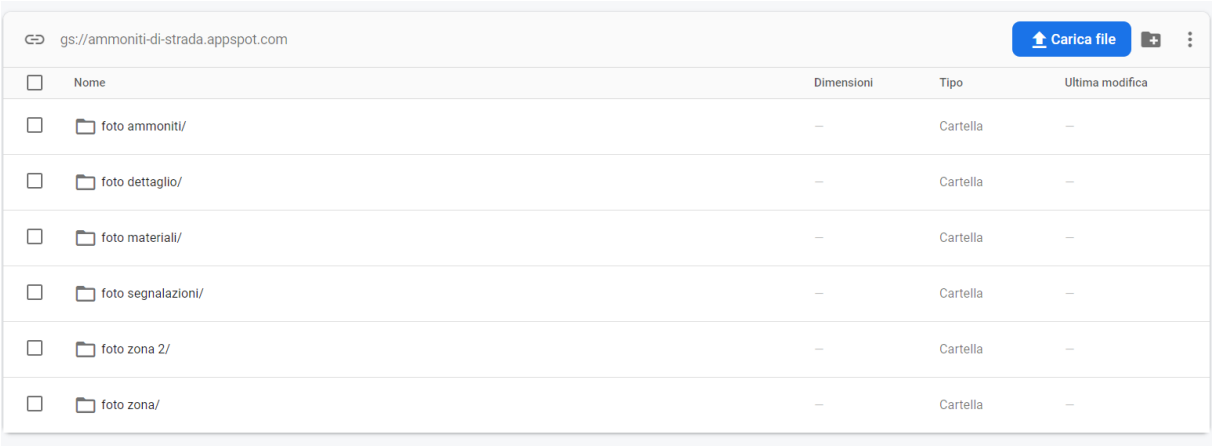

Figura 2.11: Cartelle presenti in Storage.

Per quanto riguarda la web app implementata, si utilizza nello specifico solo la cartella *foto dettaglio*.

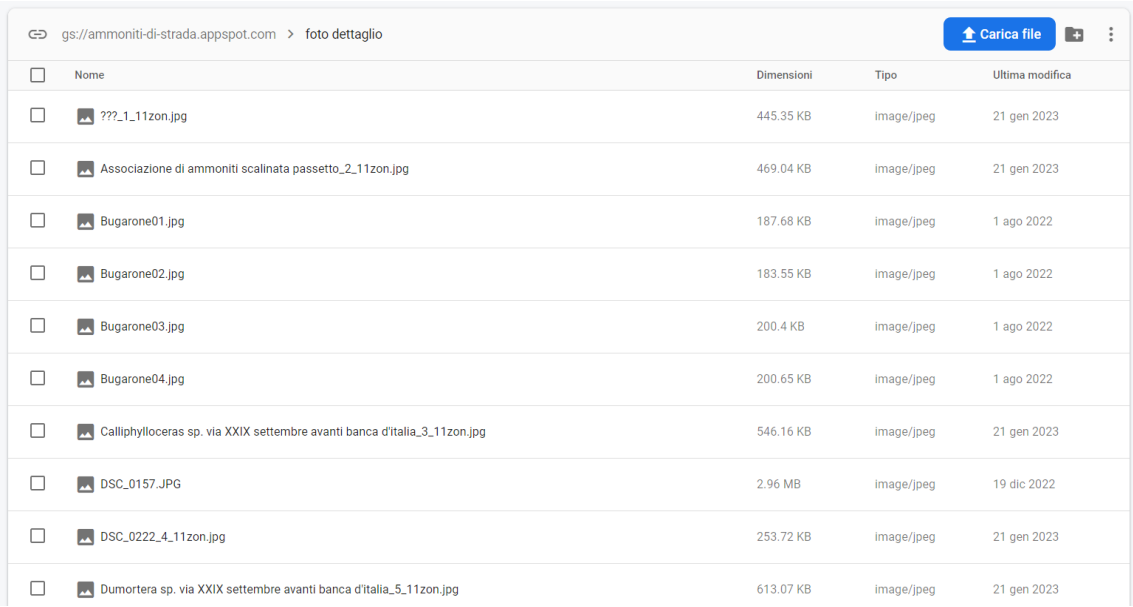

Figura 2.12: Contenuto della cartella *foto dettaglio*.

2.3 Struttura dell'applicazione

La struttura della web app è molto semplice. È composta essenzialmente da tre file, una libreria e una cartella di supporto, come mostrato in Figura 2.13.

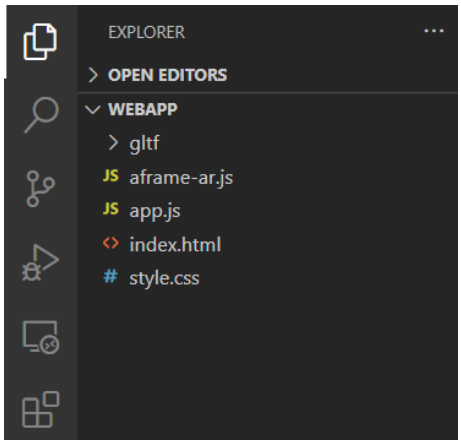

Figura 2.13: Struttura della raccolta di progetto.

Di seguito si puntualizza il contenuto di ogni elemento del progetto:

- *index.html:* file HTML per la generazione della pagina web;
- *style.css:* file con tutte le regole di stile da applicare alla pagina;
- *app.js:* file contenente tutto lo script in JavaScript, tra cui anche la connessione a Firebase;
- *aframe-ar.js:* libreria che implementa tutte le funzionalità di AR.js;
- *gltf:* cartella contenente i modelli 3D delle ammoniti da mostrare tramite la realtà aumentata.

Capitolo 3

Implementazione

In questo capitolo sarà illustrata la fase implementativa della web app, analizzando tutte le funzionalità sviluppate e descrivendo nel dettaglio come sono state utilizzate, con specifici riferimenti al codice sorgente. Saranno poi mostrate le problematiche sorte durante lo sviluppo, presentando anche le rispettive soluzioni. Infine, saranno discussi possibili miglioramenti, applicabili a una futura versione della web app.

3.1 Logica dell'applicazione

In questa sezione saranno illustrati tutti i dettagli implementativi del progetto. Si descriverà per prima la struttura del file *index.html*, in cui si inizializzano gli elementi necessari alla produzione della realtà aumentata. Si passerà poi al file *app.js*, mostrando l'implementazione di tutte le funzionalità della web app. Nello specifico, saranno analizzate la connessione a Firebase e l'estrapolazione dei dati, per la creazione degli elementi di realtà aumentata. In seguito, si discuterà l'interazione con gli oggetti di realtà aumentata, spiegando nello specifico come è implementata la visualizzazione della raccolta personale dell'utente. Infine, sarà presentata

la gestione delle ammoniti quando il segnale GPS non è sufficientemente preciso.

3.1.1 Pagina HTML

Il file *index.html* è strutturato seguendo una logica piuttosto tradizionale. La prima intestazione, *<!DOCTYPE html>*, definisce la tipologia di documento che si andrà a scrivere; il tag successivo, *<html>*, caratterizza e contiene l'intera pagina. Al suo interno si inseriscono poi il tag *<head>*, in cui si definiscono le specifiche necessarie alla creazione del documento, e il tag *<body>*, che rappresenta il vero e proprio corpo del documento.

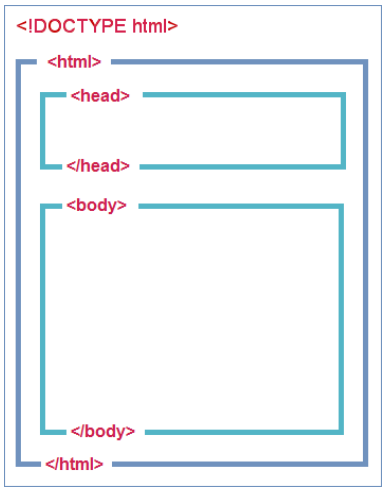

Figura 3.1: Struttura di una pagina HTML.

head

Nell'*head* sono utilizzate quattro tipologie di tag differenti, come mostrato in Figura 3.2.
Il primo tag inserito, *title*, permette di impostare il titolo che la pagina web dovrà avere.
Si utilizza poi il tag *meta*, grazie al quale si specifica la codifica scelta

(UTF-8), con cui il browser genererà la pagine. Sempre richiamando lo stesso tag si imposta la larghezza della pagina, settata in queso caso pari alla larghezza del dispositivo che la sta visualizzando, oltre che lo zoom iniziale fissato a 1.
Tramite il tag *link* si richiama un file esterno, di stile: *style.css*.
Infine sono si utilizzano diversi tag *script*, per inserire degli script all'interno del documento HTML. Nello specifico, attraverso l'attributo *src* si indicano i file dai quali recuperare tali script. Tra questi, tutti file JavaScript, due sono URL recuperati dalla rete, mentre gli altri sono memorizzati direttamente nella cartella di progetto. Entrando nel dettaglio: alla *riga 11* si inserisce la libreria *A-Frame*, alla *riga 12* la libreria di *AR.js* dedicata alle componenti *three.js*, alla *riga 13*, invece, quella dedicata alle componenti di *A-Frame*. Infine, alla *riga 15* viene indicato il file *app.js*, dove sono definite le funzioni dedicate alla web app in discussione.

```
3   <head>
4       <title>AR.js A-Frame Location-based</title>
5
6       <meta charset="UTF-8">
7       <meta name="viewport" content="width=device-width, initial-scale=1">
8
9       <link rel="stylesheet" type="text/css" href="style.css">
10
11      <script src="https://aframe.io/releases/1.0.4/aframe.min.js"></script>
12      <script type='text/javascript' src='https://raw.githack.com/AR-js-org/AR.js/master/three.js/build/ar-threex-location-only.js'></script>
13      <script type='text/javascript' src='aframe-ar.js'></script>
14
15      <script type="module" src="app.js"></script>
16  </head>
```

Figura 3.2: Screenshot del tag *head*.

body

La struttura generale del tag *body* è mostrata in Figura 3.3.
Il primo tag presente è *a-scene*, all'interno del quale sono contenuti tutti gli elementi da inserire poi nella scena virtuale. Nello specifico, sono settati alcuni attributi per rendere migliore la fruizione dei contenuti:

cursor e *raycaster*, per esempio, permettono di rilevare un click effettuato dall'utente su un elemento di tipo *a-entity*.
Il primo tag contenuto in *a-scene* è *a-camera* ed è caratterizzato da alcuni attributi, come *smoothingFactor* e *gps-projected-camera*.
Lo *smoothingFactor* rappresenta un fattore di *AR.js* che regola quanto velocemente deve variare la posizione di un oggetto, in seguito alla rotazione del dispositivo. Essenzialmente, è il coeficiente k all'interno della formula:

$$smoothedAngle = k * newValue + (1 - k) * previousSmoothedAngle.$$

È evidente che, influendo sull'importanza di un angolo di rotazione rispetto a quello precedente, k rende più o meno fluido lo spostamento degli oggetti. Minore è il valore di k, maggiore sarà il livello di levigatura.
Un'altra proprietà del tag *a-camera* è *gps-projected-camera*. Questo indica la versione della componente *a-camera* che si intende utilizzare, tra quelle messe a disposizione da *AR.js*. A sua volta, permette di settare tre propri attributi: *gpsMinDistance*, *positionMinAccuracy* e *alert*. Il primo specifica quanto deve spostarsi (in metri) l'utente prima che la sua posizione venga aggiornata, mentre *positionMinAccuracy* puntualizza la minima precisione del GPS per far funzionare la web app. Infine *alert* impostato a *true*, quando l'accuratezza del GPS è sotto la soglia impostata con *positionMinAccuracy*, permette di mostrare un alert avvertendo l'utente che la precisione del GPS è troppo bassa.
I tag successivi non sono finalizzati alla produzione di contenuti di realtà aumentata, ma classici elementi HTML utilizzati per mostrare informazioni sullo schermo. Una descrizione più accurata sarà illustrata nelle sezioni che seguono.

```
<body>
    <a-scene cursor="rayOrigin: mouse; fuseTimeout: 0" raycaster="objects: a-entity" id="scene" vr-mode-ui='enabled: false'
             arjs='sourceType: webcam; videoTexture: true; debugUIEnabled: false' renderer='antialias: true; alpha: true'>

        <a-camera id='camera1' look-controls-enabled='false' arjs-look-controls='smoothingFactor: 0.1' far=30
                  gps-projected-camera='gpsMinDistance: 1; positionMinAccuracy: 5; alert: true'></a-camera>

        <button id="listButton">Lista</button>

        <div class="message" id="message" style="display: none"></div>

        <div id="info-box" style="display: none">···
        </div>

        <li id="li_structure" style="display: none">···
        </li>

        <div id="list" style="display: none">···
        </div>

    </a-scene>
</body>
```

Figura 3.3: Screenshot del tag *body*.

3.1.2 Configurazione Firebase

In questa sezione sarà analizzata la parte iniziale del file *app.js*, dedicata alla configurazione della connessione a Firebase e del login.
Per prima cosa, si effettuano degli *import* di tutte le funzioni utilizzate per interfacciarsi con Firebase e i suoi prodotti. Il primo permette l'inizializzazione della configurazione, mentre gli altri importano le funzioni per interagire, rispettivamente, con: il database Firestore, l'Autenticathion e lo Storage. Grazie a questi *import* sarà possibile richiamare tutte le funzioni inserite tra parentesi graffe, estrapolate dai file specificati dopo il *from*; in questo caso particolare, provengono tutte da URL messi a disposizione proprio da Firebase.

```
import { initializeApp } from "https://www.gstatic.com/firebasejs/9.17.1/firebase-app.js";
import { getFirestore, collection, getDocs, doc, getDoc, updateDoc, arrayUnion, setDoc }
    from "https://www.gstatic.com/firebasejs/9.17.1/firebase-firestore.js";
import { getAuth, signInWithEmailAndPassword} from "https://www.gstatic.com/firebasejs/9.17.1/firebase-auth.js";
import { getStorage, ref, getDownloadURL } from "https://www.gstatic.com/firebasejs/9.17.1/firebase-storage.js";
```

Figura 3.4: Screenshot degli *import*.

Si definisce poi una variabile *firebaseConfig*, nella quale inserire tutti dati

necessari alla configurazione a Firebase. Questa, nello specifico, permette di identificare il progetto Firebase al quale si vuole accedere e con cui interagire.

```
7   //CONFIGURAZIONE FIREBASE
8   const firebaseConfig = {
9     apiKey: "AIzaSyAdRCwgQEkShwLFWx8lkJ0AffDlchwgN1I",
10    authDomain: "ammoniti-di-strada.firebaseapp.com",
11    projectId: "ammoniti-di-strada",
12    storageBucket: "ammoniti-di-strada.appspot.com",
13    messagingSenderId: "163593434920",
14    appId: "1:163593434920:web:d58241746bea65c310c21d",
15    measurementId: "G-CLYR7V8DLH"
16  };
```

Figura 3.5: Screenshot di *firebaseConfig*.

Successivamente, si effettua l'inizializzazione della connessione attraverso la funzione *inizializeApp*, passando come parametro *firebaseConfig* ed assegnandola alla variabile *app*.
A questo punto si definiscono ulteriori tre variabili, utilizzate in seguito, per l'accesso ai vari prodotti. La prima è *db*: si inserisce qui il riferimento al database Firestore, grazie alla funzione *getFirestore*, con parametro *app* settata precedentemente. Si specifica poi il riferimento per accedere al prodotto Authentication, tramite la funzione *getAuth*, assegnandolo alla variabile *auth*; analogamente si definisce *storage* per l'accesso allo Storage del progetto.

```
18  //INIZIALIZZAZIONE
19  const app = initializeApp(firebaseConfig);
20  const db = getFirestore(app);
21  const auth = getAuth();
22  const storage = getStorage();
```

Figura 3.6: Screenshot dell'*inizializzazione*.

Si configura a questo punto un login. Non essendo presente una pagina dedicata, si verrà sempre identificati con un utente creato appositamente per utilizzare la web app.
Il login avviene attraverso la funzione *signInWithEmailAndPassword*, come mostrato in Figura 3.7, che utilizza email e password come parametri, insieme alla variabile *auth* precedentemente settata. Dopo il login si aggiunge un controllo, per verificare che le credenziali dell'utente siano già nella raccolta *webApp* del database; al primo accesso, non essendo presenti, si andrà a definire un nuovo utente.
In sostanza, si estrapola l'uid dell'utente loggato dalla variabile *user* e si ricerca nella raccolta *webApp*, attraverso la funzione *getDoc*. Quest'ultima riceve come riferimento il documento cercato attraverso la funzione *doc*, che lavora sui parametri: *db*, nome della raccolta e id del documento da cercare. Infine, attraverso la condizione *if* alla *riga 29*, si verifica la presenza dell'utente nella raccolta. In caso contrario, si entra nell'*if* ed si aggiunge tramite la funzione *setDoc*.

```
24  //LOGIN
25  signInWithEmailAndPassword(auth, "webapp@gmail.com", "webApp")
26    .then(async (userCredential) => {
27      const user = userCredential.user;
28      const userRef = await getDoc(doc(db, "webApp", user.uid));
29      if (!userRef.exists()) {
30        try {
31          const docRef = await setDoc(doc(db, "webApp", user.uid), {
32          });
33        } catch (e) {
34          console.error("Error adding document: ", e);
35        }
36      }
37    })
38    .catch((error) => {
39      const errorCode = error.code;
40      const errorMessage = error.message;
41      console.log(errorCode);
42      console.log(errorMessage);
43    });
```

Figura 3.7: Screenshot del *login*.

3.1.3 Visualizzazione ammoniti

Dopo aver configurato il database ed effettuato il login, si devono caricare i fossili per renderli fruibili tramite realtà aumentata.
Nello specifico, si inizializza una struttura dati che li possa contenere, una *Map* denominata *ammonitiMap*. Si estrapolano poi tutte le ammoniti presenti nella raccolta *dettaglio*, mediante le funzioni *getDocs* e *collection*. In seguito, attraverso un *forEach*, si cicla sulle ammoniti: per ognuna si aggiunge l'elemento nella *ammonitiMap* e si richiama la funzione *createElement*, passando come parametro il fossile corrente, per creare l'entità da mostrare.

```
47  var ammonitiMap = new Map();
48  const ammoniti = await getDocs(collection(db, "dettaglio"));
49
50  ammoniti.forEach((ammonite) => {
51    ammonitiMap.set(ammonite.id, ammonite.data());
52    createElement(ammonite);
53  });
```

Figura 3.8: Screenshot dell'estrazione delle ammoniti dal database.

createElement

Inizialmente si definisce una costante, *entity*, alla quale si assegna un nuovo elemento *a-entity* creato con il comando *document.createElement*. Quest'ultimo è corredato da alcuni attributi: *id* per l'id del fossile, *gltf-model* per il modello 3D che deve essere riprodotto, *scale* per la dimensione e *gps-projected-entity-place* per le coordinate dell'ammonite. In questa prima versione della web app, il modello è unico per tutte le ammoniti della raccolta. Oltre a questi attributi, si associa la funzione *clickedAmm* al verificarsi del click sull'ammonite (*riga 75*), che si discuterà più in dettaglio nella Sezione 3.1.4.

In seguito, per mezzo di un'istruzione condizionale, si controlla la visualizzazione del fossile: se al momento della creazione è presente nella pagina l'alert della scarsa precisione del GPS (elemento con id *alert-popup*), si imposta l'entità come "non visibile" per non essere mostrata. Infine, attraverso la funzione *appendChild*, si aggiunge l'elemento appena creato al documento. In questo modo, quando l'utente sarà sufficientemente vicino ad un'ammonite (distanza impostata tramite l'attributo *far* nel tag *a-camera*), il fossile sarà mostrato sullo schermo.

```
55  async function createElement(ammonite) {
56    const entity = document.createElement("a-entity");
57    entity.setAttribute('id', ammonite.id);
58    entity.setAttribute('gltf-model', 'url(gltf/ammonite1/scene.gltf)');
59    //COORDINATE IN ZONE
60    const zona = await getDoc(doc(db, "zone", ammonite.data().zona));
61    entity.setAttribute('gps-projected-entity-place', {
62      latitude: zona.data().lat,
63      longitude: zona.data().long
64    });
65    //COORDINATE IN DETTAGLIO
66    /*entity.setAttribute('gps-projected-entity-place', {
67      latitude: ammonite.data().lat,
68      longitude: ammonite.data().long
69    });*/
70    entity.setAttribute('scale', {
71      x: 3,
72      y: 3,
73      z: 3
74    });
75    entity.onclick = function () { clickedAmm(ammonite.id) };
76    if (document.getElementById("alert-popup") !== null) { entity.setAttribute("visible", false) };
77    document.querySelector("a-scene").appendChild(entity);
78  }
```

Figura 3.9: Screenshot della funzione *createElement*.

3.1.4 Interazione con ammoniti

Quando l'utente esegue un click sull'ammonite si attiva la funzione *clickedAmm*, che riceve l'id del fossile in questione. Per prima cosa, si assegna la costante *distance* al risultato della funzione *checkDistance*, che restituisce la distanza dell'utente dall'ammonite. Successivamente si effettua

un controllo: se la distanza è minore di cinque metri, si richiamano le funzioni *checkList*, che verifica se il fossile è presente nella sua raccolta personale, e *showInfoBox*, che mostra sullo schermo le informazioni corrispondenti. Quando invece la distanza è maggiore o uguale a cinque metri, si richiama la funzione *showMessage*, avvertendo l'utente di avvicinarsi all'ammonite e specificando la distanza.

```
131  function clickedAmm(id) {
132    const distance = checkDistance(id).toFixed(2);
133    if (distance && distance < 5) {
134      checkList(id);
135      showInfoBox(id);
136    } else
137      showMessage("L'oggetto è distante " + distance + "m, avvicinati di più", 5000);
138  }
```

Figura 3.10: Screenshot della funzione *clickedAmm*.

checkDistance

Per verificare la distanza, si selezionano gli elementi *camera* e quello con *id* corrispondente al parametro; successivamente si estrapolano i dati relativi alla posizione geografica, utilizzando il metodo *object3D.position*. Infine si calcola la differenza, restituendo il risultato alla funzione *clickedAmm*.

```
140  function checkDistance(id) {
141    const ammoniteEntity = document.getElementById(id);
142    const camera = document.querySelector('a-camera');
143    let cameraPositon = camera.object3D.position;
144    let ammonitePosition = ammoniteEntity.object3D.position;
145    let distanceToCamera = cameraPositon.distanceTo(ammonitePosition);
146    return distanceToCamera;
147  }
```

Figura 3.11: Screenshot della funzione *checkDistance*.

checkList

La funzione *checkList* riceve l'id dell'ammonite e verifica la sua presenza nella lista della raccolta dell'utente. Nello specifico, accede al documento della raccolta *webApp* relativa allo user loggato ed estrae la lista delle ammoniti, assegnandole alla costante *list*. Successivamente, verifica se la lista esiste e se l'id dell'ammonite è presente in *list*. In caso affermativo, entra nell'*if* e, attraverso la funzione *showMessage*, avverte l'utente che l'ammonite fa già parte della sua raccolta. In caso contrario, attraverso il metodo *updateDoc*, aggiorna la lista sul database: aggiunge l'id dell'ammonite selezionato alla raccolta dell'utente e, sempre richiamando *showMessage*, avvisa che è stato aggiunto alla raccolta.

```
149  async function checkList(id) {
150    const listRef = doc(db, "webApp", auth.currentUser.uid);
151    const listSnap = await getDoc(listRef);
152    const list = listSnap.data().ammoniti;
153    if (list && list.includes(id))
154      showMessage("Fa già parte della tua collezione", 5000);
155    else {
156      updateDoc(listRef, { ammoniti: arrayUnion(id) });
157      showMessage("Aggiunto alla tua collezione!", 5000);
158    }
159  }
```

Figura 3.12: Screenshot della funzione *checkList*.

showInfoBox

La funzione in questione ha lo scopo di mostrare le informazioni dell'ammonite cliccata. In particolare, si sfrutta una struttura HTML già presente ma non visibile grazie alla regola di stile *display: none*. Questa dovrà essere popolata con delle informazioni, poi resa visibile all'utente.

La struttura in questione è mostrata nella Figura 3.13 ed è formata da due sezioni: una per contenere l'immagine e un'altra per contenere il nome e la descrizione.

```
<div id="info-box" style="display: none">
    <div class="image-info">
        <img id="img"></img>
    </div>
    <div class="description-info">
        <h1 id="title-info"></h1>
        <p id="description-info"></p>
    </div>
</div>
```

Figura 3.13: Screenshot dell'elemento *info-box*.

Per quanto riguarda la funzione, si estrapola dal database il nome della foto (presente nello Storage) relativa all'ammonite selezionata. Grazie al metodo *getDownloadURL*, si restituisce l'URL per ottenere l'immagine dallo Storage e si assegna all'attributo *src* dell'elemento con id *img*, cioè il contenitore dell'immagine nel box. Sfruttando poi le informazioni in *ammonitiMap*, si riempono i campi del titolo e della descrizione. Infine, si rende visibile il box modificando la regola di stile in *display: flex*.
È possibile nascondere poi il box: cliccandolo, lo stile si imposta nuovamente su *display: none*.

```
161 function showInfoBox(id) {
162   getDownloadURL(ref(storage, "foto dettaglio/" + ammonitiMap.get(id).fotofossile)).then(function (url) {
163     document.querySelector('#img').src = url;
164   })
165   document.querySelector('#title-info').innerHTML = id;
166   document.querySelector('#description-info').innerHTML = ammonitiMap.get(id).descrfossile;
167   document.getElementById("info-box").style.display = "flex";
168 }
```

Figura 3.14: Screenshot della funzione *showInfoBox*.

showMessage

La funzione riceve come parametri il testo e il tempo, corrispondente alla visualizzazione sul display. Anche in questo caso, si utilizza un elemento già esistente con il messaggio specifico e lo si rende visibile. Successivamente, attraverso la funzione *setTimeout* con il tempo come parametro, si rende l'elemento nuovamente non visibile.

```
170  function showMessage(text, time) {
171    const messagebox = document.getElementById("message");
172    messagebox.innerHTML = text;
173    messagebox.style.display = "block";
174    setTimeout(function () {
175      messagebox.style.display = "none";
176    }, time);
177  }
```

Figura 3.15: Screenshot della funzione *showMessage*.

3.1.5 Visualizzazione lista

Questa funzione, come altre descritte precedentemente, sfrutta due strutture già presenti, mostrate in Figura 3.16. Una rappresenta la pagina in cui visualizzare la lista, mentre l'altra, il tag *li*, è un utilizzata per creare gli elementi della lista. Quest'ultima è formata da due sezioni: la prima con foto e nome, mentre la seconda, inizialmente nascosta, contiene informazioni aggiuntive e viene mostrata quando si esegue un click sul nome.

Per aprire la lista si crea un evento che, al click sul bottone in alto a destra, richiamata la funzione *createList*: questa popola la lista e la rende poi visibile. Anche quando la lista è visibile, è presente il bottone per chiuderla: in questo caso si richiama la funzione *removeList*, che semplicemente elimina tutti gli elementi presenti e la rende non visibile.

```
<li id="li_structure" style="display: none">
    <div class="li_list">
        <img class="image-info" style="width: 20%"></img>
        <p class="nome"></p>
    </div>
    <div class="hidden_info" style="display: none">
        <p class="zona"></p>
        <p class="materiale"></p>
        <p class="descrizione"></p>
    </div>
</li>

<div id="list" style="display: none">
    <button id="closeList">Chiudi</button>
    <h1>LA TUA RACCOLTA</h1>
    <ul></ul>
</div>
```

Figura 3.16: Screenshot degli elementi che generano la lista.

createList

La funzione inizialmente estrapola tutta la lista delle ammoniti presenti nella raccolta personale dell'utente. A questo punto, attraverso un ciclo *forEach*, per ogni fossile crea l'elemento nella lista.
Nello specifico, clona la struttura *li* mostrata in Figura 3.16 attraverso il metodo *cloneNode*. Successivamente, si estraggono dal database i documenti relativi alla zona e al materiale dell'ammonite, oltre che l'URL della foto come fatto in precedenza. Questi dati vengono utilizzati per compilare l'elenco, inserendo foto e nome nella parte visibile, mentre la parte nascosta viene riempita con nome della zona, materiale e descrizione dell'ammonite. Dopodichè si definisce una funzione che, al click sul nome, verifica se la parte con le informazioni è mostrata oppure nascosta. Se risulta nascosta imposta lo stile in *display: block* e la mostra; in caso contrario, la rende non visibile associando lo stile *display: none*. Infine, attraverso il metodo *appendChild*, ogni elemento viene aggiunto alla lista.

```
async function createList() {
  const listRef = doc(db, "webApp", auth.currentUser.uid);
  const listSnap = await getDoc(listRef);
  const list = listSnap.data().ammoniti;
  list.forEach(async element => {
    const clone = document.getElementById("li_structure").cloneNode(true);
    clone.style.display = "block";
    const zona = await getDoc(doc(db, "zone", ammonitiMap.get(element).zona));
    const materiale = await getDoc(doc(db, "materiali", ammonitiMap.get(element).materiale));
    getDownloadURL(ref(storage, "foto dettaglio/" + ammonitiMap.get(element).fotofossile)).then(function (url) {
      clone.querySelector('img').src = url;
    })
    clone.querySelector(".nome").innerHTML = element;
    clone.querySelector(".zona").innerHTML = "ZONA: " + zona.data().nomezona;
    clone.querySelector(".materiale").innerHTML = "MATERIALE: " + materiale.data().nome;
    clone.querySelector(".descrizione").innerHTML = "DESCRIZIONE: " + ammonitiMap.get(element).descrfossile;
    clone.onclick = function () {
      if (clone.querySelector(".hidden_info").style.display == "none")
        clone.querySelector(".hidden_info").style.display = "block";
      else clone.querySelector(".hidden_info").style.display = "none";
    };
    document.querySelector("ul").appendChild(clone);
  });
}
```

Figura 3.17: Screenshot della funzione *createList*.

3.1.6 Gestione GPS

Oltre alla verifica sulla precisione del GPS, effettuata con la creazione dell'elemento che rappresenta l'ammonite, si predispone un controllo durante il funzionamento dell'app. Infatti, AR.js mostra un alert sullo schermo quando la precisione del segnale è sotto la soglia scelta. La web app sfrutta proprio questo messaggio: al verificarsi dell'evento *DOMNodeInserted*, cioè quando si inserisce un nuovo nodo nel documento, si controlla se questo ha id *alert-popup* (ovvero l'alert creato da AR.js); in caso affermativo, si selezionano tutti gli elementi *a-entity* rendendoli non visibili, settando *visible* a *false*. Così facendo, quando il GPS scende sotto la soglia minima, tutte le ammoniti vengono nascoste dallo schermo. Analogamente, con l'evento *DOMNodeRemoved*, alla rimozione dell'alert si reimpostano tutti gli oggetti come visibili.

```
84  document.addEventListener("DOMNodeInserted", function (e) {
85    if (e.target.id == "alert-popup") {
86      const entities = document.querySelectorAll("a-entity");
87      entities.forEach(element => {
88        element.setAttribute("visible", false);
89      });
90    };
91  });
92
93  document.addEventListener("DOMNodeRemoved", function (e) {
94    if (e.target.id == "alert-popup") {
95      const entities = document.querySelectorAll("a-entity");
96      entities.forEach(element => {
97        element.setAttribute("visible", true);
98      });
99    };
100 });
```

Figura 3.18: Screenshot delle fuzioni che gestiscono l'accuratezza del GPS.

3.2 Problematiche e soluzioni

Durante lo sviluppo della web app, più di una volta si sono verificati problemi sul corretto funzionamento delle varie versioni.
Il primo problema, incontrato nelle fasi iniziali di implementazione, è stato lo spostamento repentino che l'oggetto 3D mostrava quando inquadrato con il dispositivo. Questo era causato dalle tante correzioni che il GPS esegue automaticamente, cercando di individuare la posizione più precisa possibile. In questo modo, anche la posizione del dispositivo rispetto al fossile si aggiornava continuamente, spostando ogni volta l'ammonite. Non è stato possibile risolvere del tutto il problema, in realtà, ma l'effetto visivo è stato migliorato significativamente modificando alcuni parametri nel tag HTML *a-camera*. Più nello specifico, si è lavorato sullo *smoothingFactor*, presentato in precedenza, e sul *gpsMinDistance*. La soluzione che minimizzava il più possibile il problema assegna allo *smoothingFactor* il valore 0.1 e al *gpsMinDistance* 1. Essenzialmente, lo

spostamento minimo per far aggiornare la posizione è un metro.
Un altro problema si riscontrava durante l'avviamento della web app: in quel momento il GPS non raggiungeva mai una precisione sufficiente. In questo caso, a tutte le ammoniti veniva associata la posizione *0, -1.6, 0*, cioè esattamente 1,6 metri sotto il dispositivo: essendo la precisione del GPS troppo bassa, le coordinate geografiche non erano effettivamente lette ma assegnate di default. Per risolvere il problema, come illustrato nella Sezione 3.1.3, si effettua una verifica della presenza dell'alert per scarsa precisione del GPS, nascondendo le ammoniti. Quando si torna in una zona con buona precisione, la libreria assegna in automatico le coordinate e, con le funzioni presentate nella Sezione 3.1.6, le ammoniti vengono impostate nuovamente come visibili.
Inoltre, le ammoniti inizialmente riprodotte tramite realtà aumentata non risultavano cliccabili: questo causava errori nell'implementazione delle funzioni di interazione. Per poter rilevare il click sui fossili, sono stati aggiunti degli attributi al tag *a-scene*. In particolare, sono stati settati gli attributi *cursor="rayOrigin: mouse; fuseTimeout: 0"* e *raycaster="objects: a-entity"*. Il primo permette di leggere i click, mentre il secondo restringe la rilevazione a quelli effettuati su elementi *a-entity*.
Infine, si è presentato un problema con la libreria AR.js, che veniva importata ma restituiva degli errori e non funzionava. È stato risolto grazie alla repository GitHub della libreria, dalla quale era importata, e grazie alla possibilità di accedere alle versioni vecchie messe a disposizione dalla piattaforma. Scegliendo la versione precedente e caricandola direttamente nella cartella di progetto, con il nome *aframe-ar.js*, tutte le funzionalità sono state ripristinate.

3.3 Miglioramenti

Di seguito saranno elencati alcuni possibili miglioramenti che possono essere implementati per la web app in futuro.
Il primo sviluppo suggerito consistere nel trovare dei modelli 3D più leggeri degli attuali, in modo da rendere la fruizione del contenuto più fluida ed appesantire meno il funzionamento dell'applicazione.
Si potrebbe poi lavorare per rendere più accattivante la grafica, specialmente per quanto riguarda la visualizzazione della raccolta personale oppure il box delle informazioni di un'ammonite.
Per correggere ulteriormente il problema, presentato precedentemente, degli spostamenti bruschi delle ammoniti, si potrebbe pensare lavorare sul valore di *gpsMinDistance*. Rendendo questa quantità variabile e fissandola ad un valore maggiore dell'effettiva correzione del GPS, quando l'utente è in prossimità dell'ammonite, la posizione non verrebbe aggiornata e che l'ammonite non si muoverebbe.
Un'alternativa, più sofisticata e soprattutto molto più efficace, sarebbe ancorare l'ammonite in un punto preciso dopo il primo piazzamento e sfruttare i sensori del dispositivo e la fotocamera per rilevare gli spostamenti, senza più basarsi sulle coordinate GPS.
Un'ultima possibilità potrebbe riguardare l'aggiornamento della posizione dell'ammonite: anzichè spostarsi direttamente da una posizione all'altra, si potrebbe tener conto di step intermedi per rendere lo spostamento più fluido.
Infine, si potrebbe implementare una variazione del modello 3D da proiettare, che cambi in base a qualche informazione sul database, aggiuntiva o tra quelle già presenti, come ad esempio il nome oppure il materiale.

Conclusioni

In questo elaborato è stato analizzato il processo che ha portato alla progettazione e all'implementazione di una web app, utilizzata per la fruizione mediante realtà aumentata delle ammoniti presenti nelle strade della città di Ancona. Si è partiti da una fase di analisi, nella quale è stata illustrata l'applicazione "Ammoniti di strada", di cui la web app è un'estensione. In questa sezione sono stati elencati i requisiti funzionali e non funzionali della web app, ed è stato presentato il concetto della realtà aumentata e molte delle sue possibili applicazioni.
Successivamente, si è passati alla trattazione della fase di progettazione, illustrando tutti gli strumenti necessari allo sviluppo. In seguito è stata analizzata la struttura del database, soffermandosi maggiormente sui dati effettivamente utili alla web app. Infine, è stata presentata la struttura vera e propria del progetto.
Nell'ultimo capitolo è stata illustrata la fase implementativa, discutendo nel dettaglio come le varie funzionalità sono state implementate. Infine, si è parlato delle problematiche riscontrate durante lo sviluppo, la loro risoluzione ed i possibili miglioramenti futuri.
La web app risulterebbe molto utile, soprattutto se migliorata in alcuni aspetti, a rendere più piacevole la scoperta dei reperti nella città, rappresentando uno strumento con cui ampliare la proprio conoscenza, anche grazie alle informazioni condivise con l'applicazione "Ammoniti di stra-

da".

Nel complesso, lo sviluppo della web app può ritenersi completato e riuscito. In primo luogo, tutti i requisiti previsti sono stati rispettati, salvo alcuni piccoli problemi tuttavia risolvibili. Inoltre, è risultato molto istruttivo e funzionale, a livello personale, soprattutto per quanto riguarda l'approccio e l'utilizzo della realtà aumentata. Questa tecnologia sta vedendo una diffusione sempre più rapida ed ampia, e con tutta probabilità ricoprirà un ruolo cruciale in molti settori, in un'ottica futura. Averla già trattata ed incontrata rappresenta un punto di inizio, oltre che di crescita personale, piuttosto significativo.

Appendice A

Screenshot della web app

Si riportano di seguito le schermate della web app, nella versione presentata in questo elaborato, registrate durante un test eseguito post implementazione, per verificare le funzionalità e le risoluzioni ai problemi incontrati precedentemente.

Figura A.1: Alert precisione GPS inferiore alla soglia impostata.

Figura A.2: Visualizzazione di un'ammonite vicina.

Figura A.3: Click su un'ammonite presente nella lista.

Figura A.4: Click su un'ammonite troppo lontana.

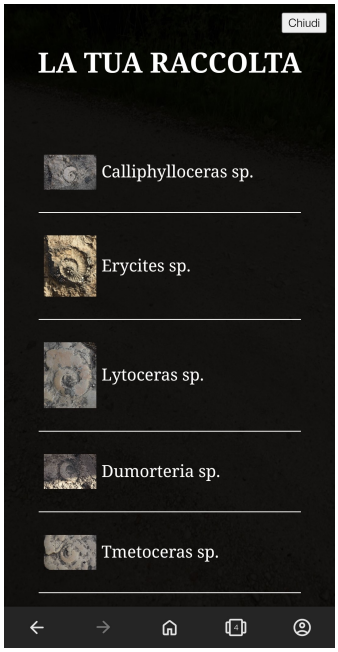

Figura A.5: Visualizzazione della raccolta personale.

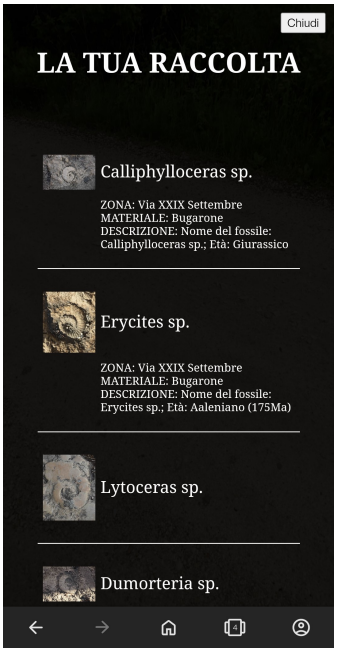

Figura A.6: Lista con maggiori informazioni.

Bibliografia

[1] Cenni storici della realtà aumentata. https://www.adobe.com/it/products/substance3d/discover/what-is-ar.html.

[2] Tipologie di realtà aumentata. https://digitalpromise.org/initiative/360-story-lab/360-production-guide/investigate/augmented-reality/getting-started-with-ar/types-of-ar/.

[3] Tipologie di realtà aumentata. https://www.nextechar.com/blog/what-are-the-different-types-of-augmented-reality.

[4] HTML. https://it.wikipedia.org/wiki/HTML.

[5] CSS. https://www.html.it/guide/guida-css-di-base/.

[6] JavaScript. https://www.javascript.com/.

[7] Visual Studio Code. https://code.visualstudio.com/.

[8] Documentazione AR.js. https://ar-js-org.github.io/AR.js-Docs/.

[9] Firebase. https://firebase.google.com/.

[10] Google Chrome. https://www.google.com/intl/it_it/chrome/.

[11] GitHub. https://github.com/.

[12] Prodotti Firebase. https://firebase.google.com/products-build.