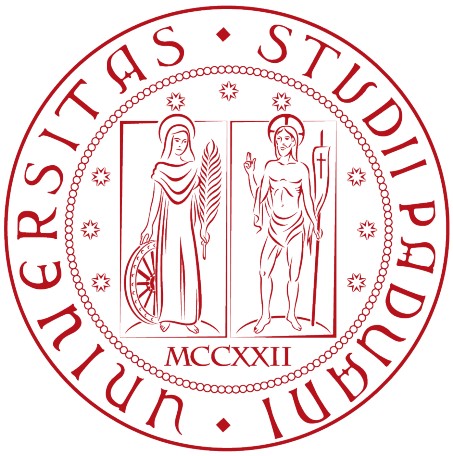
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea Triennale in Informatica



Gestione asset per gemello digitale immerso in realtà aumentata

Tesi di laurea

Relatore
Prof. Tullio Vardanega

Laureando
Bellò Marco

Anno Accademico 2022-2023

Sommario

Il presente documento è il resoconto della mia attività di tirocinio, della durata di circa trecento ore, presso l'azienda Datasoil S.r.l.
La maggioranza dei programmi oggi distribuiti sono applicazioni mobile, il cui mercato è diviso tra due sistemi operativi concorrenti e incompatibili: iOS e Android. Questo dualismo si traduce nella necessità di sviluppare ogni applicativo due volte, il che ha favorito la nascita di framework come Flutter: esso permette di scrivere applicazioni frontend in Dart, che traduce poi autonomamente nelle rispettive versioni native (condivide quindi l'ethos di Java "Write Once, Run Anywhere").
Una tecnologia ancora poco esplorata in campo mobile è quella della realtà aumentata, nonostante le "Big Five", ovvero Google (Alphabet), Amazon, Facebook (Meta), Apple e Microsoft, investano tutte nel campo dell'augmented reality (AR) e alcune (Meta e Microsoft) producano anche hardware specifico (Quest e HoloLens rispettivamente). L'integrazione di queste tecnologie è però limitata a SDK e framework già consolidati, come Unity.
Questa tesi studia le possibilità di integrare una vista AR in un'applicazione di auditing aziendale (preesistente) sviluppata in Flutter, tramite i "method channels" che consentono di innestare codice nativo (come Swift o Java) in Dart.
A seguito di uno studio delle (poche) tecnologie esistenti, mi sono concentrato sul comprendere il funzionamento del plugin scelto e del pacchetto di API di cui esso si serve (Sceneform) per poi integrare al suo interno le Azure Spatial Anchors di Microsoft, necessarie ad agganciarsi alle componenti geospaziali già presenti nell'applicazione sulla quale ho svolto il lavoro di tesi.
Il risultato ottenuto è soddisfacente, soprattutto considerando l'estrema scarsità documentale associata a questo ecosistema tecnologico.

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Tullio Vardanega, relatore della mia tesi, per il supporto preciso e puntuale fornitomi durante l'intera durata dello stage e la conseguente stesura della tesi.

Un ringraziamento va ad Andrea e Pietro, riferimenti aziendali, e in generale a tutto il team Datasoil S.r.l. che mi ha permesso di vivere un'esperienza arricchente in un ambiente lavorativo sereno.

Desidero infine ringraziare i miei familiari senza i quali mai avrei potuto intraprendere un percorso impegnativo come un corso di laurea.

Padova, Febbraio 2023 Bellò Marco

Indice

1 Contesto aziendale **1**
1.1 L'azienda . . . 1
1.1.1 Descrizione . . . 1
1.1.2 Clientela . . . 2
1.1.3 Processi interni . . . 3
1.2 Tecnologie e strumenti . . . 3
1.2.1 Tecnologie e strumenti di sviluppo . . . 3
1.2.2 Strumenti organizzativi . . . 8
1.3 Innovazione, stage e prodotti . . . 10

2 Lo stage **11**
2.1 Strategia aziendale . . . 11
2.2 Descrizione e temi . . . 12
2.3 Obiettivi . . . 14
2.4 Vincoli . . . 14
2.5 Pianificazione . . . 15
2.6 Motivazione scelta . . . 16

3 Realizzazione **20**
3.1 Studio delle tecnologie . . . 20
3.1.1 Realtà aumentata . . . 20
3.1.2 Microsoft Azure . . . 22
3.1.3 Framework realtà aumentata per Flutter . . . 23
3.2 Analisi dei requisiti . . . 28
3.2.1 Requisiti funzionali . . . 29
3.2.2 Requisiti di vincolo . . . 30
3.2.3 Riepilogo requisiti . . . 30
3.3 Progettazione . . . 31
3.3.1 Architettura . . . 31
3.3.2 Interfaccia utente . . . 31
3.4 Codifica . . . 32
3.5 Verifica . . . 37
3.6 Risultati raggiunti . . . 41

4 Valutazione retrospettiva **45**
4.1 Difficoltà incontrate . . . 45
4.2 Raggiungimento obiettivi . . . 46
4.2.1 Obiettivi proponente . . . 46
4.2.2 Obiettivi personali . . . 47
4.3 Competenze . . . 48

Appendici **50**

A Flutter **51**

A.1 Dart 51
A.2 Widget 54
A.3 Method channels 57

Elenco delle figure

1.1 Visualizzazione 3D di impianto produttivo . . . 1
1.2 Metriche di controllo *asset* aziendale . . . 2
1.3 Immagine *home page* Datasoil . . . 2
1.4 Jira *continuous delivery pipeline* . . . 3
1.5 *Integrated development environments* e linguaggi . . . 7
1.6 Azure Spatial Anchors e linguaggi . . . 8
1.7 *Framework*, Flutter e Azure Spatial Anchors . . . 8
1.8 Strumenti di sviluppo . . . 9
1.9 Menù HoloLens . . . 10
1.10 Posizionamento *asset* . . . 10
1.11 HoloLens usato in azienda . . . 10

2.1 Allerte meteo in SYN . . . 11
2.2 *Screenshot* schermate SYN fieldOps . . . 12
2.3 *Asset* . . . 13
2.4 *Card* e *anchor* . . . 13
2.5 *Asset Compressor Intake* e *card* . . . 13
2.6 Confronto *backend* e *frontend* . . . 16
2.7 Equilibrio vita-lavoro . . . 17
2.8 *Market share desktop* e *mobile* . . . 18
2.9 Ikea realtà aumentata . . . 18
2.10 Nintendo realtà aumentata . . . 18
2.11 Ikea e Nintendo usano realtà aumentata . . . 18

3.1 Ikea realtà aumentata *in-app* . . . 21
3.2 Piani in realtà aumentata . . . 21
3.3 Ancoraggi singoli per multipli elementi . . . 22
3.4 Azure Portal creazione risorse . . . 22
3.5 Azure Portal Azure Spatial Anchors . . . 23
3.6 Realtà aumentata con ARwayKit . . . 25
3.7 Applicazione realtà aumentata ar_flutter_plugin . . . 26
3.8 Schema ar_flutter_plugin . . . 26
3.9 Architettura *app* . . . 31
3.10 *App mockup* . . . 32
3.11 Creazione, caricamento e recupero di *asset* . . . 42
3.12 *Bottom sheet asset* per eliminazione e lista *asset* . . . 43
3.13 Caricamento *ticket* realtà aumentata e *ticket* su *asset* . . . 44

4.1 Ricerca esatta Flutter e Azure Spatial Anchors 23 novembre . . . 45
4.2 Ricerca esatta Flutter e Azure Spatial Anchors 8 febbraio . . . 46

A.1 I tre alberi di Flutter . . . 54

A.2 *Widget tree* app esempio 55

Elenco delle tabelle

3.1 Confronto *framework* per realtà aumentata 28
3.2 Tabella dei requisiti funzionali 30
3.3 Tabella dei requisiti di vincolo 30
3.4 Numero di requisiti per obbligatorietà 30
3.5 Numero di requisiti per tipologia 30
3.6 Requisiti soddisfatti nei frammenti: 3.1, 3.2, 3.3, 3.4, 3.5. 38
3.7 Requisiti soddisfatti nei frammenti: 3.6, 3.7, 3.8, 3.9, 3.10. 38
3.8 Requisiti soddisfatti nel frammento: 3.11 39
3.9 Requisiti soddisfatti in figura 3.11 39
3.10 Requisiti soddisfatti in figura 3.12 40
3.11 Requisiti soddisfatti in figura 3.13 40
3.12 Requisiti non soddisfatti 40
3.13 Tabella requisiti - chi li soddisfa 41

4.1 Tabella dei requisiti funzionali 47
4.2 Tabella dei requisiti funzionali 47

Elenco dei frammenti di codice

3.1 mobilesyn *asset provider* 32
3.2 mobilesyn *ticket provider* 33
3.3 mobilesyn *asset ticket provider* 33
3.4 mobilesyn *managers* 33
3.5 mobilesyn *on ar view created* 34
3.6 ar_flutter_plugin *create, delete, ulpoad, delete cloud anchor* tramite *method channel* 34
3.7 Android chiamate dei *method channel* per effettuare *create, delete, ulpoad, delete cloud anchor* 35
3.8 iOS chiamate dei *method channel* per effettuare *create, delete, ulpoad, delete cloud anchor* 35
3.9 Eliminazione *cloud anchor* lato Android, chiamata 36
3.10 Eliminazione *cloud anchor* lato Android, chiamata 36
3.11 Frammento di codice per vista in realtà aumenta in Dart 37
A.1 Dart *null safety* 51
A.2 Dart interpolazone stringhe 51

A.3 Dart parametri funzioni 51
A.4 Dart funzioni anonime 52
A.5 Dart espressioni ternarie 52
A.6 Dart operatori '?' e '!' 53
A.7 Dart programmazione asincrona 53
A.8 Codice rilevante per la parte grafica dell'*app* di esempio di Flutter . . 54
A.9 Creazione *stateless widget* 56
A.10 Creazione *hook widget* 56
A.11 Android *manifest* 57
A.12 Java *main activity* 57
A.13 Dart *main* 58

Capitolo 1

Contesto aziendale

1.1 L'azienda

1.1.1 Descrizione

Le informazioni a seguito sono state prese direttamente dall'azienda, tuttavia ritengono siano rappresentative della realtà considerando sia il progetto al quale ho partecipato, sia il resto dell'offerta *software*.
Datasoil S.r.l. è una *startup* che si occupa di sviluppare applicativi dedicati alla gestione aziendale, altamente integrati con industria 4.0 e *digital twin* (controparte digitale di un oggetto reale, ad esempio un macchinario di una linea produttiva): vengono impiegati apprendimento automatico[1] e analisi predittive per ottenere analitiche migliori rispetto alla concorrenza, con un occhio attento a innovazione, sicurezza e scalabilità, ottenute anche delegando la gestione delle componenti *server* a servizi terzi come Amazon Web Services[2].

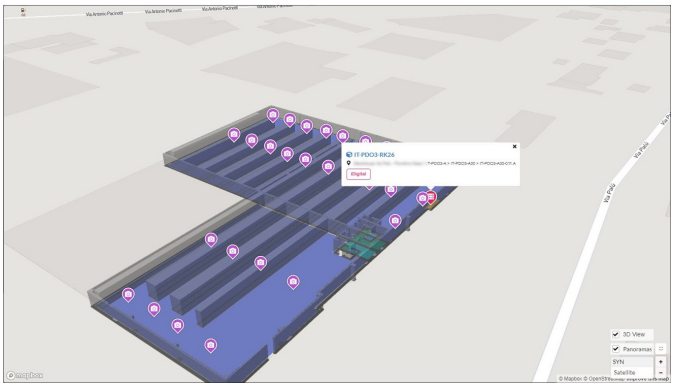


Figura 1.1: Visualizzazione 3D di impianto produttivo con *asset* mappati.[3]

L'obiettivo è manipolare ed elaborare dati di natura inerentemente caotica, ordinarli e presentarli all'utente finale tramite interfacce grafiche intuitive (impiegando tecnologie all'avanguardia sia lato *frontend* che lato *backend*[4]).

[1] Anche detto *machine learning* in inglese è una branca dell'intelligenza artificiale che utilizza metodi statistici per migliorare la performance di un algoritmo nell'identificare pattern nei dati, nella quale in una macchina si predispone l'abilità di apprendere in maniera autonoma(fonte: https://it.wikipedia.org/wiki/Apprendimento_automatico).
[2] Ecosistema *software* fornito da Amazon, che va da semplici piattaforme di *cloud storage* fino a sistemi predittivi basati su *machine learning*
[3] Fonte: https://datasoil.it/industry-40-smartcity-syn-asset-performance/auditing-inventory/
[4] *Fronted* e *backend* denotano, rispettivamente, la parte visibile all'utente di un programma e con cui egli può interagire (tipicamente un'interfaccia utente) e la parte che permette l'effettivo funzionamento

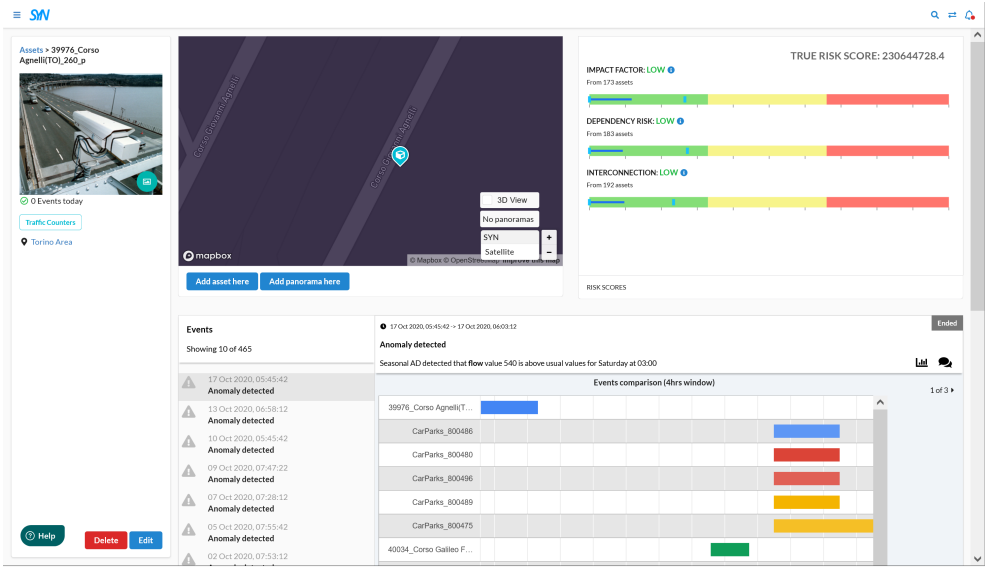


Figura 1.2: *Screenshot* visualizzazione grafica di metriche di controllo per *asset* aziendale.[5]

Ho avuto modo di verificare molte di queste affermazioni direttamente tramite il progetto proposto, in quanto si tratta di un applicativo legato all'industria 4.0 che mappa *asset* aziendali (come ad esempio macchinari) a un gemello digitale, permettendo sia di localizzarli su una mappa tridimensionale sia, obiettivo dello stage, di visualizzarli in realtà aumentata.

1.1.2 Clientela

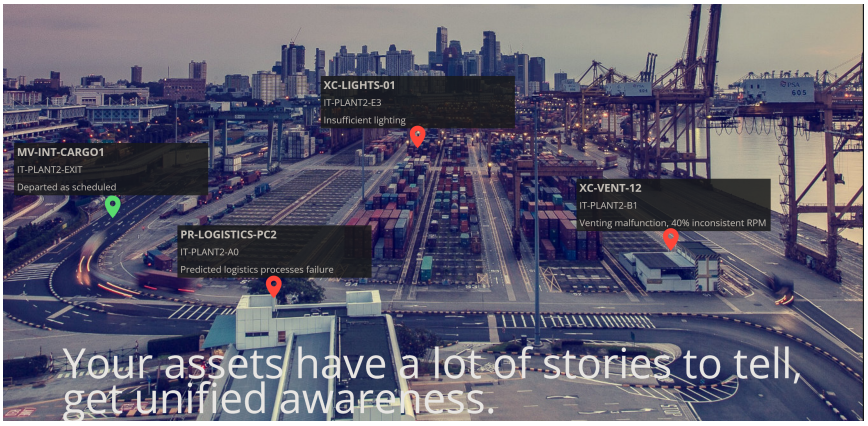


Figura 1.3: L'immagine scelta dall'azienda per il proprio sito mostra una chiara direzione aziendale.[6]

di queste interazioni (fonte: https://it.wikipedia.org/wiki/Front-end_e_back-end).

[5]Fonte: https://synmgr.com

[6]Fonte: https://datasoil.it/industry-40-smartcity-syn-asset-performance/

Si evince da quanto discusso nella sezione precedente che l'offerta di Datasoil sia indirizzata a una clientela professionale, o meglio, aziendale, piuttosto che al mondo *consumer*: l'applicazione sulla quale ho svolto lo Stage (MobileSYN) si occupa di gestione e controllo per *asset* aziendali, e un altro prodotto da loro sviluppato, chiamato LifestyleSync, fornisce strumenti di monitoraggio della salute dei dipendenti, con *app* di supporto "LSCoach" che permette a un allenatore di preparare allenamenti personalizzati e verificarne il progresso.
In sintesi Datasoil si focalizza su aziende industriali *asset intensive* (ovvero con molti macchinari), manifatturiere e petrolifere, come ad esempio Stevanato Group.

1.1.3 Processi interni

L'azienda si serve di servizi noti e standardizzati per organizzare il lavoro e la comunicazione interna, ovvero Slack per la messaggistica, Meet per la comunicazione vocale a distanza, Jira per gestire il tracciamento delle *issue* ("cose da fare" in ambito *software*) e facilitare meccanismi agili di integrazione continua, e infine GitHub per quanto riguarda il controllo di versione.
Durante il progetto di stage che, per sua natura, è altamente sperimentale e ha quindi richiesto un approccio *"trial and error"* ci si è limitati a usare una piccola Kanban Board di Jira per pianificare, per quanto possibile, il lavoro da svolgere.
L'azienda adotta un sistema di lavoro agile (come accennato sopra sfruttando Jira per applicare *continuous delivery*) e fornisce anche ad associazioni esterne *tutoring* riguardo a queste tematiche.

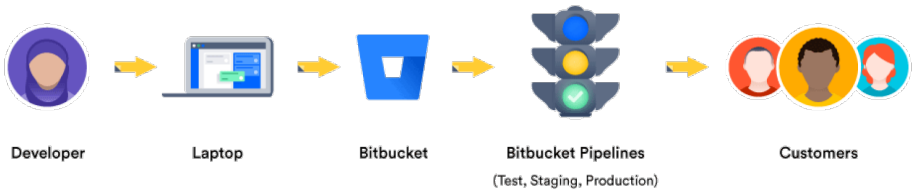


Figura 1.4: *Continuous delivery pipeline* tramite Jira.[7]

1.2 Tecnologie e strumenti

Di seguito vengono riportate le tecnologie e gli strumenti utilizzati durante il corso dello stage, divisi in due macrocategorie: di sviluppo (quindi legati strettamente alla produzione di *software*) e organizzativi (più interessati alla gestione del codice prodotto e la comunicazione interna).
Verranno prima presentati singolarmente e in dettaglio, per poi mostrarne una visione d'insieme.

1.2.1 Tecnologie e strumenti di sviluppo

Visual Studio Code

Visual Studio Code è un *editor* per codice sorgente, sviluppato da Microsoft servendosi del *framework* Electron, disponibile per Windows, Linux e macOS. Le sue funzionalità

[7] Fonte: https://www.atlassian.com/continuous-delivery

comprendo supporto per *debugging*, *syntax highlighting*, *intelligent code completion*, *code refactoring*, e integrazione con Git.
Gli utenti possono configurare tema, *macro*, preferenze e installare estensioni che ne aumentano le funzionalità aggiungendo ad esempio il supporto ai maggiori linguaggi di programmazione attualmente presenti.
Nella "Stack Overflow 2021 Developer Survey", Visual Studio Code è risultato essere l'IDE più popolare, adottato dal 70% degli intervistati (fonte: https://en.wikipedia.org/wiki/Visual_Studio_Code).
È inoltre presente e molto comoda l'estensione di Flutter che, tra le altre cose, permette di gestire direttamente nell'interfaccia i vari dispositivi (*smartphone*, *browser* o emulatore) sui quali installare e lanciare l'applicazione che si sta programmando.

Android Studio

Android Studio è l'IDE (*Integrated Development Environment*, ovvero un *software* per facilitare la scrittura di codice) ufficiale del sistema operativo di Gooogle, sostituendo Eclipse dal 2015, costruito sul *software* IntelliJ di JetBrains e progettato specificatamente per lo sviluppo Android. È disponibile per Windows, Linux e macOS.
Il 7 maggio 2019 Kotlin rimpiazzò Java come linguaggio consigliato per Android, favorendo un'integrazione ancora maggiore con JetBrains che, appunto, ha sviluppato e prodotto Kotlin stesso.
Personalmente ho cercato nonostante tutto di programmare il più possibile su Visual Studio Code preferendolo quindi ad Android Studio in virtù della sua maggiore leggerezza e delle sue più ampie possibilità di personalizzazione.

Xcode

Xcode è l'*integrated development environment* per macOS, utilizzato obbligatoriamente per sviluppare *software* per macOS, iOS, iPadOS, watchOS e tvOS.
Supporta codice sorgente i seguenti linguaggi: C, C++, Objective-C, Objective-C++, Java, AppleScript, Python, Ruby, ResEdit (Rez), e Swift (quest'ultimo è per noi di particolare interesse).
In generale è un *integrated development environment* molto impopolare e viene percepita con grande frustrazione la sua obbligatorietà per lo sviluppo all'interno dell'ecosistema Apple. Personalmente mi sento allineato alla negativa opinione generale, complice soprattutto la scarsa responsività del programma.

Dart

Dart è un linguaggio di programmazione sviluppato da Google per il *client development*, ovvero per applicazioni *web* e *mobile*, e può anche essere impiegato per la costruzione di applicazione *desktop* e *server*.
È un linguaggio orientato agli oggetti, basato su classi, *garbage-collected*, fortemente tipizzato e con una sintassi nello stile di C. Può compilare sia codice macchina che JavaScript e supporta interefacce, *mixins*, classi astratte, *refined generics* e inferenza di tipo.
Personalmente l'ho trovato elegante e conciso nella sua struttura, e presenta dei vantaggi consistenti come le espressioni ternarie e la *null safety* (ovvero i valori di tipo *null* devono essere esplicitati con sintassi specifiche che riducono enormemente la possibilità di incontrare errori a tempo di esecuzione).

Flutter

Flutter è un *framework open-source* creato da Google per lo sviluppo dell'interfaccia utente di applicazioni multipiattaforma per Android, iOS, Linux, macOS, Windows, Google Fuchsia e il *web* partendo da un singolo *codebase* (collezione di *file* sorgente necessari a ottenere un *software* completo e funzionante).
Rilasciato a maggio 2017, le applicazioni in Flutter sono scritte in Dart e sfruttano molte delle funzionalità avanzate del linguaggio. Le componenti principali del *framework* sono:

- **Dart platform:** Flutter gira all'interno della Dart Virtual Machine, che si serve di un *execution engine just-in-time*;
- **Flutter engine:** scritto primariamente in C++, fornisce supporto a basso livello per il *rendering* e implementa accessibilità, *file* e *network input/output*, supporto nativo per i *plugin* e molto altro;
- **Foundation library:** scritta in Dart, fornisce classi e funzioni di base che vengono usate per costruire gli applcativi in Flutter, come ad esempio le *Application Programming Interfaces* per la comunicazione con l'*engine*;
- **Design-specific widgets:** Flutter contiene due famiglie di *widget* che si conformano a delle specifiche scuole di *design*: "Material" per Google e "Cupertino" per Apple;
- **Flutter DevTools:** famiglia di strumenti generici che vengono sfruttati per lo sviluppo di *software* in Flutter.

Inoltre, una libreria fondamentale per semplificare l'utilizzo di Flutter è "Flutter Hooks": lo scopo è implementare delle strutture simili agli Hooks di React, che consentono di sostituire gli "Stateful Widget" riducendo la quantità di codice ripetuto e semplificando la gestione del ciclo di vita dei *widget* grazie alla funzione "useEffect".

Java

Java è un linguaggio di programmazione ad alto livello orientato agli oggetti progettato per avere il minor numero possibile di dipendenze implementative (favorendo quindi l'uso di interfacce e classi astratte) e, soprattutto, per aderire al motto *"write once, run anywhere"*: il *bytecode* di Java può infatti girare senza necessità di ricompilazione su qualsiasi piattaforma che supporti la Java Virtual Machine (comunemente abbreviata in JVM).
La sintassi è simile a quelle di C o C++, ma fornisce funzionalità dinamiche generalmente inedite per i linguaggi compilati, come *reflection* (un programma in esecuzione può modificare se stesso) e modifica del codice a *runtime*.
Originalmente rilasciato nel 1995 come componente centrale della piattaforma Java per Sun Microsystems e sotto licenza proprietaria, dal 2007 la maggior parte dei suoi componenti sono diventati *open-source* sotto l'ombrello della "GPL-2.0-only" (licenza specifica per *software* libero pubblicata dalla Free Software Foundation).
Sebbene Oracle (attuale proprietario di Sun Microsystems) offra un'implementazione proprietaria chiamata "HotSpot JVM", la specifica ufficiale è la "OpenJDK JVM" che è gratuita, *open-source* e predefinita nella maggior parte delle distribuzioni Linux.

Kotlin

Kotlin è un linguaggio di programmazione multipiattaforma staticamente tipizzato. Progettato per avere completa interoperabilità con Java, gode di una sintassi più concisa di quest'ultimo grazie all'inferenza di tipo di cui è dotato. Dipende dalla Java Class Library per la versione della Java Virtual Machine da usare.
È inoltre in grado di compilare codice in JavaScript (sfruttato poi da applicazioni *frontend*) o codice nativo tramite "LLVM" (per app iOS che condividono le logiche operative con applicazioni Android).
Incluso in Android Studio come alternativa al compilatore standard Java dal 2017, produce *bytecode* Java 8 di default ma permette al programmatore di scegliere delle versioni successive (fino a Java 18).

Swift

Swift è un linguaggio di programmazione ad alto livello, compilato, sviluppato da Apple e dalla comunità *open-source*.
Inizialmente rilasciato nel 2014, il suo scopo è rimpiazzare il precedente linguaggio di Apple Objective-C, ormai obsoleto visto che è rimasto largamente invariato dagli anni '80 ed è quindi manchevole di funzionalità moderne.
Swift lavora con i *framework* Cocoa e Cocoa Touch, e un aspetto chiave del suo *design* è l'interoperabilità con una grossa parte del codice esistente in Objective-C (che, naturalmente, permea l'ecosistema Apple) sfruttando la *runtime library* di Objective-C (che permette anche di far girare C e C++).
È stato costruito con il *framework* di compilazione LLVM ed è stato incluso in Xcode dalla sesta versione, rilasciata nel 2014.

Azure Spatial Anchors

Azure Spatial Anchors è un servizio di Microsoft mirato a fornire strumenti per sviluppare applicazioni in realtà aumentata su dispositivi iOS o Android predisposti per, rispettivamente, ARKit e ARCore, oppure su Microsoft HoloLens.
Permette di riconoscere spazi tridimensionali all'interno dei quali posizionare punti di interesse, chiamati Spatial Anchors, che possono essere salvati in *cloud* e poi recuperati, e possono essere messi in relazione a oggetti reali (come macchinari in un contesto industriale) o virtuali.

ARCore

ARCore, conosciuto anche come Google Play Services per AR, è un *software development kit* (generalmente abbreviato in SDK) prodotto da Google per permettere lo sviluppo di applicazioni in realtà aumentata.
Si serve di tre componenti principali:

- **6DOF:** il dispositivo sfrutta una piattaforma inerziale a sei assi per comprendere e tracciare la propria posizione relativa allo spazio;
- **Environmental understanding:** permette riconoscere dimensioni e locazione delle superfici lisce;
- **Light estimation:** permette al telefono di stimare le condizioni di luce dell'ambiente.

Azure Spatial Anchors si appoggia proprio ad ARCore per implementare le proprie funzionalità di realtà aumentata su Android.

ARKit

ARKit è la controparte Apple di ARCore, ovvero è un *software development kit* che permette lo sviluppo di applicazioni in realtà aumentata su dispositivi Apple.
Combina i dati del dispositivo di tracciamento spaziale (come la piattaforma inerziale a sei assi) con un *processing* avanzato della scena ripresa per fornire un'esperienza in realtà aumentata convincente.
Indubbiamente migliore di ARCore, produce risultati più fluidi (inteso come effettiva fluidità percepita guardando la scena attraverso il dispositivo), meglio implementati con le tecnologie esistenti e risparmia al programmatore molti oneri (come la gestione delle sessioni) occupandosene direttamente in modo autonomo.

Visione d'insieme

Visual Studio Code è un *editor* per codice sorgente che permette grande estensibilità tramite *plugin* appositi che, per il progetto, ho configurato in modo tale da aggiungerci la compatibilità con il *framework* utilizzato per lo sviluppo *mobile*, ovvero Flutter, che si appoggia sul linguaggio Dart. Quest'ultimo è un linguaggio orientato agli oggetti e fortemente tipizzato che nasce per la creazione di interfacce grafiche *frontend mobile*.
Integrando Flutter in Visual Studio Code si ottengono vari vantaggi, come ad esempio la possibilità di gestire eventuali emulatori o dispositivi Android con facilità.
Essendo lo scopo del progetto l'integrazione di Azure Spatial Anchors (tecnologia per gestire ancoraggi in realtà aumentata) in un'applicazione Flutter, è stato necessario studiare anche gli *software development kit* sui quali si appoggia il servizio di Microsoft, ovvero ARCore (lato Android) e ARKit (lato iOS). Azure Spatial Anchors implementa tali funzionalità in linguaggi nativi, ovvero Java per Android e Swift per iOS, il che ha portato la necessità a utilizzare due *integrated development environment*(abbreviato in IDE) ulteriori a Visual Studio Code: Xcode e Android Studio.
Xcode è obbligatorio per programmare in Swift, mentre Android Studio da un grosso aiuto al programmatore quando lo usa per sviluppare applicazioni Android in Java o Kotlin (linguaggio che si è dovuto utilizzare per ar_flutter_plugin, ovvero il *framework* ultimamente selezionato che tratteremo nel terzo capitolo di questo documento).

Strumenti di sviluppo e rispettivi linguaggi:

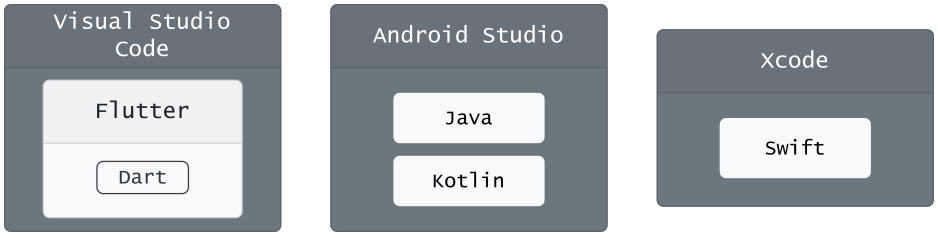


Figura 1.5: *Integrated development environments* e linguaggi per i quali sono stati usati.

. **Tecnologie realtà aumentata, linguaggi e sistemi operativi:**

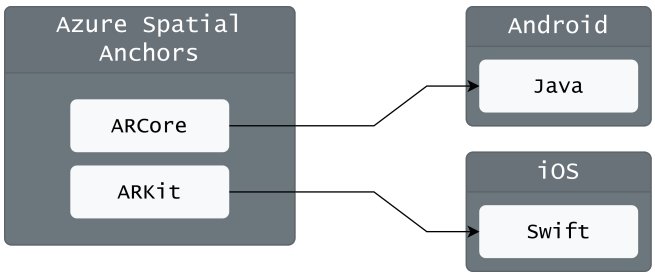


Figura 1.6: Azure Spatial Anchors e linguaggi nativi utilizzati per implementare le sue componenti in Android e iOS.

***Framework* scelto e sua relazione con tecnologie e linguaggi:**
In questo caso il *framework* ha le componenti native scritte in Kotlin che è compatibile nativamente con Java e funge quindi da contenitore (*"wrapper"*) per quest'ultimo.

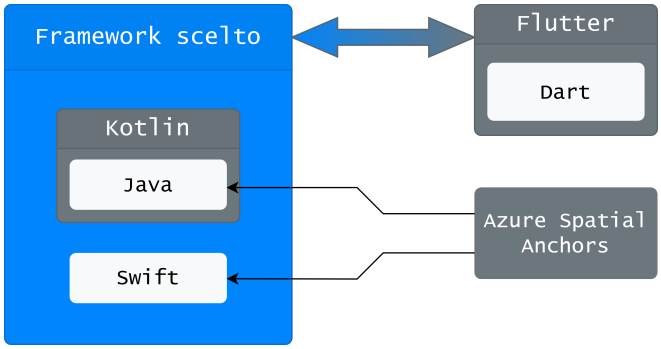


Figura 1.7: Il *framework* selezionato comunica con Flutter e implementa delle componenti native alle quali si interfacciano le Azure Spatial Anchors.

1.2.2 Strumenti organizzativi

Slack

Slack è una piattaforma di comunicazione istantanea di proprietà di Salesforce e sviluppata da Slack Technologies per Windows, Linux, macOS, Android e iOS.
Permette di comunicare tramite messaggi, chiamate vocali e video, e di inviare *media* e *files* nelle *chat* private o nei canali. Quest'ultimi fungono da aggregatori e permettono di suddividere tematicamente la comunicazione.
È inoltre possibile suddividere a un livello ancora più alto tramite i *workspaces*, che aggregano al proprio interno utenti, canali e applicazioni: sono proprio quest'ultime che mostrano con maggiore chiarezza l'orientamento aziendale di Slack, che infatti fornisce integrazione nativa con i maggiori software gestionali e organizzativi (come ad esempio Jira, Zoom, Drive, Outlook, etc).

Github

GitHub è il più grande servizio di *hosting* di codice sorgente al mondo, abitualmente usato per sviluppare progetti *open-source*, e fornisce vari servizi: controllo di versione tramite Git distribuito e *access control*, *bug tracking*, *issue tracking*, richieste di modifiche *software*, *task management*, integrazione continua e *wiki* per ogni progetto.
Permette inoltre, tramite *forking*, *pull request* e commenti, il miglioramento collaborativo del *software*, ad esempio risolvendo errori, aggiungendo funzionalità o migliorando quelle presenti.

Jira

Jira è un *software* proprietario sviluppato da Atlassian che si occupa di fornire una piattaforma potente e funzionale di *bug* e *issue tracking*, presentando il tutto tramite interfacce grafiche chiare e complete. Favorisce inoltre la pianificazione e la collaborazione all'interno dei *team* utilizzando, oltre alle *issues*, anche *tasks* e *user stories*.
Fornisce inoltre *template* già pronti per implementare i più noti *framework* di lavoro come Scrum e Kanban.
Jira mira inoltre a ottimizzare il flusso di lavoro e favorire un miglioramento continuo fornendo visualizzazioni espressive di metriche per tenere traccia dei tempi di sviluppo e scoprire colli di bottiglia.

Visione d'insieme

La messaggistica interna all'azienda avviene sulla piattaforma Slack, che fornisce un'integrazione con strumenti gestionali come Jira (o Google Calendar, Google Meet, etc), e questo fa si che se in una conversazione emerge la necessità di apportare modifiche a un prodotto si possa direttamente aprire una *issue* in Jira, che successivamente viene assegnata a uno sviluppatore che, dopo aver caricato il codice necessario sul *repository* opportuno di GitHub, notifica le avvenute modifiche e quindi chiude, previa approvazione, l'*issue* associata.

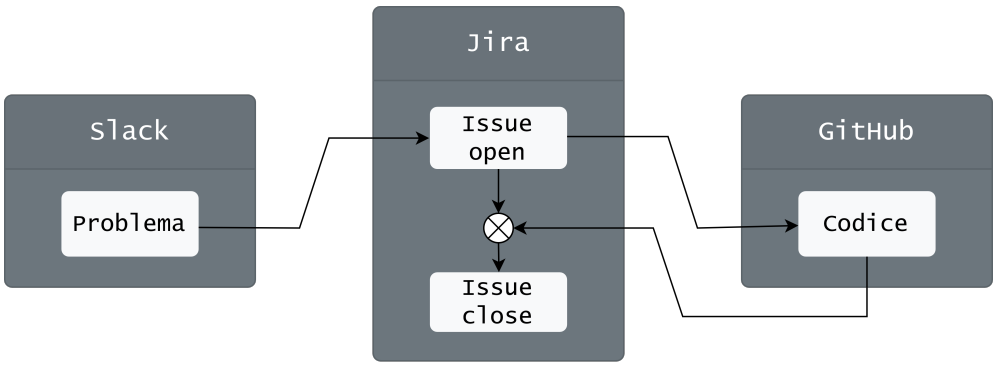


Figura 1.8: Una comunicazione su Slack (ad esempio riguardo un *bug* scoperto) può tradursi nell'apertura di un'*issue* su Jira che viene chiusa quando, tramite codifica, viene risolto il problema che l'ha generata.

[7]Fonte: https://github.com/Azure/azure-spatial-anchors-samples

1.3 Innovazione, stage e prodotti

È immediatamente evidente guardando le tecnologie impiegate che Datasoil pone particolare attenzione sul tema dell'innovazione: lato *backend* troviamo servizi Amazon Web Services impiegati per decentralizzare la gestione del *server* (ad esempio S3), e ovviamente il già citato Azure Spatial Anchors, mentre lato *frontend* vediamo l'impiego di linguaggi di programmazione e *framework*[8] recenti come Dart e Flutter (usati per sviluppare tutt i prodotti in campo *mobile*).
È bene sottolineare, però, che gli strumenti scelti non sono stati selezionati solo per rincorrere un ideale di avanguardia: hanno infatti tutti alle spalle delle grosse corporazioni (Amazon nel caso di Amazon Web Services, Microsoft nel caso di Azure Spatial Anchors e Google nel caso di Dart e Flutter) che garantiscono loro un certo grado di stabilità e sicurezza.
L'attenzione all'innovazione mostrata da Datasoil si riflette sia negli *stage* proposti che nei prodotti e servizi direttamente offerti dall'azienda: ad esempio nell'uso di HoloLens per implementare funzioni di realtà aumentata oppure nelle sessioni di *tutoring* per terzi (riguardo appunto queste tecnologie o pratiche di sviluppo agile) che i *senior* svolgono a cadenza regolare.

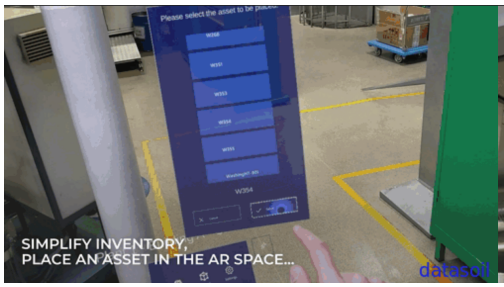


Figura 1.9: Menù HoloLens

Figura 1.10: Posizionamento *asset*

Figura 1.11: HoloLens usato in azienda per gestire *asset* in realtà aumentata.[9]

Gli *stage* riescono a essere innovativi sia direttamente che indirettamente: se si tratta di un progetto sperimentale vero e proprio lo è per sua stessa natura, tuttavia anche nel caso in cui l'obiettivo sia limitato a sviluppare funzioni in maniera più "meccanica", porta comunque lo studente ad avere a che fare con gli strumenti moderni che vengono impiegati giornalmente in azienda.
Vengono quindi integrati nel contesto aziendale in modalità duplice: a volte sono funzionali alla sperimentazione di nuove soluzioni, altre volte sono già inquadrati in una *pipeline* produttiva (ad esempio potrebbero trattare lo sviluppo di alcune pagine di un'applicazione *mobile*). Anche nel primo caso però se ciò che è stato prodotto risulta interessante per dei possibili clienti viene portato avanti e integrato anche quando lo *stage* si è ormai concluso.

[8] Collezione di codice che fornisce funzionalità generiche che possono essere sovrascritte, specializzandole, per adattarle agli scopi voluti. Sono simili alle librerie se visti come astrazioni riutilizzabili di codice incluso in APIs (*Application Programming Interfaces*, permettono a due o più componenti *software* di comunicare) ben definite, tuttavia ciò che le differenzia è che il controllo di esecuzione non è dettato dal chiamante, ma dal *framework* stesso. È proprio questa inversione di controllo la caratteristica principale dei *framework*.

[9] Fonte: https://datasoil.it/industry-40-smartcity-syn-asset-performance/auditing-inventory/

Capitolo 2

Lo stage

2.1 Strategia aziendale

L'azienda si rapporta con gli *stage* in un'ottica produttiva: si tratta spesso di attività che si innestano in lavori già in atto o che sarebbero comunque stati eseguiti, a volte più meccanici, a volte più sperimentali (come nel mio caso che si è legato fortemente al concetto d'innovazione per la natura stessa del progetto trattato) e permettono inoltre di effettuare un primo *screening* per eventuali assunzioni di *junior* o stagisti.
Per avvalorare le tesi appena esposte prendiamo ad esempio gli *stage* proposti durante lo stage-it 2022: il primo è quello che ultimamente è stato assegnato a me, il secondo comporta lo sviluppo di funzionalità di monitoraggio e inserimento di *asset* e spazi aziendali interni, *ticketing* e gestione eventi in tempo reale per i moduli della piattaforma SYN (prodotto interno di Datasoil). Il terzo *stage* prevede, sempre per SYN, lo sviluppo di un servizio per l'amministrazione della piattaforma permettendo la creazione, modifica e supervisione dei diversi *tenant* legati ai diversi clienti, garantendo isolamento e correttezza delle informazioni.

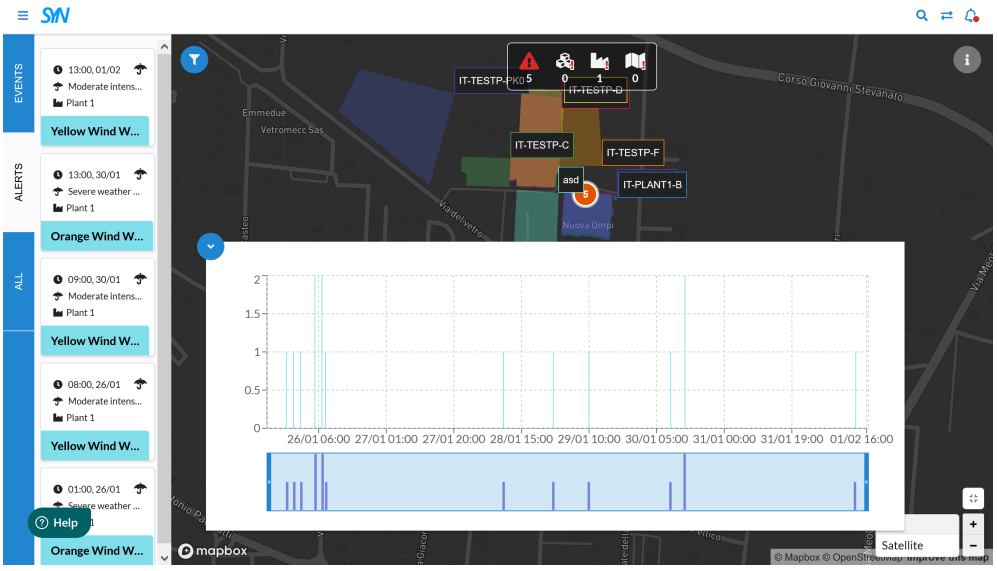


Figura 2.1: *Screenshot* portale SYN di infografica allerte meteo per stabilimento "Plant 1".[1]

[1]Fonte: https://synmgr.com

Lo *stage* a me proposto è stato inquadrato in una strategia che prevede l'espansione di funzionalità per applicativi esistenti nell'ottica di fornire valore aggiunto ai clienti attuali e futuri: precedentemente a questo progetto non era presente alcuna componente in realtà aumentata in MobileSYN, mentre la strategia futura è un continuo miglioramento di ciò che è stato aggiunto. Infatti non sono state sufficienti otto settimane per ottenere un risultato professionalmente soddisfacente, e Datasoil nel futuro si occuperà di limare l'interfaccia grafica (anche osservando sul campo le abitudini d'uso della stessa da parte degli operatori) e migliorare pulizia ed efficienza del codice prodotto.

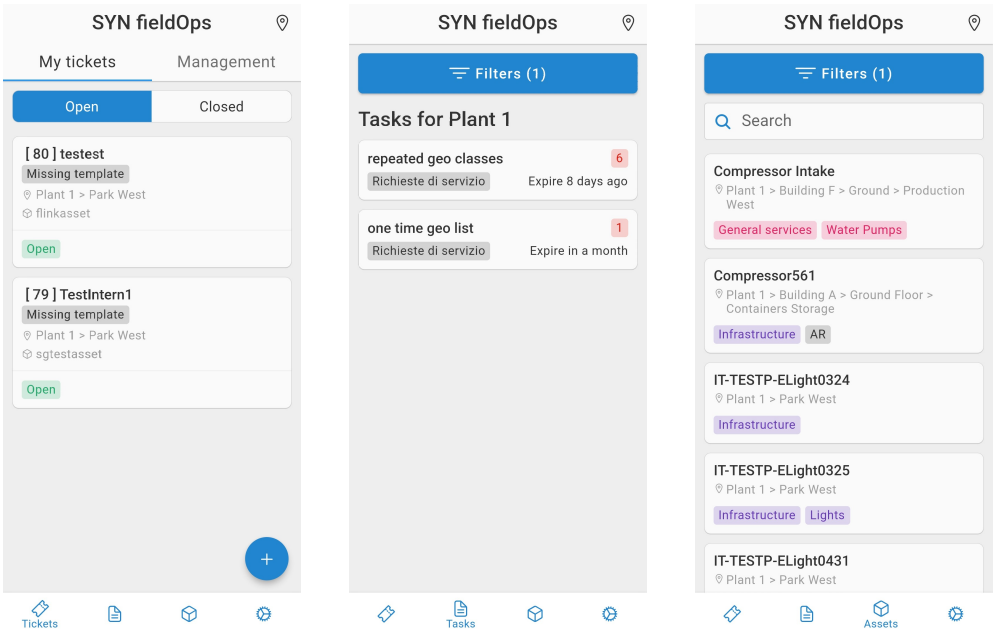


Figura 2.2: *Screenshot* di SYN fieldOps (internamente chiamata MobileSYN) delle schermate "*Tickets*", "*Tasks*" e "*Assets*" per lo stabilimento "Plant 1".

L'azienda si è attivamente impiegata a supportare il mio *stage* con tutta una serie di attività: sono stato direttamente ospitato in ufficio e ho avuto accesso a risorse aziendali (*account* Amazon Web Services per accedere a MobileSYN) e credenziali Azure Spatial Anchors, inoltre sia il *junior* che il *senior developer* di Datasoil che si occupano del lato *frontend* mi hanno attivamente affiancato durante lo sviluppo nella fase di codifica. Ho inoltre ricevuto supporto nella fase di analisi dei requisiti e in altri piccoli problemi relativi alla configurazione dei dispositivi e dei programmi necessari ad affrontare lo *stage*.

2.2 Descrizione e temi

Il progetto proposto prevede l'implementazione di una vista in realtà aumentata[2] per l'applicazione di *ticketing* e monitoraggio aziendale "MobileSYN".
Questo applicativo permette di creare un gemello digitale di un *asset* aziendale e posizionarlo all'interno di uno spazio virtuale tridimensionale, così da rendere visivamente

[2]Schermata che mostra il *feed* della fotocamera e permette di interagirci come fosse uno spazio 3D renderizzato

più chiaro quali macchinari di un dato stabilimento industriale abbiano necessità di manutenzione (mediante apertura di *ticket* appositi).
Un *ticket* è semplicemente una notifica relativa a un *asset* (perdita d'olio per una pompa idraulica, carta esaurita per una stampante, etc) ma può anche essere libero da questo vincolo e trovarsi, in autonomia, posizionato nello spazio (ad esempio per segnalare una crepa su un muro).

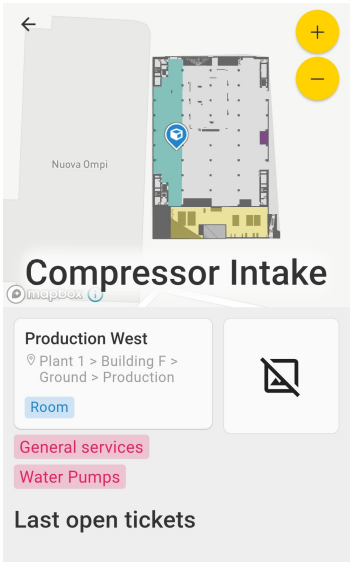


Figura 2.3: *Asset*

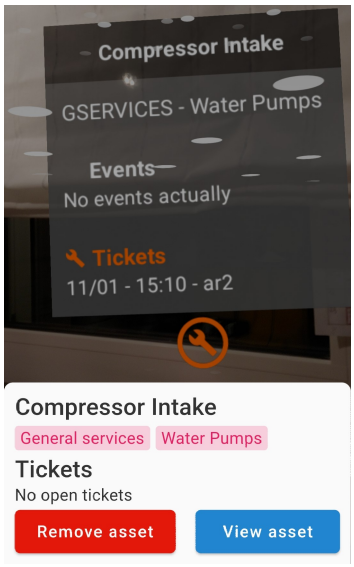


Figura 2.4: *Card* e *anchor*

Figura 2.5: Nella prima figura vediamo l'*asset* "*Compressor Intake*" in stabilimento "Plant 1", mentre nella seconda una *card* (in primo piano) relativa a una *anchor* (in secondo piano) in vista in realtà aumentata.

Tutte le funzionalità in realtà aumentata dovranno necessariamente appoggiarsi al servizio di Azure Spatial Anchors perché i gemelli digitali già presenti nel sistema sfruttano proprio questa tecnologia per la localizzazione spaziale e, vista la natura multipiattaforma dell'applicazione, dovranno essere implementate sia in Android che in iOS.
Dalla vista dovrà essere possibile interagire con gli *asset* e i *ticket*, provocando l'apparizione di una *card* all' *on-tap* dell'*anchor*[3], ovvero mostrando a schermo tramite una piccola finestra in sovrimpressione i vari *ticket* relativi all'*asset* dopo che questo viene toccato, permettendo anche la visione del dettaglio o la rimozione dello stesso.
Per portare a termine gli obiettivi prefissati sarà necessario studiare dei *framework*, dei *plugin* o dei *software development kit* che permettono l'integrazione del codice nativo di Azure Spatial Anchors (che sarà Java per Android e Swift per iOS) con il *framework* Flutter e il linguaggio su cui si poggia, ovvero Dart.

[3]Traducibile in italiano con il termine "ancoraggio" è un punto di interesse in uno spazio tridimensionale, ovvero un punto che possiede delle coordinate *(x,y,z)*, al quale possono essere collegati *asset*, *ticket* o altri ancoraggi.

2.3 Obiettivi

Riporto di seguito la lista degli obiettivi posti dall'azienda per questa esperienza di *stage*:

- Minimi:
  - Studio e comprensione del linguaggio di programmazione Dart e del *framework* Flutter;
  - Analisi strumenti Azure: Azure Console e Azure Spatial Anchors;
  - Ricerca di *framework* e *software development kit* per implementare le Azure Spatial Anchors in Flutter;
  - Eventuale studio di linguaggi necessari alle implementazioni native, come ad esempio Kotlin, Java, Unity o Swift;
  - Completamento del *framework* scelto nell'app Flutter esistente;
  - Completamento dello sviluppo dei componenti per rappresentare le *anchor* nello spazio in realtà aumentata;
  - Completamento dello sviluppo delle *application programming interfaces* per l'interazione utente con gli ancoraggi in realtà aumentata;
- Massimi:
  - Sviluppo di componenti per mappare *asset* aziendali ad *anchor*;
  - Sviluppo di componenti grafici per la visualizzazione di metriche e informazioni di controllo.

2.4 Vincoli

Come precedentemente accennato, i vincoli in questo progetto sono due:

- Utilizzare Flutter come *framework* sul quale implementare le componenti in realtà aumentata;
- La tecnologia scelta per gestire gli ancoraggi in realtà aumentata deve essere Azure Spatial Anchors.

Entrambi i vincoli nascono dal fatto che tutto il lavoro di *stage* va svolto su un'applicazione preesistente, MobileSYN, che è sviluppata in Flutter e già implementa la localizzazione geografica degli *asset* tramite Azure Spatial Anchors. Di conseguenza per visualizzare gli *asset* in realtà aumentata bisogna interfacciarsi con gli ancoraggi di Azure e il tutto va chiaramente implementato nel *framework* sul quale l'applicazione è svilppata.
Eccezion fatta per queste specifiche tecnologiche il resto è stato lasciato a mia discrezione personale, sebbene siano stati consigliati degli strumenti di sviluppo (Visual Studio Code e Android Studio come *integrated development environment*).
Vincoli non tecnologici ma organizzativi sono da identificarsi nell'uso di Slack e GitHub: il primo usato per la comunicazione interna con i membri di dell'azienda, il secondo invece è il *repository*[4] dove si trova il codice sorgente di MobileSYN.

[4]Spesso abbreviato con *"repo"* è un generico archivio di codice sorgente e pacchetti *software*. Spesso contiene anche *metadata* e guide per la configurazione, uso ed estensione del codice che contiene. In genere un *repo* ha una qualche forma di controllo di versione integrato.

2.5 Pianificazione

La natura estremamente sperimentale di questo progetto ha reso vano ogni tentativo di pianificare accuratamente il lavoro (complici anche le problematiche che vedremo nella sezione 4.1).
Per questo motivo, una volta scelto il *framework* per l'implementazione di Azure Spatial Anchors in Flutter l'approccio scelto è stato di *trial-and-error*, e ha impegnato una buona parte del tempo di stage sia mio che dei membri di Datasoil responsabili del *frontend*.
È stata comunque fatta, di settimana in settimana, una scaletta degli obiettivi da raggiungere per il lunedì successivo, tuttavia sono riuscito a seguirla solo fintanto che comprendeva lo studio delle tecnologie e la creazione di brevi *proof of concept* (ad esempio costruire un'applicazione in Dart e poi tradurre i suoi *stateful widget* in *hook widget*).
Riporto quindi la programmazione settimanale che, siccome questo documento è scritto al termine dei lavori, è in parte anche un rendicontazione:

- **Settimana 1:**
  - Installazione e configurazione del *framework* Flutter e del *software development kit* per Android
  - Configurazione emulatore Android (scelto Pixel 6 API 33) e *developers options* sul mio telefono personale;
  - Configurazione Visual Studio Code e Android Studio;
  - Sviluppo di applicazione d'esempio con Flutter che sfrutta gli *stateful widgets*;
- **Settimana 2:**
  - Studio degli *hook widgets*;
  - Traduzione degli *stateful widgets* dell'applicazione d'esempio in *hook widgets*;
  - Sviluppo applicazione d'esempio con gli *hook widgets*;
- **Settimana 3:**
  - Studio delle Azure Spatial Anchors;
  - Creazione risorse necessarie nel portale di Azure;
  - Sviluppo applicazione di esempio per Azure Spatial Anchors;
- **Settimana 4:** Studio dei metodi per implementare le Azure Spatial Anchors in Flutter:
  - Studio dei *method channels*;
  - Studio di ARwayKit e valutazione del suo approccio tramite Unity;
  - Studio di ar_flutter_plugin;
- **Settimana 5:**
  - Adattamento dell'esempio ar_flutter_plugin per integrarci le Azure Spatial Anchors;

  - Lettura dei *logs* per filtrarne gli *output*;

- **Settimana 6:**
  - Utilizzo diretto di ARCore in Flutter per capirne il flusso;
  - Integrazione Azure Spatial Anchors in Flutter;
- **Settimana 7:** Integrazione componenti realtà aumentata in MobileSYN lato Android;
- **Settimana 8:** Integrazione componenti realtà aumentata in MobileSYN lato iOS;

I processi di verifica e validazione non hanno potuto fare affidamento su sistemi consolidati, in quanto tutto il lavoro è stato genuinamente sperimentale, e quindi ci siamo affidati al *testing* "sul campo", ovvero provando direttamente l'applicazione (a un certo punto usando la versione in produzione) per valutare il progresso dei lavori e scoprire la presenza o meno di errori.
Di grande aiuto sono stati i *logs* delle varie applicazioni provate (da MobileSYN, all'applicazione di esempio di ar_flutter_plugin fino a quella di Azure Spatial Anchors) che hanno, in parte, sopperito alle gravi mancanze documentali delle tecnologie trattate.

2.6 Motivazione scelta

I motivi che mi hanno spinto ad accettare questo stage rispetto ad altri sono molteplici, ma i principali sono quattro: toccare con mano attività legate al *frontend*, vivere in prima persona l'ambiente di lavoro del mio ambito di studi, avere la possibilità di sviluppare in ambito *mobile* e una personale curiosità nei confronti delle tecnologie trattate, ovvero la realtà aumentata.

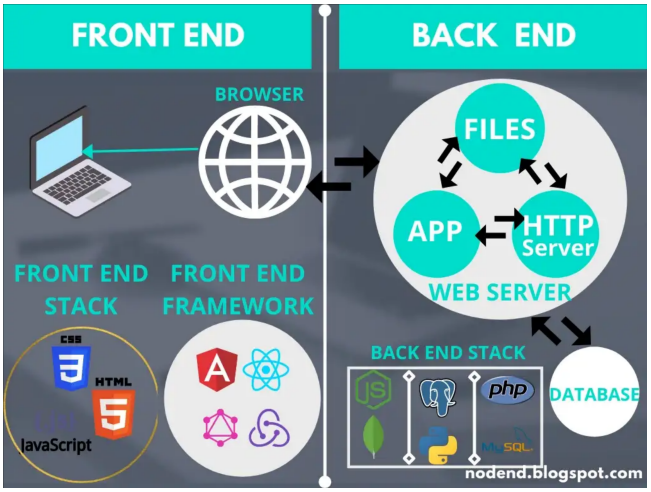


Figura 2.6: Confronto *backend* e *frontend*.[5]

Elaboro ora con più completezza i quattro punti di cui sopra: per quanto riguarda il *frontend*, è un ambito o comunque sia una branca del mondo dello sviluppo *software* che guardavo con interesse in quanto raramente ho avuto occasione di occuparmene durante il corso di studi. Infatti nella maggioranza dei progetti e degli *assignment* trattati mi sono sempre occupato del lato *backend*, e il cambio di approccio si è dimostrato difficoltoso, costringendomi a ragionare e a vedere il codice in maniera diversa. Flutter

[5]Fonte: https://dharmanikheem.medium.com/front-end-vs-back-end-what-is-the-difference-9a83a65ee74f

inoltre ha reso il tutto ancora più arduo siccome mischia aspetti di interfaccia grafica con logiche di controllo.
Vivere uno *stage* a contatto con degli sviluppatori di grado *senior* e lavorare in un ambiente con un certo grado di controllo era per me molto importante: da una parte per garantirmi di assorbire conoscenze da personale esperto, dall'altra per assicurarmi di mantenere un *focus* quanto più alto possibile durante il corso del progetto. Era inoltre fondamentale avere un'idea più chiara di come sia il mondo del lavoro in ambito *software*, per capire se il mio desiderio di, in futuro, mettermi in proprio valga la pena essere perseguito o meno.

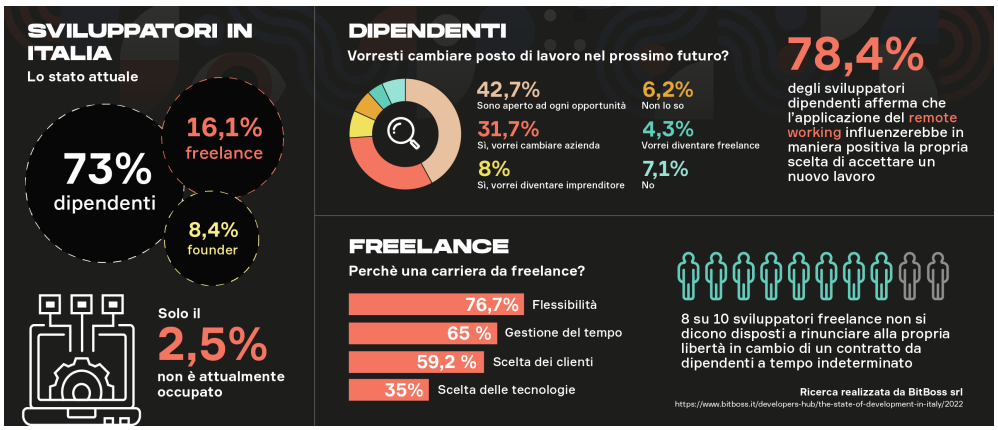


Figura 2.7: Infografica che evidenzia quanto sia fondamentale l'equilibrio vita-lavoro.[6]

Il terzo punto nasce da una considerazione: il mondo dello sviluppo *mobile* è quello più ampio e in continua crescita. Ormai i *computer* sono sempre più rari e vengono generalmente acquistati da persone ad alto livello di specializzazione, mentre il pubblico generalista preferisce usare gli *smartphone*. È quindi saggio a mio avviso sviluppare delle competenze in questo ambito perché saranno sicuramente molto spendibili.

[6]Fonte: https://www.bitboss.it/developers-hub/the-state-of-development-in-italy/2022

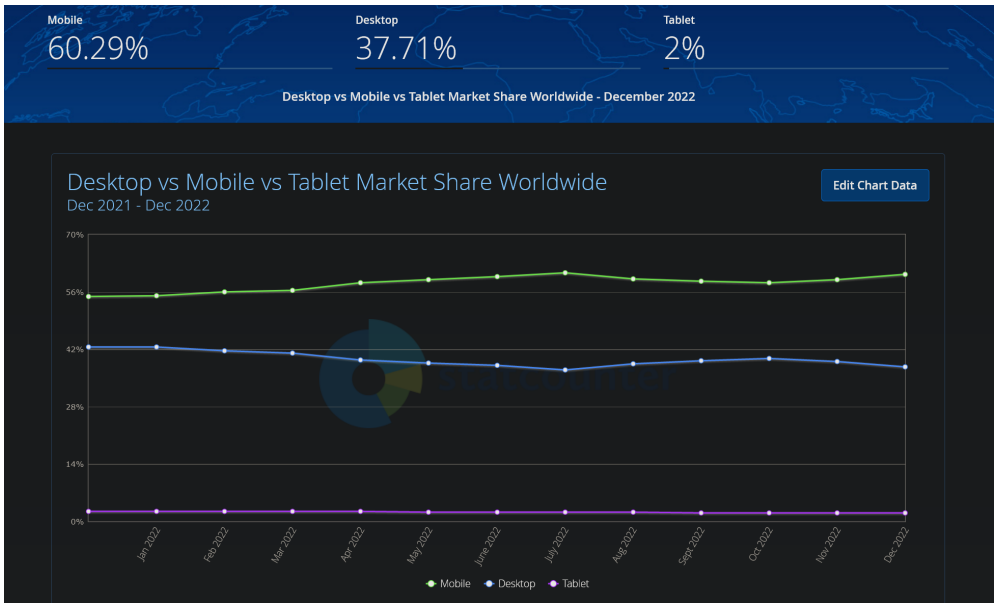


Figura 2.8: Infografica utilizzo applicazioni *mobile* rispetto a *desktop* e *tablet*.[7]

Infine, il quarto e ultimo motivo che mi ha spinto a scegliere questo *stage* rispetto ad altri riguarda la tecnologia affrontata: trovo affascinante l'impiego della realtà aumentata perché è molto poco comune e, probabilmente, diventerà man mano più diffusa con il passare degli anni. Inoltre tra quelle disponibili ho trovato intelligente da parte di Datasoil scegliere proprio quella di Microsoft visto che, forte di disponibilità economiche fuori misura, difficilmente abbandonerà un progetto sul quale ha già investito così largamente.

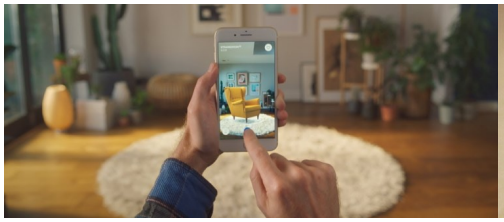


Figura 2.9: Ikea realtà aumentata

Figura 2.10: Nintendo realtà aumentata

Figura 2.11: Ikea sperimenta con la realtà aumentata per vendere mobilia, Nintendo per scopi ludici.[8]

Naturalmente oltre a tutto ciò sono ho avuto un'impressione umana molto positiva durante la riunione di primo contatto avuta con Andrea Ongaro (che poi è diventato il mio *tutor* aziendale) e Pietro Decaro (amministratore delegato di Datasoil), il che mi ha incentivato ad accettare questo specifico progetto.
Schematizzo quindi gli obiettivi che voglio perseguire tramite l'esperienza di *stage*:

[7] Fonte: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet

[8] Fonte Ikea: https://www.ikea.com/au/en/customer-service/mobile-apps/say-hej-to-ikea-place-pub1f8af050
Fonte Nintendo: https://vrscout.com/news/nintendo-president-says-company-exploring-ar/

- Acquisire competenze di programmazione *frontend*:
  - Utilizzare linguaggi specifici (ad esempio Dart);
  - Sviluppare componenti grafiche per applicazioni *mobile*;
  - Sviluppare familiarità con strumenti comunemente usati (ad esempio Visual Studio Code);
- Osservare direttamente il lavoro in azienda *software*:
  - Vedere organizzazione giornaliera lavori;
  - Valutare tempistiche e ritmi lavorativi;
  - Valutare equilibrio vita-lavoro;
- Acquisire competenze di sviluppo *mobile*:
  - Sviluppare familiarità con strumenti comunemente usati (ad esempio emulatori);
  - Sviluppare componenti applicazione *mobile*;
  - Utilizzare *framework* specifici (ad esempio Flutter);
- Acquisire competenze di realtà aumentata:
  - Comprendere teoria dietro concetti come ancoraggi e tridimensionalità;
  - Conoscere tecnologie principali usate nel campo;
  - Valutare se sia di mio interesse perseguire studi futuri sull'argomento.

Capitolo 3

Realizzazione

3.1 Studio delle tecnologie

In questa sezione presento le tecnologie principali utilizzate durante il progetto, cercando di essere quanto più chiaro e sintetico possibile. Questo allo scopo di agevolare la comprensione delle parti finali del capitolo.

3.1.1 Realtà aumentata

La realtà aumentata consiste nel prendere lo spazio tridimensionale reale e trasporlo, tramite tecniche di mappatura, in formato digitale per poi poterci effettuare delle manipolazioni tramite *software*. Ad esempio è possibile, nell'applicazione di Ikea, inquadrare il proprio salotto e posizionarci dei modelli poligonali[1] che rappresentano la mobilia venduta dal colosso svedese.

[1]Detti anche *asset* poligonali, prendono questo nome perché nel mondo della modellazione digitale tridimensionale le entità sono composte da triangoli adiacenti tra loro (poligoni appunto) che costruiscono una figura volumetrica. Questo perché il triangolo è la più semplice forma bidimensionale esistente, e quindi la meno esosa da calcolare.

Figura 3.1: Tramite l'applicazione di Ikea è possibile posizionare in realtà aumenta sedie o altri mobili.

Sebbene sia possibile immaginare lo spazio virtuale come un cubo vuoto all'interno del quale posizionare *asset*, vengono applicate delle semplificazioni logiche per una questione di complessità ed efficienza di calcolo: lo spazio viene processato come una serie di piani bidimensionali sovrapposti a diverse altezze (una sorta di *stack* di battaglie navali) e gli ancoraggi si legano quindi a una coppia di coordinate *(x,y)* relativa al piano più la coordinata *(z)* del piano stesso. I piani vengono divisi in verticali e orizzontali.

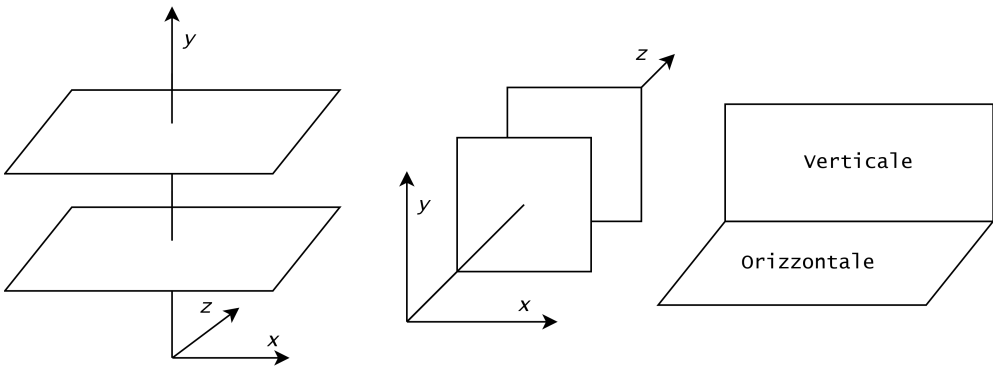


Figura 3.2: In realtà aumentata vengono identificati due tipologie di piano: orizzontale e verticale. Possono essere usati sia singolarmente (prima e seconda figura) che assieme (ultima figura).

Nella maggior parte dei casi è bene che oggetti multipli siano legati a un ancoraggio singolo, questo perché la *pose* degli oggetti (ovvero rotazione verticale e orizzontale) è legata a quella della *anchor* associata. Di conseguenza se vogliamo rappresentare

in una stanza un insieme realistico di mobili (o meglio, di modelli poligonali e quindi gemelli digitali di mobili), e li leghiamo a un ancoraggio centrale rispetto alla stanza tutti gli elementi avranno sempre una posizione relativa corretta tra loro. Se invece ogni elemento ha la sua *anchor* potremmo vedere rotazioni errate oppure potrebbero spostarsi anche di qualche metro rispetto alla loro posizione originale (entrambi questi scenari vanno intesi in relazione con gli altri ancoraggi e quindi con gli altri oggetti).

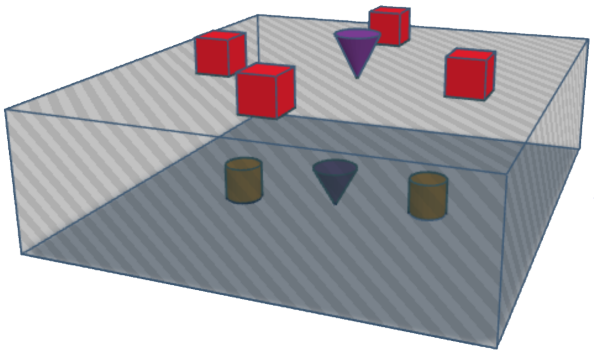


Figura 3.3: A un ancoraggio (in viola) possono essere collegati molteplici elementi (rispettivamente i cubi rossi per l'ancoraggio in alto e i cilindri gialli per quello in basso), e gli ancoraggi possono essere nello stesso spazio ma su due piani diversi.

3.1.2 Microsoft Azure

Azure è l'ombrello sotto al quale vivono tutti i servizi *cloud* di Microsoft esterni al pacchetto Office 365, e comprendono *database* relazionali con scalabilità automatica (Azure SQL o Azure Cosmos DB), macchine virtuali remote (Virtual Machines e Azure Virtual Desktop), ma anche funzionalità estremamente avanzate come Azure Kubernetes Service che fornisce un sistema automatizzato per rilasciare, scalare e gestire grandi applicativi *sofware* (ad esempio per i *data center*), oppure Azure Quantum che mette a disposizioni soluzioni per scrivere e provare *software* su *hardware* quantistico. La *suite* Azure, quindi, è indirizzata a utenti altamente specializzati e ancora più spesso a organizzazioni vere e proprie.

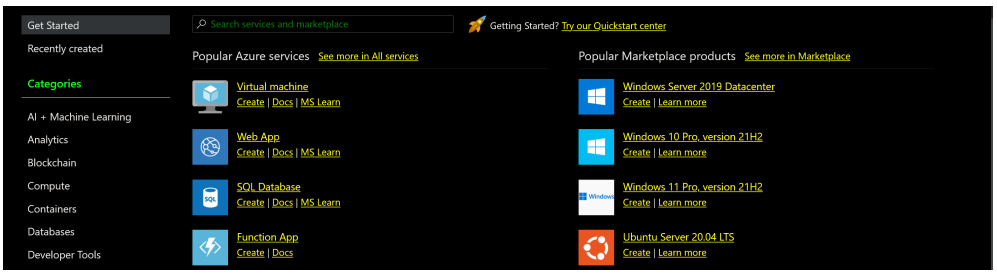


Figura 3.4: Tramite l'Azure Portal è possibile creare tutte le risorse di Azure, come *server* Ubuntu o macchine virtuali.

Per gli scopi di questo progetto uso due degli strumenti forniti: l'Azure Portal e le Azure Spatial Anchors.
Il primo è un portale accessibile tramite *browser* che serve a gestire, creare e monitorare tutte le risorse Azure legate a un *account*, mentre le seconde sono il servizio di Microsoft per gestire ancoraggi geospaziali persistenti che possono anche essere utilizzati in un contesto di realtà aumentata.

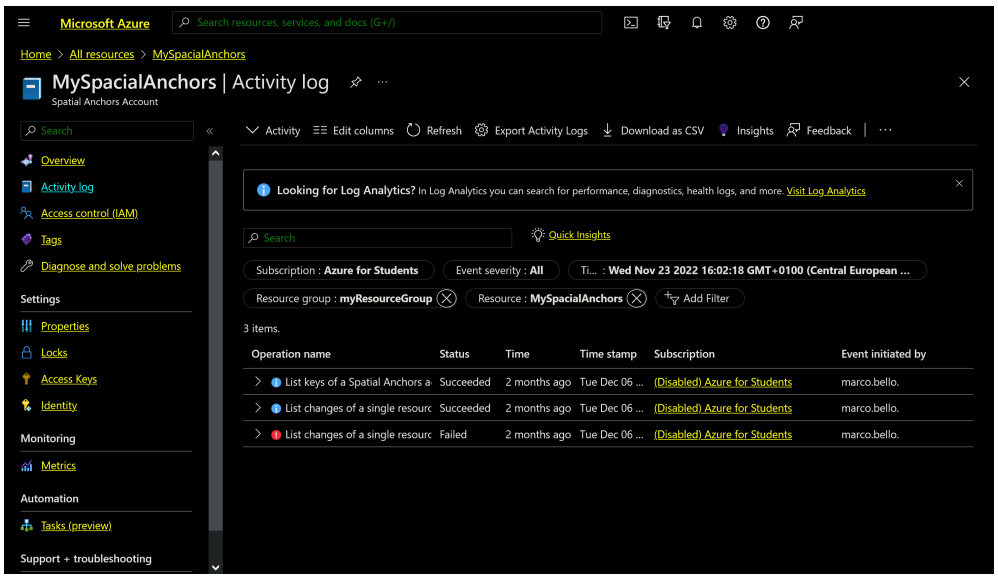


Figura 3.5: Pagina dell'Azure Portal che mostra un dettaglio delle Azure Spatial Anchors, nello specifico il *log* delle attività.

Le Azure Spatial Anchors forniscono tre *software development kit*s, uno per Microsoft HoloLens, uno per iOS appoggiandosi su ARKit e uno per Android basato su ARCore, possono collegarsi tra loro creando relazioni che formano dei veri e propri percorsi (funzionalità usata spesso nel contesto degli *asset* aziendali, ad esempio in una stessa linea produttiva) e possono essere contenuti dentro un *resource group* di Azure che funge da contenitore di risorse dentro al quale esse vengono create e gestite.

3.1.3 Framework realtà aumentata per Flutter

Essendo lo scopo di questo stage, e quindi il caso di studio di questa tesi, l'implementazione di una vista in realtà aumentata (nello specifico sfruttando Azure Spatial Anchors) in un'applicazione preesistente sviluppata in Flutter, una grossa parte del lavoro svolto si è incentrato sullo studio dei *framework* e/o *plugin* disponibili.
Sfortunatamente, complici la giovinezza di Flutter e la scarsa diffusione di tecnologie per la realtà aumentata, le opzioni sono alquanto ridotte:

- ar_flutter_plugin: *plugin*[2] che punta a implementare le componenti in realtà aumentata in modalità *cross-platform*, quindi adattandosi autonomamente ad Android e iOS (sfruttando tuttavia le Cloud Anchor di Google). Nasce partendo da due *plugin* più specializzati:

[2]Fonte: https://pub.dev/packages/ar_flutter_plugin

  - arcore_flutter_plugin per Android[3];
  - arkit_flutter_plugin per iOS[4];

- ARwayKit: *framework* che mira a fornire un'integrazione semplificata, e in parte già completata, di una componente in realtà aumentata (ottenuta tramite Azure Spatial Anchors) in Flutter tramite vista in Unity.

È necessario notificare che al momento della stesura di questo testo non è più possibile accedere liberamente alla documentazione relativa ad ARwayKit[5], tuttavia è ancora reperibile un articolo sul sito Medium che ricalca la guida introduttiva precedentemente fornita dal sito ufficiale dell'azienda[6]. È anche possibile trovarne una versione nella Wayback Machine[7].

ARWayKit

ARwayKit è un *framework* formato da un insieme di componenti che si pone l'obiettivo di fornire un'esperienza in realtà aumentata persistente, e lo persegue fornendo un *software development kit* Unity, un'applicazione di mappatura e un insieme di *application programming interfaces* REST[8], che implementano le componenti in realtà aumentata in Flutter sfruttando il *plugin* flutter_unity_widget[9].
È multipiattaforma, implementa gli ancoraggi tramite Azure Spatial Anchors e funziona in ambienti *"GPS-denied"*, ovvero dove i dati di posizione del *global positioning system*[10] non vengono utilizzati per muoversi all'interno dell'ambiente virtuale (ad esempio in ambienti con protocolli di *privacy* elevati). Le caratteristiche appena descritte rendono ARwayKit apparentemente perfetto per il caso d'uso specifico di questa tesi.

[3]Fonte: https://github.com/giandifra/arcore_flutter_plugin
[4]Fonte: https://github.com/olexale/arkit_flutter_plugin
[5]Fonte: https://app.gitbook.com/s/-MCtct_TY9f3e8PrcV9T/arwaykit-with-flutter/quickstart-in-flutter
[6]Fonte: https://medium.com/arway/building-ar-navigation-apps-with-flutter-and-arwaykit-280b69401cd9
[7]Fonte: https://web.archive.org/web/20220525060655/https://docs.arway.app/arwaykit-with-flutter/quickstart-in-flutter
[8]Conosciute anche come RESTful API, sono delle *application programming interfaces* che si conformano allo stile architetturale e alle norme per sistemi distribuiti chiamato *representational state transfer* largamente utilizzato nella comunicazione *web*.
[9]Fonte: https://pub.dev/packages/flutter_unity_widget
[10]Sistema di satelliti in grado di fornire a un ricevitore le sue coordinate geografiche.

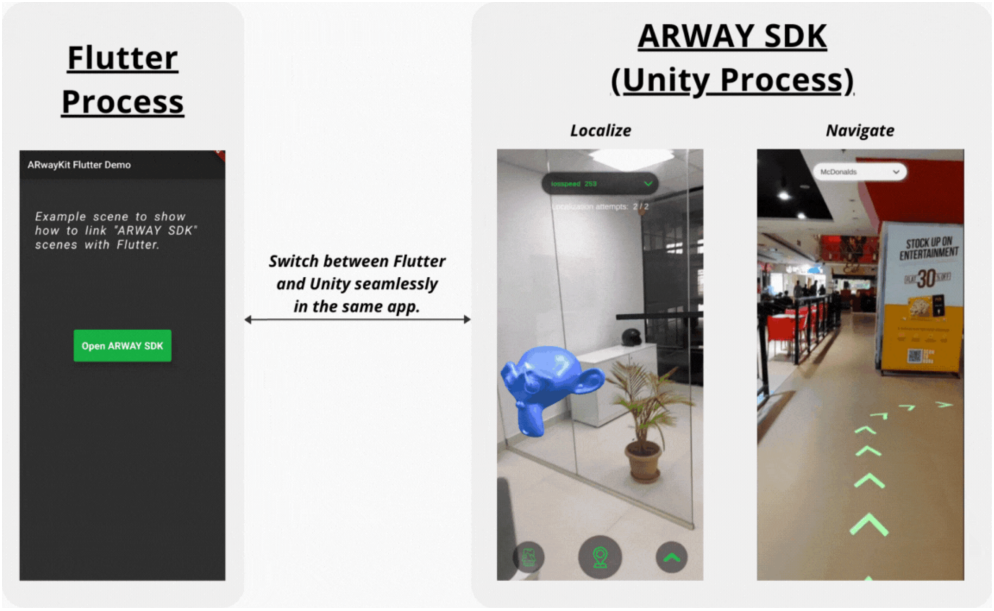


Figura 3.6: Immagine esempio di utilizzo realtà aumentata con ARwayKit[11]

Tuttavia, non è esente da criticità: in primo luogo obbliga l'uso di Unity (sfruttando la sua caratteristica peculiare di poter essere importato come fosse una libreria) per sviluppare la vista in realtà aumentata, il che comporta non solo dover imparare il linguaggio ma dover anche configurare ulteriori componenti (come ad esempio Unity Hub) che poi devono essere adottati assieme a quelli già in uso, appesantendo quindi il processo di codifica.
In secondo luogo si individua il problema maggiore: la vista verrebbe realizzata nativamente in Unity, ovvero si tratta di una componente Unity separata rispetto all'applicazione che la lancia. Questo renderebbe complesso se non impossibile usare dentro essa delle funzionalità Flutter: ad esempio, un bottone a schermo sarebbe un pulsante Unity, non Flutter, obbligando quindi poi a costruire una mappatura tra i due linguaggi per ogni funzionalità visualizzata.
Una vista così costruita non solo è scomoda da programmare, ma anche difficile da manutenere.

ar_flutter_plugin

ar_flutter_plugin è un *plugin open-source* collaborativo estremamente giovane (il primo *commit* sulla *repository* pubblica risale al 6 febbraio 2021[12]) che si pone l'obiettivo di implementare componenti in realtà aumentata in Flutter.
Per raggiungere lo scopo si serve della libreria archiviata Sceneform[13], che si occupa di "renderizzare scene tridimensionali realistiche in applicazioni in realtà aumentata o meno, senza dover imparare OpenGL".

[11] Fonte: https://medium.com/arway/building-ar-navigation-apps-with-flutter-and-arwaykit-280b69401cd9
[12] Fonte: https://github.com/CariusLars/ar_flutter_plugin/commit/da9ab148219d833755ef7a9b1e3536a3cae865d1
[13] Fonte: https://developers.google.com/sceneform/develop

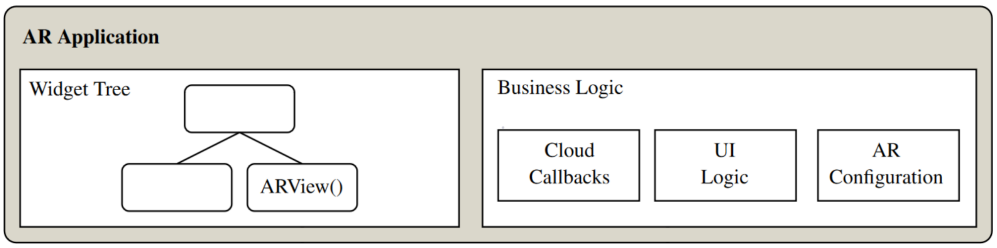


Figura 3.7: Schematizzazione applicazione cliente per ar_flutter_plugin.[14]

L'architettura di ar_flutter_plugin è composta da due componenti: delle *application programming interfaces* multipiattaforma unificate che forniscono un'interfaccia alle applicazioni tramite il *plugin* e le implementazioni specifiche per le piattaforme Android (in Kotlin) e iOS (in Swift) costruite su ARCore e ARKit rispettivamente, così da garantire accesso continuativo nel tempo a funzionalità aggiornate.
ar_flutter_plugin espone dei *widget* che possono essere inclusi nel *widget tree* dell'applicazione cliente e delle classi *manager* per gestire funzionalità e logiche di controllo del *plugin* e delle componenti in realtà aumentata: *session manager*, *object manager*, *anchor manager* e *location manager*.
Il *session manager* gestisce le configurazioni di tracciamento (ad esempio se usare piani orizzontali, verticali o entrambi), opzioni di *debugging* (come la visualizzazione dei piani) e le *callbacks* per gli *hit-test*[15] e i *gesture events*[16].

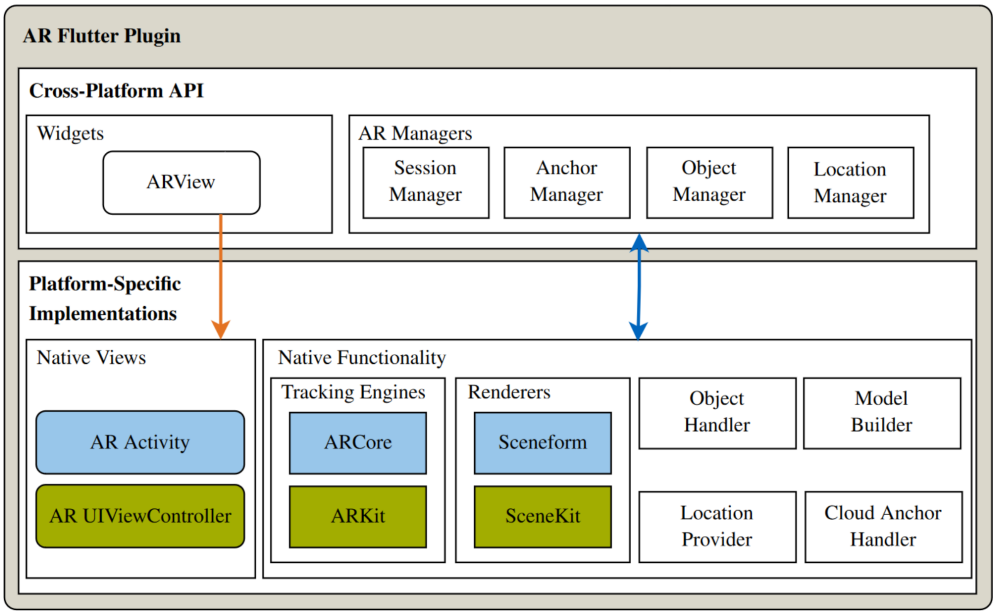


Figura 3.8: Schematizzazione logica interna di ar_flutter_plugin.[17]

[14]Fonte: https://lars.carius.io/blog/ar_flutter_plugin.html
[15]Immaginando un vettore che parte dall'ottica del dispositivo e tocca una superficie, possiamo considerare quel contatto un *"hit"*. Con *hit-testing* si intende valutare la capacità di tracciare correttamente lo spazio inteso come distanza tra i punti dello stesso e il dispositivo.
[16]Azioni che il sistema deve fare quando vengono effettuati degli *input* tramite *touch screen* del dispositivo

L'*object manager* gestisce i nodi (le entità) in realtà aumentata del *plugin* che servono ad astrarre rispetto alle funzionalità native delle varie piattaforme (ARCore e ARKit ad esempio) e permettono di aggiungere, modificare o rimuovere nodi basandosi su formati GLTF2 o GLB che possono essere caricati a *runtime* da un *file system* o dalla rete.
L'*anchor manager* contiene funzioni per caricare e ottenere ancoraggi da servizi *cloud* servendosi delle *application programming interfaces* fornite da Google Cloud Anchor. Gli ambienti virtuali registrati localmente possono essere salvati in rete e comparati con quelli già in *storage* persistente per scaricare ancoraggi e oggetti precedentemente posizionati nella scena.
Infine il *location manager* si occupa di fornire le coordinate del *global positioning system* per permettere un'interrogazione efficiente degli ancoraggi basati su posizione geografica.
Il *plugin* ha un'architettura largamente *clout-agnostic*, cioè non si interessa dell'implementazione specifica per i servizi di salvataggio persistente, il che gli permette di usare facilmente gestori di contenuti esterni.

Confronto

Schematizzo quindi pregi e difetti delle due soluzioni trovate per metterle a confronto.

Pregi ARwayKit:

- Implementa nativamente le Azure Spatial Anchors;
- Funziona in ambienti *GPS-denied*;
- È pensato per uso aziendale.

Difetti ARwayKit:

- La vista e le componenti in realtà aumentata non sono implementate in Flutter;
- Richiede l'uso di Unity che, essendo un motore per videogiochi, ha funzionalità in più (e in meno) non necessarie (e necessarie) rispetto a un uso aziendale;
- Ha una documentazione difficile da reperire e di dubbia completezza;
- È codice proprietario, quindi non c'è modo di visionare il sorgente.

Pregi ar_flutter_plugin:

- La vista e le componenti in realtà aumentata sono implementate in Flutter;
- Ha un documentazione discreta;
- È *open-source*, quindi al bisogno è possibile visionare il codice sorgente;
- È pensato per uso aziendale;
- La struttura logica, divisa nei vari *manager*, è modulare e chiara da comprendere, il che dovrebbe facilitarne modifica ed estensione;

[17] Fonte: https://lars.carius.io/blog/ar_flutter_plugin.html

- Promette di essere *cloud-agnostic*, quindi favorisce l'uso di gestori esterni come ad esempio Azure Spatial Anchors

Difetti ar_flutter_plugin:

- Non implementa nativamente le Azure Spatial Anchors, usando invece le Google Cloud Anchors;
- Non è chiaro se funzioni nativamente in ambienti *GPS-denied*;

Confronto schematico:

Caratteristica	ARwayKit	Plugin
Azure Spatial Anchors native	SI	NO
Ambienti *GPS-denied*	SI	NON CHIARO
Vista realtà aumentata implementata in Flutter	NO	SI
Codice sorgente visibile	NO	SI
Pensato per uso aziendale	SI	SI

Tabella 3.1: Confronto *framework* per realtà aumentata

Ho infine scelto, in comune accordo con il *tutor* aziendale, ar_flutter_plugin rispetto ad ARwayKit per evitare di interfacciarsi con Unity, per la necessità di avere la vista in realtà aumentata implementata direttamente in Flutter e per il grande vantaggio che comporta il suo essere *open-source*.

3.2 Analisi dei requisiti

Di seguito riporto i requisti individuati nel piano di lavoro proposto (a seguito di opportuni confronti con il *tutor* aziendale). Sono catalogati secondo la dicitura:

R[Obbligatorietà][Tipologia][Codice]

Dove:

- **Obbligatorietà**: specifica quanto un requisito sia vincolante per la riuscita del prodotto e può assumere i seguenti valori:
  - **1:** requisito obbligatorio;
  - **2:** requisito desiderabile ma non essenziale per il funzionamento;
  - **3:** requisito opzionale;
- **Tipologia**: specifica la tipologia del requisito e può assumere i seguenti valori:
  - **F:** *funzionale,* determina una funzionalità necessaria all'applicazione;
  - **V:** *vincolo,* riguarda una caratteristica del prodotto decisa a monte;
- **Codice**: identifica univocamente un requisito all'interno della sua tipologia (ovvero possono esistere due requisiti con lo stesso codice a patto che siano uno funzionale e uno di vincolo). Per i requisti subordinati si usa il "punto" come divisorio (*ReqPadre, ReqPadre.Figlio1*).

3.2.1 Requisiti funzionali

Codice	Descrizione
R1F1	Il *plugin* deve rappresentare *asset* tramite ancoraggio in realtà aumentata
R1F2	Il *plugin* deve rappresentare *ticket* tramite ancoraggio in realtà aumentata
R1F3	Il *plugin* deve integrare gli ancoraggi tramite Azure Spatial Anchors
R1F3.1	Permettere aggiunta di Azure Spatial Anchors
R1F3.2	Permettere recupero e visualizzazione di Azure Spatial Anchors
R2F3.3	Permettere modifica di Azure Spatial Anchors
R1F3.4	Permettere eliminazione di Azure Spatial Anchors
R1F4	Comunicare con le *application programming interfaces* di Syn
R1F4.1	Ricevere *asset* con ancoraggio associato
R1F4.2	Aggiungere *asset* con ancoraggio associato
R2F4.3	Ricevere *ticket* con ancoraggio associato
R2F4.4	Aggiungere *ticket* con ancoraggio associato
R1F5	Utente deve poter vedere quali *asset* hanno ancoraggio associato
R2F6	Utente deve poter vedere quali *ticket* hanno ancoraggio associato
R1F7	Utente deve poter raggiungere la vista in realtà aumentata dalla schermata degli *asset*
R1F8	*On-Tap* su una ancoraggio deve aprire una *bottom sheet* contestuale
R1F8.1	*Bottom sheet* deve presentare identificatore per *asset* o *ticket* associato all'ancoraggio
R1F8.2	*Bottom sheet* associato a un *asset* mostra *ticket* aperti se presenti
R1F8.3	*Bottom sheet* deve fornire *Call-To-Action* per eliminare l'ancoraggio
R1F8.4	*Bottom sheet* deve fornire *Call-To-Action* per raggiungere pagina di dettaglio
R1F9	Le informazioni contestuali di un *ticket* includono data e ora di creazione
R1F10	Gli ancoraggi hanno rappresentazione visiva contestuale
R1F11	*On-Tap* sullo spazio permette di creare una ancoraggio in posizione
R1F11.1	Ancoraggio posizionato nello spazio può essere salvato
R1F11.2	Ancoraggio posizionato nello spazio può essere eliminato
R1F12	Il salvataggio di un ancoraggio è disponibile solo quando è sicuro vada a buon fine

Codice	Descrizione
R1F12.1	Viene mostrato a schermo un feedback riguardo il livello di sicurezza raggiunto

Tabella 3.2: Tabella dei requisiti funzionali

3.2.2 Requisiti di vincolo

Codice	Descrizione
R1V1	*Framework* scelto funziona su Android
R2V2	*Framework* scelto funziona su iOS
R1V3	*Framework* scelto si integra con le *application programming interfaces* di Syn
R1V3.1	*Framework* ottiene con ancoraggio associato i dati degli *asset*
R1V3.2	*Framework* ottiene con ancoraggio associato i dati dei *ticket*
R1V4	Il *framework* scelto utilizza Azure Spatial Anchors
R1V5	La vista in realtà aumentata deve essere sviluppata in Flutter

Tabella 3.3: Tabella dei requisiti di vincolo

3.2.3 Riepilogo requisiti

Ho individuato un totale di 36 requisiti, 29 funzionali e 7 di vincolo, di seguito schematizzati:

Obbligatorietà	Quantità
Obbligatori	31
Desiderabili	5

Tabella 3.4: Numero di requisiti per obbligatorietà

Tipologia	Quantità
Funzionali	29
Di Vincolo	7

Tabella 3.5: Numero di requisiti per tipologia

3.3 Progettazione

Tratterò la progettazione del sistema da due punti di vista: uno architetturale, che quindi pone l'accento su come il sistema sottostante svolga il lavoro di comunicazione con Azure Spatial Anchors e con le *application programming interfaces* di Syn, e uno orientato invece all'aspetto grafico dell'applicazione, ovvero l'interfaccia utente.

3.3.1 Architettura

L'idea è di avere due sistemi comunicanti (gestiti con due diverse *repository*): l'applicazione cliente MobileSYN che si interfaccia con Syn e il *plugin* scelto che invece si occupa di gestire la parte in realtà aumentata e si interfaccia con Azure Spatial Anchors.
Bisogna fare chiarezza su un punto: Syn ha già in sistema degli *asset* geolocalizzati tramite Azure Spatial Anchors, cioè quindi gli *asset* sono legati a un ancoraggio del servizio di Microsoft, però questo è indipendente dal fatto che è sempre con Azure Spatial Anchors che implementiamo le componenti in realtà aumentata (in sostanza non è obbligatorio lavorare in *augmented reality* per sfruttare gli ancoraggi di Azure). Avremo quindi, a livello logico, MobileSYN che otterrà *ticket* e *asset* da Syn e poi li invierà ad ar_flutter_plugin il quale si occuperà di mostrarli in realtà aumentata, fornendo anche la vista.

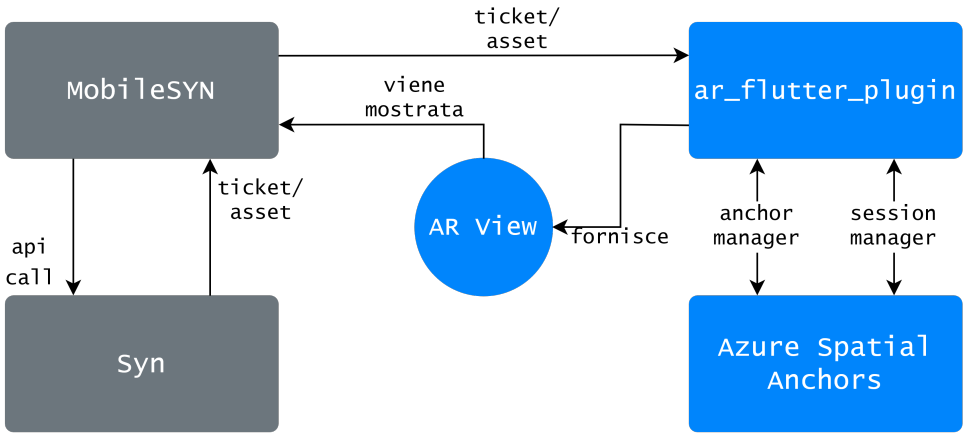


Figura 3.9: Rappresentazione dell'architettura dell'applicazione. MobileSYN fornisce *ticket* e *asset* a ar_flutter_plugin che di ritorno comunica con Azure Spatial Anchors per fornire all'applicazione cliente la vista in realtà aumentata.

3.3.2 Interfaccia utente

Mostro dei *mockup* che danno un'idea di massima di come l'interfaccia utente dovrebbe presentarsi: dai requisiti, l'utente deve poter posizionare ancoraggi, caricarli sul *cloud* quando sono stati ottenuti sufficienti dati spaziali e, all'*on-tap* su una *anchor*, deve vedere una *bottom sheet* che gli presenti informazioni riguardo all'*asset* collegato (ad esempio i *ticket* aperti) e che gli permetta di raggiungere i dettagli dell'*asset* o di eliminare l'*anchor*.
Inoltre nella lista degli *asset* devono essere presenti sia un'icona che identifichi quali

hanno un ancoraggio associato posizionato in realtà aumentata che un bottone (in alto a destra in questo caso) che permetta di lanciare la vista.

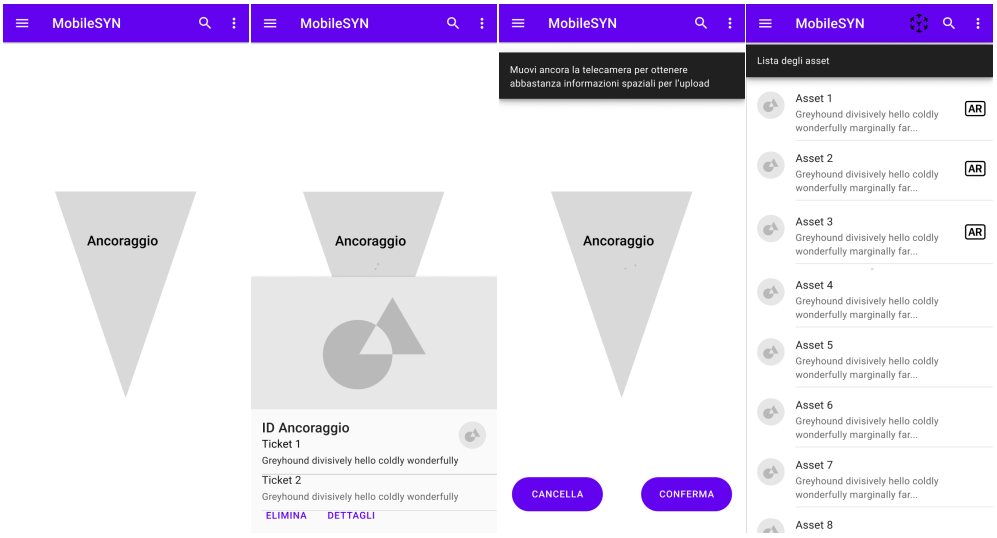


Figura 3.10: *Mockup* per l'applicazione MobileSYN. In ordine: ancoraggio visto dalla telecamera, *bottom sheet* contestuale che appare all'*on-tap* sull'*anchor*, schermata per caricare una *anchor* in *cloud*, lista degli *asset*.

3.4 Codifica

Mostro alcuni frammenti di codice che cercano di dare un'idea di massima di come, nel pratico, ho implementato i componenti in realtà aumentata. Il codice mostrato dovrà essere estremamente parziale, pena la creazione di un documento di diverse centinaia di pagine.
Il tutto è gestito con due *repository* parallele: "mobilesyn" (che contiene il sorgente dell'applicazione vera e propria) e un *fork* di "ar_flutter_plugin" alla quale sono state apportate le modifiche necessarie per interfacciarsi con SYN e le Azure Spatial Anchors. Partiamo dalla prima.
Vediamo un esempio di *asset provider:*

```
Future<void> fetchAssets(BusinessFunction? bf, GeoContainer? geoc) async
  {
  debugPrint("FETCH-ASSETS");
  state = const AsyncValue.loading();
  try {
    final response = await read(synApiProvider).fetchAssets(
        count: 200, skip: 0, functionId: bf?.id, geocId: geoc?.id);
    state = AsyncValue.data(response);
  } catch (e) {
    debugPrint(e.toString());
    state = const AsyncValue.error('error_assets');
  }
}
```

Listing 3.1: mobilesyn *asset provider*

E di *ticket provider*:

```
Future<void> fetchTickets(String flowId, bool closed) async {
  debugPrint("FETCH-TICKETS");
  state = const AsyncValue.loading();
  try {
    var tpl = read(ticketsfiltersProvider).template;
    var since = read(ticketsfiltersProvider).since;
    var till = read(ticketsfiltersProvider).till;
    var geoc = read(selectedGeocP);
    final tickets = await read(synApiProvider).fetchTickets(
        skip: 0,
        count: 100,
        flowId: flowId,
        closed: closed,
        tplId: tpl?.id,
        since: closed ? since : null,
        till: closed ? till : null,
        geocId: geoc?.id);
    final List<TicketAssetInfo> temp =
        tickets.map((t) => t.asset).whereType<TicketAssetInfo>()
                                   .toList();
    final strings = temp.map((e) => e.toJson()).toList().toSet()
                                    .toList();
    final List<TicketAssetInfo> assets =
        strings.map((e) => TicketAssetInfo.fromJson(e)).toList();
    final FeatureList featured = FeatureList.fromTicketList(tickets);
    state = AsyncValue.data(TicketStateModel(
        tickets: tickets, assets: assets, featured: featured));
  } catch (e) {
    debugPrint(e.toString());
    state = const AsyncValue.error('error_asset_tickets');
  }
}
```

Listing 3.2: mobilesyn *ticket provider*

Notare, per entrambi, l'uso di un `Future`. Nella pagina che si occupa di mostrare i dettagli di un *asset* abbiamo una funzione specifica per ottenere i *ticket* a esso associati:

```
Future<void> fetchAssetTickets() async {
  debugPrint('FETCH ASSET TICKETS - ${asset.cod}');
  try {
    loadingTickets.value = true;
    errorTickets.value = false;
    final response = await ref
        .read(synApiProvider)
        .fetchTickets(skip: 0, count: 3, assetId: asset.id!);
    loadingTickets.value = false;
    errorTickets.value = false;
    tickets.value = response;
  } catch (e) {
    debugPrint(e.toString());
    loadingTickets.value = false;
    errorTickets.value = true;
    tickets.value = [];
  }
}
```

Listing 3.3: mobilesyn *asset ticket provider*

Per quanto riguarda invece i *manager* delle componenti in realtà aumenta essi vengono presi da ar_flutter_plugin:

```dart
import 'package:ar_flutter_plugin/managers/ar_anchor_manager.dart';
import 'package:ar_flutter_plugin/managers/ar_session_manager.dart';

class ManagementArView extends HookConsumerWidget {
  const ManagementArView({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final arSessionManager = useState<ARSessionManager?>(null);
    final arAnchorManager = useState<ARAnchorManager?>(null);
```

Listing 3.4: mobilesyn *managers*

E sono presenti diverse funzioni di utilità, ad esempio il seguente codice viene chiamato alla creazione della vista:

```dart
void onARViewCreated(
  ARSessionManager _arSessionManager, ARAnchorManager _arAnchorManager)
    {
arSessionManager.value = _arSessionManager;
arAnchorManager.value = _arAnchorManager;
arAnchorManager.value!.onAssetTap = onAssetTap;
arAnchorManager.value!.onTicketTap = onTicketTap;
arViewInitialized.value = true;
}
```

Listing 3.5: mobilesyn *on ar view created*

Mostro ora alcuni frammenti dei *manager* di ar_flutter_plugin.
Riporto i metodi per creazione ed eliminazione di *anchor* localmente e per *upload* ed eliminazione di ancoraggi in *cloud* presenti nel *manager* per gli ancoraggi (si noti che tutti i metodi usano i *method channels*):

```dart
Future<void> createAnchor(
    {required Matrix4 transformation,
    required Map<String, dynamic> info}) async {
  return await _channel.invokeMethod<void>('createAnchor', {
    "transformation": MatrixConverter().toJson(transformation),
    "info": info
  });
}

/// Remove given anchor and all its children from the AR Scene
Future<void> deleteAnchor({required String anchorId}) async {
  return await _channel.invokeMethod<void>('deleteAnchor',
                                           {'id': anchorId});
}

/// Upload the latest added Anchor to the ASA Cloud
Future<String?> uploadAnchor() async {
  try {
    return await _channel.invokeMethod<String?>('uploadAnchor');
  } on PlatformException catch (_) {
    return null;
  }
}

/// Delete the given anchor from the ASA Cloud and the AR Scene
Future<bool?> deleteCloudAnchor({required String anchorId}) async {
  try {
    return await _channel
        .invokeMethod<bool?>('deleteCloudAnchor', {'id': anchorId});
```

```
  } on PlatformException catch (_) {
    return null;
  }
}
```

Listing 3.6: ar_flutter_plugin *create, delete, ulpoad, delete cloud anchor* tramite *method channel*

Queste chiamate in Android sono implementati tramite chiamate Java innestate in Kotlin:

```
private val onAnchorMethodCall =
MethodChannel.MethodCallHandler { call, result ->
    when (call.method) {
        "createAnchor" -> {
            val dictInfo = call.argument("info") as HashMap<String,
                                                                Any>?
            val transform = call.argument("transformation")
                                as ArrayList<Double>?
            createAnchor(transform!!, AnchorInfo(dictInfo!!))
            result.success(null)
        }
        "uploadAnchor" -> {
            uploadAnchor(result)
        }
        "deleteAnchor" -> {
            val infoId = call.argument("id") as String?
            deleteAnchor(infoId!!)
            result.success(null)
        }
        "deleteCloudAnchor" -> {
            val infoId = call.argument("id") as String?
            deleteCloudAnchor(infoId!!, result)
        }
        else -> {
            Log.d(TAG, call.method + "not supported on anchorManager")
        }
    }
}
```

Listing 3.7: Android chiamate dei *method channel* per effettuare *create, delete, ulpoad, delete cloud anchor*

Mentre per quanto riguarda iOS sono implementate in Swift:

```
func onAnchorMethodCalled(_ call: FlutterMethodCall, _ result:
                                    @escaping FlutterResult) {
  let arguments = call.arguments as? [String: Any]
  NSLog("IosARView: onAnchorMethodCalled \(call.method)")
  switch call.method {
      case "createAnchor":
          let dictInfo = arguments?["info"] as! [String: Any]
          let transform = arguments?["transformation"] as! [NSNumber]
          createAnchor(transform: transform, info:
                                            AnchorInfo(val: dictInfo))
          result(nil)
      case "uploadAnchor":
          uploadAnchor(result: result)
      case "deleteAnchor":
          let infoId = arguments!["id"] as! String
          deleteAnchor(infoId: infoId)
          result(nil)
      case "deleteCloudAnchor":
```

```
            let infoId = arguments!["id"] as! String
            deleteCloudAnchor(infoId: infoId, result: result)
        default:
            result(FlutterMethodNotImplemented)
    }
}
```

Listing 3.8: iOS chiamate dei *method channel* per effettuare *create, delete, ulpoad, delete cloud anchor*

Mostro ora un piccolo esempio di come ho implementato, in Android, le Azure Spatial Anchors. Non porterò in questo caso l'esempio speculare di iOS in quanto poco interessante: come si vede dai frammenti di codice precedenti le differenze sono più tecniche che macroscopiche.
Vediamo l'eliminazione di un ancoraggio salvato in *cloud*, riprendendo il codice in 3.7 guardiamo il flusso che provoca la chiamata `"deleteCloudAnchor"`:

```
private fun deleteCloudAnchor(infoId: String,
                              result: MethodChannel.Result) {
  Log.d(TAG, "removeCloudAnchor $infoId")
  val visual = anchorVisuals[infoId]
  if (visual?.cloudAnchor != null) {
      azureSpatialAnchorsManager!!.
      deleteAnchorAsync(visual.cloudAnchor!!)

  //la funzione prosegue ma non ci interessa per questa dimostrazione
```

Listing 3.9: Eliminazione *cloud anchor* lato Android, chiamata

È stato chiamato in causa il *manager* per le Azure Spatial Anchors, e seguendone il flusso:

```
internal class AzureSpatialAnchorsManager(arCoreSession: Session?,
                                          apiKey: String, apiId: String)
    {
  private val executorService: ExecutorService = Executors.
                                                 newFixedThreadPool(2)
  var isRunning = false
  private val spatialAnchorsSession: CloudSpatialAnchorSession
  // Set this string to the account ID provided
  //for the Azure Spatial Anchors account resource.
  private val SpatialAnchorsAccountId = apiId

  // Set this string to the account key provided
  //for the Azure Spatial Anchors account resource.
  private val SpatialAnchorsAccountKey = apiKey

  //ora viene chiamata l'effettiva eliminazione
  fun deleteAnchorAsync(anchor: CloudSpatialAnchor?):
                               CompletableFuture<*> {
        return toEmptyCompletableFuture(spatialAnchorsSession.
                                        deleteAnchorAsync(anchor))
  }
```

Listing 3.10: Eliminazione *cloud anchor* lato Android, chiamata

Il codice di cui sopra mostra che l'applicazione comunica con le *application programming interfaces* di SYN per ottenere *ticket* e *asset*, e che ottiene gli ancoraggi e che questi ancoraggi sono gestiti tramite Azure Spatial Anchors. Tutto questo, inoltre, è stato implementato sia in Android che in iOS.

Mostro infine una piccola parte del codice che si occupa di creare l'effettiva interfaccia utente della vista in realtà aumentata:

```
/// Android-specific implementation of [PlatformARView]
class AndroidARView implements PlatformARView {
  late BuildContext? _context;
  late ARViewCreatedCallback? _arViewCreatedCallback;

  @override
  void onPlatformViewCreated(int id) {
    print("Android platform view created!");
    createManagers(id, _context, _arViewCreatedCallback);
  }

  @override
  Widget build(
      {BuildContext? context,
      ARViewCreatedCallback? arViewCreatedCallback,
      String? apiKey,
      String? apiId,
      List<Map<String, dynamic>>? assets,
      List<Map<String, dynamic>>? tickets}) {
      \\altro codice
      }
  \\altro codice
}


/// iOS-specific implementation of [PlatformARView]
class IosARView implements PlatformARView {
  BuildContext? _context;
  ARViewCreatedCallback? _arViewCreatedCallback;

  @override
  void onPlatformViewCreated(int id) {
    print("iOS platform view created!");
    createManagers(id, _context, _arViewCreatedCallback);
  }

  @override
  Widget build(
      {BuildContext? context,
      ARViewCreatedCallback? arViewCreatedCallback,
      String? apiKey,
      String? apiId,
      List<Map<String, dynamic>>? assets,
      List<Map<String, dynamic>>? tickets}) {
      \\altro codice
      }
  \\altro codice
}
```

Listing 3.11: Frammento di codice per vista in realtà aumenta in Dart

3.5 Verifica

Non ho implementato nessun sistema di verifica standardizzato, quindi mi è impossibile mostrare tabelle di *test* di unità, integrazione, sistema o accettazione e, di conseguenza, è anche impossibile effettuare delle verifiche calcolabili come *test* di *code coverage*. Posso tuttavia verificare la copertura dei requisti tramite il codice e le *scrennshot*

dell'applicazione proposte.
I frammenti 3.1, 3.2, 3.3 mostrano i *provider* di MobileSYN che servono a comunicare con le *application programming interfaces* di Syn, e ottengono quindi i dati relativi a *ticket* e *asset*. I frammenti 3.4, 3.5 mostrano che MobileSYN ottiene i *manager* per la realtà aumentata da ar_flutter_plugin.

Codice	Descrizione
R1V3	*Framework* scelto si integra con le *application programming interfaces* di Syn
R1V3.1	*Framework* ottiene con ancoraggio associato i dati degli *asset*
R1V3.2	*Framework* ottiene con ancoraggio associato i dati dei *ticket*
R1F4	Comunicare con le *application programming interfaces* di Syn
R1F4.1	Ricevere *asset* con ancoraggio associato
R1F4.2	Aggiungere *asset* con ancoraggio associato
R2F4.3	Ricevere *ticket* con ancoraggio associato
R2F4.4	Aggiungere *ticket* con ancoraggio associato

Tabella 3.6: Requisiti soddisfatti nei frammenti: 3.1, 3.2, 3.3, 3.4, 3.5.

I frammenti 3.6, 3.7, 3.8 mostrano i *method channel* per lanciare le chiamate per agire sugli ancoraggi in codice nativo nel quale, come viene mostrato dai frammenti 3.9 e 3.10, ho implementato le *anchor* tramite Azure Spatial Anchors.

Codice	Descrizione
R1V1	*Framework* scelto funziona su Android
R2V2	*Framework* scelto funziona su iOS
R1V4	Il *framework* scelto utilizza Azure Spatial Anchors
R1F3	Il *plugin* deve integrare gli ancoraggi tramite Azure Spatial Anchors
R1F3.1	Permettere aggiunta di Azure Spatial Anchors
R1F3.2	Permettere recupero e visualizzazione di Azure Spatial Anchors
R2F3.3	Permettere modifica di Azure Spatial Anchors
R1F3.4	Permettere eliminazione di Azure Spatial Anchors

Tabella 3.7: Requisiti soddisfatti nei frammenti: 3.6, 3.7, 3.8, 3.9, 3.10.

Il frammento di codice 3.11 mostra che le viste in realtà aumenta per Android e iOS sono implementate in Flutter.

Codice	Descrizione
R1V5	La vista in realtà aumentata deve essere sviluppata in Flutter

Tabella 3.8: Requisiti soddisfatti nel frammento: 3.11

L'immagine 3.11 mostra tre *screenshot* dell'applicazione dove un ancoraggio viene creato, salvato in *cloud* quando si sono ottenuti abbastanza dati spaziali e poi viene recuperato alla riapertura dell'applicazione.

Codice	Descrizione
R1F1	Il *plugin* deve rappresentare *asset* tramite ancoraggio in realtà aumentata
R1F11	*On-Tap* sullo spazio permette di creare una ancoraggio in posizione
R1F11.1	Ancoraggio posizionato nello spazio può essere salvato
R1F11.2	Ancoraggio posizionato nello spazio può essere eliminato
R1F12	Il salvataggio di un ancoraggio è disponibile solo quando è sicuro vada a buon fine
R1F12.1	Viene mostrato a schermo un feedback riguardo il livello di sicurezza raggiunto

Tabella 3.9: Requisiti soddisfatti in figura 3.11

L'immagine 3.12 mostra che effettuando un *on-tap* su un ancoraggio vengono mostrate le corrette informazioni contestuali e viene permesso tramite bottoni di raggiungere i dettagli dell'*asset* o di eliminare l'*anchor*. Inoltre nella lista degli *asset* quelli che sono già collegati a un ancoraggio in realtà aumentata lo comunicano tramite un'icona con dicitura "AR" e che è presente il bottone per raggiungere la vista in *augmented reality*.

Codice	Descrizione
R1F5	Utente deve poter vedere quali *asset* hanno ancoraggio associato
R1F7	Utente deve poter raggiungere la vista in realtà aumentata dalla schermata degli *asset*
R1F8	*On-Tap* su una ancoraggio deve aprire una *bottom sheet* contestuale
R1F8.1	*Bottom sheet* deve presentare identificatore per *asset* o *ticket* associato all'ancoraggio
R1F8.2	*Bottom sheet* associato a un *asset* mostra *ticket* aperti se presenti
R1F8.3	*Bottom sheet* deve fornire *Call-To-Action* per eliminare l'ancoraggio
R1F8.4	*Bottom sheet* deve fornire *Call-To-Action* per raggiungere pagina di dettaglio

Codice	Descrizione

Tabella 3.10: Requisiti soddisfatti in figura 3.12

L'immagine 3.13 mostra che è possibile posizionare un *ticket* in realtà aumentata e che i *ticket* relativi a un *anchor* mostrano data e ora di apertura.

Codice	Descrizione
R1F2	Il *plugin* deve rappresentare *ticket* tramite ancoraggio in realtà aumentata
R1F9	Le informazioni contestuali di un *ticket* includono data e ora di creazione
R1F10	Gli ancoraggi hanno rappresentazione visiva contestuale

Tabella 3.11: Requisiti soddisfatti in figura 3.13

In conclusione, l'unico requisito rimasto insoddisfatto è il seguente:

Codice	Descrizione
R2F6	Utente deve poter vedere quali *ticket* hanno ancoraggio associato

Tabella 3.12: Requisiti non soddisfatti

Portando quindi la copertura totale dei requisiti a un livello più che soddisfacente. Mostro quindi una sintesi, in tabella singola, dei requisti soddisfatti e dove vederne la dimostrazione (figure o frammenti di codice).

Codice requisito	Dove trovarne il soddisfacimento
R1V1	3.6, 3.7, 3.8, 3.9, 3.10
R2V2	3.6, 3.7, 3.8, 3.9, 3.10
R1V3	3.1, 3.2, 3.3, 3.4, 3.5
R1V3.1	3.1, 3.2, 3.3, 3.4, 3.5
R1V3.2	3.1, 3.2, 3.3, 3.4, 3.5
R1V4	3.6, 3.7, 3.8, 3.9, 3.10
R1V5	3.11
R1F1	3.11
R1F2	3.13
R1F3	3.6, 3.7, 3.8, 3.9, 3.10

Codice requisito	Dove trovarne il soddisfacimento
R1F3.1	3.6, 3.7, 3.8, 3.9, 3.10
R1F3.2	3.6, 3.7, 3.8, 3.9, 3.10
R2F3.3	3.6, 3.7, 3.8, 3.9, 3.10
R1F3.4	3.6, 3.7, 3.8, 3.9, 3.10
R1F4	3.1, 3.2, 3.3, 3.4, 3.5
R1F4.1	3.1, 3.2, 3.3, 3.4, 3.5
R1F4.2	3.1, 3.2, 3.3, 3.4, 3.5
R2F4.3	3.1, 3.2, 3.3, 3.4, 3.5
R2F4.4	3.1, 3.2, 3.3, 3.4, 3.5
R1F5	3.12
R2F6	Non soddisfatto
R1F7	3.12
R1F8	3.12
R1F8.1	3.12
R1F8.2	3.12
R1F8.3	3.12
R1F8.4	3.12
R1F9	3.13
R1F10	3.13
R1F11	3.11
R1F11.1	3.11
R1F11.2	3.11
R1F12	3.11
R1F12.1	3.11

Tabella 3.13: Tabella che mette in relazione i requisiti e dove si vede il loro soddisfacimento.

3.6 Risultati raggiunti

In quest'ultima sezione riporto il risultato finale del lavoro di stage tramite *screenshot* di MobileSYN stessa.
Le seguenti immagini mostrano il progresso in vista in realtà aumentata che porta a creare un ancoraggio relativo all'*asset* "IT-TESTP-ELight0324" (privo di eventi o *ticket* correlati), per poi caricarlo in *cloud* e, successivamente a chiusura e riapertura

dell'applicazione, ritrovarlo in posizione (si noti che è leggermente arretrato).
Inoltre nella prima immagine, viene anche mostrato il livello di progressione di *scan* dell'ambiente, necessario a caricare correttamente un ancoraggio.

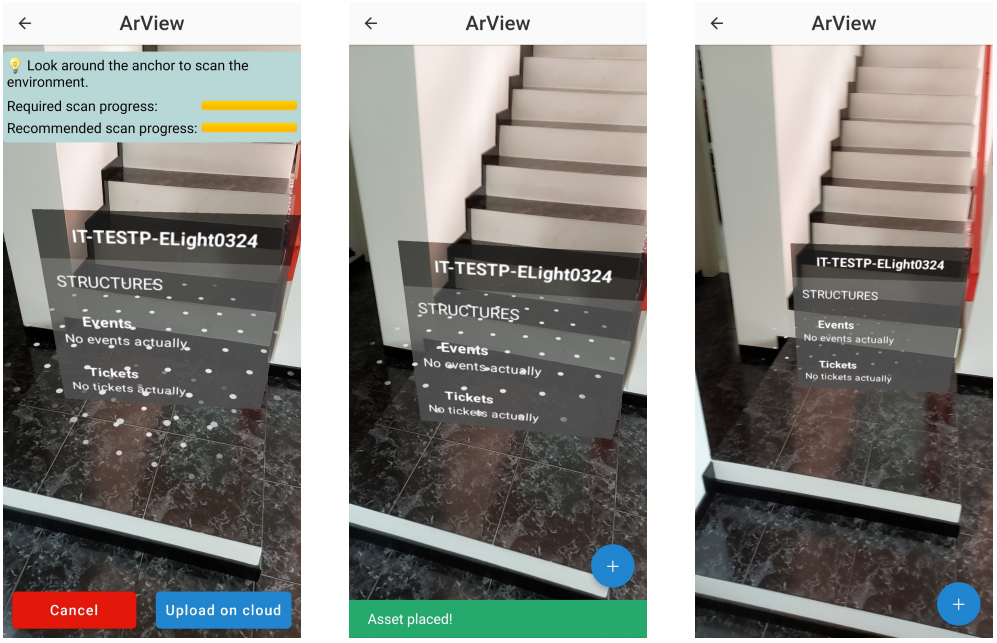


Figura 3.11: In ordine: *asset* viene posizionato e viene mostrato il livello di *scan* dell'ambiente, viene caricato e, previa chiusura e riapertura della vista, viene recuperato dal *cloud*.

Effettuando un *On-Tap* su un ancoraggio appare una *bottom sheet* che mostra l'identificatore dell'*asset* e, se ci sono, gli eventi e i *ticket* aperti a lui associati. Permette inoltre di raggiungere i dettagli dell'*asset* o di rimuoverlo.
Poi, nella lista degli *asset* disponibili, quelli già collegati a un ancoraggio in realtà aumentata presentano un piccolo marcatore con dicitura "AR", permettendo di raggiungere la vista con il bottone a forma di cubo in altro a destra. È grigio perché l'uso richiederebbe di essere fisicamente presenti in "Plant 1" e di selezionare inoltre stabilimento, piano e stanza.

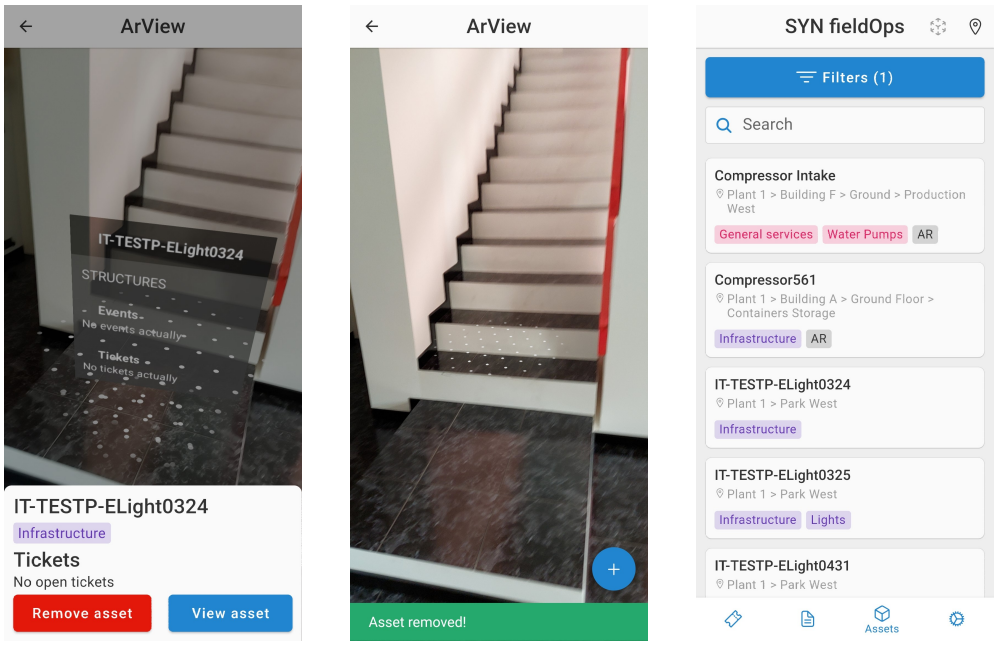


Figura 3.12: *Bottom sheet* per l'*asset* "IT-TESTP-ELight0324" e successiva eliminazione tramite bottone con dicitura *"Remove Asset"*. L'ultima immagine mostra invece la lista degli *asset* presenti con il bottone per raggiungere la vista in realtà aumentata.

Per quanto riguarda i *ticket* invece l'approccio che ho scelto è quello di far apparire, al completamento della compilazione di un *ticket*, una finestra che permette di posizionarlo in realtà aumentata. In questo caso si può notare la *card* di stato aggiornata per quanto riguarda la notifica del livello di *scan* per il caricamento dell'ancoraggio. Inoltre che i *ticket* aperti di un *asset* mettono in chiaro data e ora di apertura.

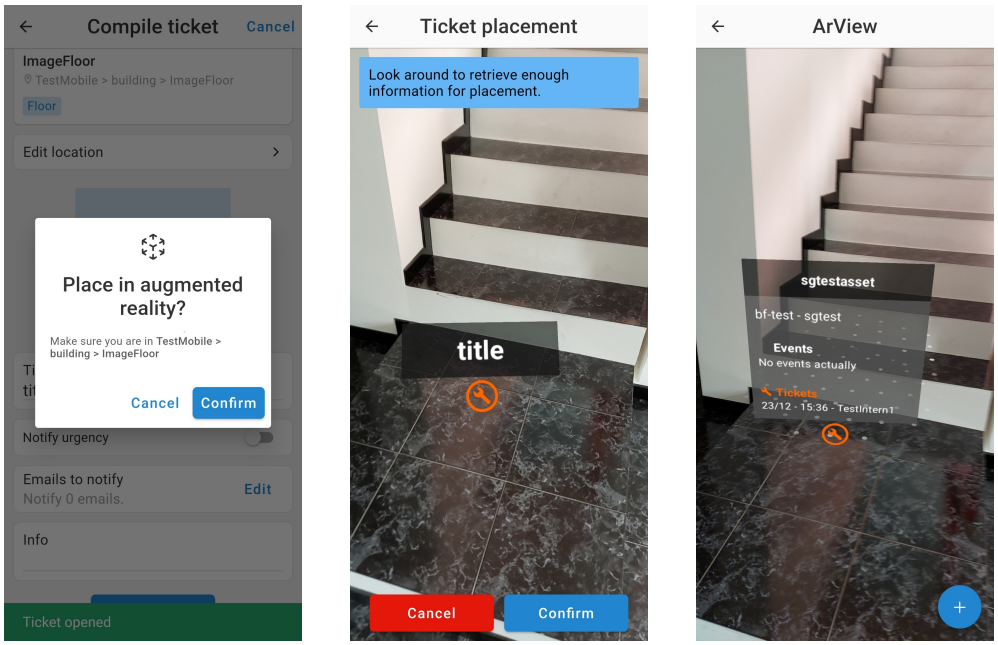


Figura 3.13: Una volta compilato il *ticket* appare una finestra in sovrimpressione che permette di posizionarlo in realtà aumentata. Inoltre i *ticket* di un *asset* mostrano data e ora di apertura.

Capitolo 4

Valutazione retrospettiva

4.1 Difficoltà incontrate

Le prime difficoltà che ho incontrato sono state relative ai linguaggi: non avevo mai programmato in Dart e soprattutto non mi ero mai trovato a dover gestire la caratteristica che, quando innestato in Flutter, viene usato sia per le logiche di controllo che per produrre componenti grafiche dell'interfaccia utente. A questo si è aggiunta la necessità di dover usare altri tre linguaggi per le implementazioni native, ovvero Java e Kotlin per il lato Android e Swift per la parte iOS, che ho poi dovuto mettere in comunicazione diretta con Flutter (e nel caso di Java e Kotlin anche tra di loro).
Ho quindi speso una buona parte dello *stage* a comprendere e gestire la grande varietà di linguaggi diversi.

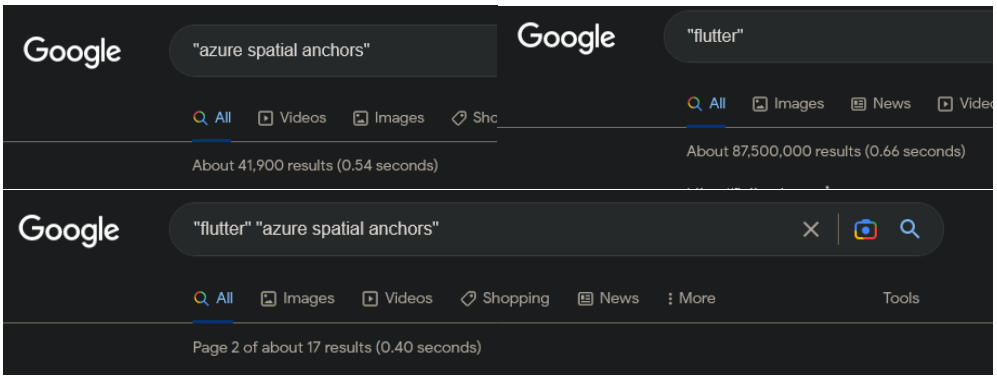


Figura 4.1: Al 23 novembre 2022, una ricerca esatta di "flutter" mostra 87.5 milioni di risultati rilevanti, di "azure spatial anchors" 41 milioni mentre la combinazione delle due, solo 17

Le due problematiche che ritengo più rilevanti, però, sono state la mancanza di documentazione adeguata e la mancanza di supporto da parte della comunità degli sviluppatori.
Microsoft non fornisce un *set* documentale adeguato per quanto riguarda Azure Spatial Anchors, mostrando il meno possibile della struttura interna (ad esempio come viene rappresentato un ancoraggio) e fornendo solo *application programming interfaces* per effettuare operazioni ad alto livello (come il salvataggio in *cloud* di un'*anchor*), inoltre la ricerca della documentazione è macchinosa.
L'altro problema risiede nella difficoltà estrema di trovare supporto di terze parti (ad esempio in siti come https://stackoverflow.com), come mostro nelle figure 4.1 e 4.2.

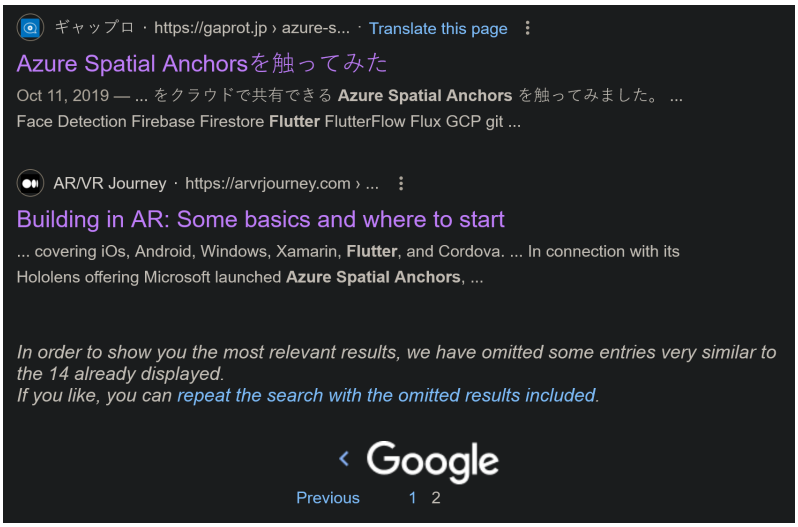


Figura 4.2: All'otto febbraio 2023 non si notano miglioramenti: solo 14 risultati rilevanti, alcuni in lingua giapponese, mostrano quando sia difficile trovare informazioni sull'argomento.

Ho quindi dovuto, a causa di questa situazione, risolvere tutti i problemi incontrati senza poter fare affidamento su alcun aiuto esterno e questo ha esacerbato ulteriormente l'approccio *trial-and-error* di cui si è parlato nella sezione 2.5.

4.2 Raggiungimento obiettivi

4.2.1 Obiettivi proponente

Riprendo quanto riportato in sezione 2.3 per trarre un bilancio sul grado di soddisfacimento dell'azienda nei confronti dei risultati ottenuti nel corso dello *stage*, che schematizzo di seguito (riportando obiettivo, se è stato raggiunto e dove trovarne la prova).

Obiettivi minimi:

Obiettivo	Raggiunto	Fonte
Studio e comprensione del linguaggio di programmazione Dart e del *framework* Flutter	SI	A
Analisi strumenti Azure: Azure Console e Azure Spatial Anchors	SI	3.1.2
Ricerca di *framework* e *software development kit* per implementare le Azure Spatial Anchors in Flutter	SI	3.1.3
Eventuale studio di linguaggi necessari alle implementazioni native, come ad esempio Kotlin, Java, Unity o Swift	SI	3.7, 3.8.

Obiettivo	Raggiunto	Fonte
Completamento del *framework* scelto nell'app Flutter esistente	SI	3.4, 3.6
Completamento dello sviluppo dei componenti per rappresentare le *anchor* nello spazio in realtà aumentata	SI	3.11
Completamento dello sviluppo delle *application programming interfaces* per l'interazione utente con gli ancoraggi in realtà aumentata	SI	3.1, 3.2, 3.3

Tabella 4.1: Tabella dei requisiti funzionali

Obiettivi massimi:

Obiettivo	Raggiunto	Fonte
Sviluppo di componenti per mappare *asset* aziendali ad *anchor*	SI	3.11
Sviluppo di componenti grafici per la visualizzazione di metriche e informazioni di controllo	NO	

Tabella 4.2: Tabella dei requisiti funzionali

Complessivamente ritengo di aver soddisfatto gli obiettivi posti dal proponente con un buon grado di completezza.

4.2.2 Obiettivi personali

Riprendendo quanto detto in sezione 2.6 traggo un bilancio sul mio grado di soddisfacimento personale rispetto allo *stage* svolto.

- Acquisire competenze di programmazione *frontend*:
  - Utilizzare linguaggi specifici (ad esempio Dart);
  - Sviluppare componenti grafiche per applicazioni *mobile*;
  - Sviluppare familiarità con strumenti comunemente usati (ad esempio Visual Studio Code);

Questo primo obiettivo l'ho soddisfatto quasi nella sua interezza: ho avuto modo di sviluppare con Dart e ho usato estensivamente Visual Studio Code e Android Studio, tuttavia il lavoro nel suo complesso si è incentrato sull'infrastruttura sottostante piuttosto che sulla creazione di molteplici elementi grafici e quindi la sensazione di aver "creato un'applicazione", o meglio di aver lavorato sul *fronted* piuttosto che sul *backend*, è stata meno forte del previsto.

- Osservare direttamente il lavoro in azienda *software*:
  - Vedere organizzazione giornaliera lavori;

  - Valutare tempistiche e ritmi lavorativi;
  - Valutare equilibrio vita-lavoro;

Il secondo obiettivo l'ho completamente soddisfatto: ho potuto vedere i ritmi lavorativi all'interno dell'ufficio e farmi un'idea molto più chiara di cosa aspettarmi una volta uscito dal cammino universitario.
Inoltre, complice un'organizzazione settimanale che ha compreso tre giorni in presenza e tre giorni di lavoro a distanza, ho avuto un'impressione nel complesso positiva di quello che è l'equilibrio vita-lavoro in ambiente aziendale (o perlomeno in ambiente di *startup*).

- Acquisire competenze di sviluppo *mobile*:
  - Sviluppare familiarità con strumenti comunemente usati (ad esempio emulatori);
  - Sviluppare componenti applicazione *mobile*;
  - Utilizzare *framework* specifici (ad esempio Flutter);

Il terzo obiettivo l'ho completamente soddisfatto: ho avuto modo di sviluppare e provare applicazioni costruite in Flutter sia su emulatore che sul mio telefono personale, usando strumenti specifici com Android Studio o i vari *plugin* di Flutter per Visual Studio Code.

- Acquisire competenze di realtà aumentata:
  - Comprendere teoria dietro concetti come ancoraggi e tridimensionalità;
  - Conoscere tecnologie principali usate nel campo;
  - Valutare se sia di mio interesse perseguire studi futuri sull'argomento.

Il quarto obiettivo l'ho soddisfatto a un livello che ritengo soddisfacente: ho avuto modo di usare un sottoinsieme rilevante delle maggiori tecnologie per la realtà aumentata (ARCore, ARKit e Azure Spatial Anchors) e ne ho studiato la teoria sottostante. Sebbene abbia ancora qualche riserva sull'uso di queste risorse in ambito professionale, ritengo abbiano un grande potenziale.
Nel complesso ritengo questa esperienza di *stage* più che positiva e arricchente da un punto di vista sia tecnico che professionale.

4.3 Competenze

Durante questo stage ho acquisito o affinato un discreto insieme di competenze, a partire da linguaggi mai usati prima (Dart, Kotlin e Swift) per passare allo sviluppo nel *framework* Flutter fino ad arrivare all'uso del *sdk* Azure Spatial Anchors.
Ho imparato a comprendere e sfruttare codice di terze parti, preso da *repository* e quindi affinando la mia comprensione dello strumento GitHub, e a destreggiarmi in mancanza di documentazione o direzioni esplicite.
Ho sviluppato componenti grafiche e in generale componenti per applicazioni *mobile frontend* e ho acquisito una maggiore dimestichezza con degli *integrated development environment* largamente usati, ovvero Visual Studio Code, Android Studio e Xcode.
Inoltre mi sono trovato a dover gestire l'interoperabilità tra linguaggi diversi ed effettuare operazioni di *debugging* sfruttando i *logs* dell'applicazione.
Per schematizzare:

- Uso dei linguaggi:
  - Dart;
  - Kotlin;
  - Java;
  - Swift;
- Uso del *framework* Flutter;
- Utilizzo degli *integrated development environment*s:
  - Visual Studio Code;
  - Android Studio;
  - Xcode;
- Competenze di programmazione *frontend* e *mobile*:
  - Sviluppo componenti grafiche;
  - Sviluppo componenti di infrastruttura;
- Competenze teoriche su tecnologie per realtà aumentata:
  - ARCore;
  - ARKit;
  - Azure Spatial Anchors;
- Uso di codice di terzi preso da *repository* esterne;
- Gestione interoperabilità di codice scritto in linguaggi diversi;
- Lettura e comprensione dei *logs* di *debugging* per le applicazioni a *runtime*.

Passando invece alle competenze delle quali mi sono scoperto manchevole, tralasciando le ovvie lacune sulle tecnologie tratte (come Flutter o Azure Spatial Anchors), quelle di cui ho più accusato l'assenza sono di natura più pratica che teorica: infatti non ero abituato a gestire codice di terzi, preso da *repository*, leggerlo e integrarlo nel mio programma. Così come non mi era ancora capitato di gestire l'interoperabilità di linguaggi diversi per raggiungere un fine.
Queste lacune sono facilmente reinterpretabili in una critica al corso di studi, che forse lascia troppo poco spazio all'effettivo lavoro di progettazione e codifica in favore di un approccio più teorico.
Un modo per arginare il problema sarebbe potenziare le attività didattiche pratiche (comunque presenti, come gli *assignment* dei corsi di Programmazione ad Oggetti e Tecnologie Web o il progetto del corso di Ingegneria del Software) a discapito di quelle più teoriche (come Logica) o non incentrate sullo sviluppo (come Calcolo Numerico).

Appendici

Appendice A

Flutter

E' impossibile parlare di Flutter senza prima discutere del linguaggio su cui si basa: Dart.
Faremo quindi un'introduzione quanto più sintetica possibile sulle sue caratteristiche più proprietarie, tralasciando invece le similitudini a linguaggi popolari che possono facilmente essere intuite dal lettore.

A.1 Dart

Dart è un linguaggio compilato fortemente tipizzato con sintassi simile a C, e adotta molte funzionalità tipiche dei linguaggi moderni: considera infatti ogni variabile un oggetto e implementa la *null safety*, obbligando il programmatore a esplicitare con sintassi specifiche la presenza o meno del valore `null`.

```
1  //dichiarazione di variabili
2  String variabileNotNullable = '';  //non puo' mai essere null
3  String? variabileNullable = null;  //puo' anche essere null
4
5  //utilizzo esplicito di variabile nullable
6  fun (variableNullable);   //compile error
7  fun (?variableNullable);  //esplicito che puo' essere nulla
8
9  //utilizzo esplicito di variabile not nullable
10 if (value != null)
11     fun (!value);   //dico al compilatore che sono certo value != null
```

Listing A.1: Dart *null safety*

Permette inoltre l'interpolazione di stringhe:

```
1  //sfrutto il simbolo $
2  'number = $variableName';              //inserisco in stringa
3                                         //direttamente una variabile
4  'expressionResult = ${expression}';    //inserisco in stringa
5                                         //direttamente un'espressione
```

Listing A.2: Dart interpolazone stringhe

Dart tratta in maniera particolare le funzioni, che sono oggetti, hanno un tipo (`Function`), possono avere parametri posizioni o nominali, obbligatori od opzionali:

```
1  //parametri obbligatori posizionali
2  void fun (int a, int b);  //non possono essere nulli
3
4  //possono essere anteceduti da parametri posizionali opzionali
5  void fun (int a, int b, [String opz1 = '', String opz2 = '']){}
6  void fun (int a, int b, [String? opz1nullable, String? opz2nullable]){}
7
```

```
//oppure possono essere anteceduti da parametri nominali opzionali
void fun (int a, int b, {String opz1 = '', String opz2 = ''}){}
void fun (int a, int b, {String? opz1nullable, String? opz2nullable}){}

//non possono essere anteceduti da entrambi
//il seguente codice non compila:
void fun (int a, int b, [String opz1 = ''], {String opz2 = ''}){}

//tutti i costruttori in Dart hanno solo parametri nominali
//di default i parametri nominali sono opzionali
//possono pero' essere resi obbligatori con la keyword 'required'
void fun ({required String obb1,
           required String? obb2,
           String opz1 = ''}){}
```

Listing A.3: Dart parametri funzioni

Esistono le funzioni anonime (largamente sfruttate nei metodi `build` di Flutter), che possono essere ovviamente passate come argomento ad altre funzioni, essendo oggetti:

```
//prima sintassi
(){
    //corpo della funzione anonima
}

//seconda sintassi
() => return_expression

//assegnamento di funzione anonima ad una variabile
//ritorna una stringa con un messaggio portato in upper case
var upperfy = (msg) => '${msg.toUpperCase()}';

//passaggio di funzione come parametro
fun (firstValue, () => secondValue);
```

Listing A.4: Dart funzioni anonime

Abbiamo delle espressioni condizionali specifiche del linguaggio dalla grande espressività e che quindi vengono largamente usate:

```
//primo tipo di condizione, detta ternaria
condizione ? expr1 : expr2;

//equivale a (in pseudocodice):
if (condizione == true) return expr1
else                    return expr2

//vediamo un esempio
int? a;               //a==null di default
a = a==null ? 1 : 0;  //ad 'a' viene assegnato il valore 1
                      //siccome a==null


//################################
//secondo tipo di condizione peculiare:
expr1 ?? expr2;

//equivale a (in pseudocodice):
if (expr1 != null) return expr1
else               return expr2

//vediamo un esempio
int? a;      //a==null di default
```

```
24 a = a ?? 1;  //ad 'a' viene assegnato il valore 1 siccome a==null
```

Listing A.5: Dart espressioni ternarie

I caratteri `?` e `!` sono, come si evince dalle espressioni ternarie e dalla *null safety*, di cruciale importanza in Dart, ed è quindi opportuno vederne ancora qualche esempio:

```
1  list[1]     //accede al secondo elemento della lista
2
3  list.?[1]
4  //equivale a (in pseudocodice):
5  if (list[1] != null)    return list[1]
6  else                    return null
7
8  //##############################
9
10 foo?.bar
11 //equivale a (in pseudocodice):
12 if (foo.bar != null)    return foo.bar
13 else                    return null
14
15 //##############################
16
17 foo!.bar
18 //equivale a (in pseudocodice):
19 if (foo.bar != null)    return foo.bar
20 else                    throw (runtimeException)
```

Listing A.6: Dart operatori '?' e '!'

Le applicazioni *mobile* e *web* spesso si trovano a eseguire istruzioni che richiedono l'attesa di risorse esterne (ottenere dati dalla rete o leggerli da un *file*, oppure scrivere su un *database*). Per evitare il completo blocco dell'esecuzione (disastroso per un applicativo *consumer*) Dart implementa nativamente delle funzioni di programmazione asincrona, che permetto di svolgere operazioni anche mentre altre sono in attesa.
Per fare ciò si serve di tre *keyword*: `async`, `await` e `Future`:

```
1 Future<void> checkVersion () async {
2     var version = await lookUpVersion();
3 }
4
5 //un'espressione marcata con la keyword 'await' ritorna sempre Future<T>
```

Listing A.7: Dart programmazione asincrona

Trattiamo infine le caratteristiche peculiari di Dart per quanto riguarda classi e metodi:

- Esistono i costruttori costanti, marcati `const`, che sono inizializzati a *compile time*. Tutti gli altri sono implicitamente dinamici (quindi *runtime*);
- Posso creare costruttori nominati per implementare costruttori multipli tramite la sintassi:

```
1 ClassName.constructorName () :
2     firstField = firstValue,
3     secondField = secondValue;
```

- Ogni entità è una classe, ogni classe discende da `Object` (a eccezione di `Null`);
- Per ogni classe viene creata un'interfaccia implicita che contiene tutti i membri d' istanza della classe;

- I metodi *getter* e *setter* di una classe vengono creati implicitamente per variabili non `final`, ma possono essere definiti esplicitamente con le *keyword* `get` e `set`.

A.2 Widget

Mostriamo ora alcuni componenti di Flutter necessari al proseguimento della lettura, partendo dai *widget*: riprendono l'idea di React di costruire tutta l'interfaccia grafica tramite uno *stack* di *widget*, che definiscono il proprio aspetto basandosi sulla configurazione e lo stato attuali. Quando lo stato di un *widget* cambia, esso riscrive la propria descrizione (insieme di istruzioni e valori che lo determinano) e il *framework* valuta le differenze tra questa e la descrizione precedente per apportare le modifiche necessarie al suo *render tree* e quindi, ultimamente, modificarne l'aspetto e finalizzare il cambio di stato.

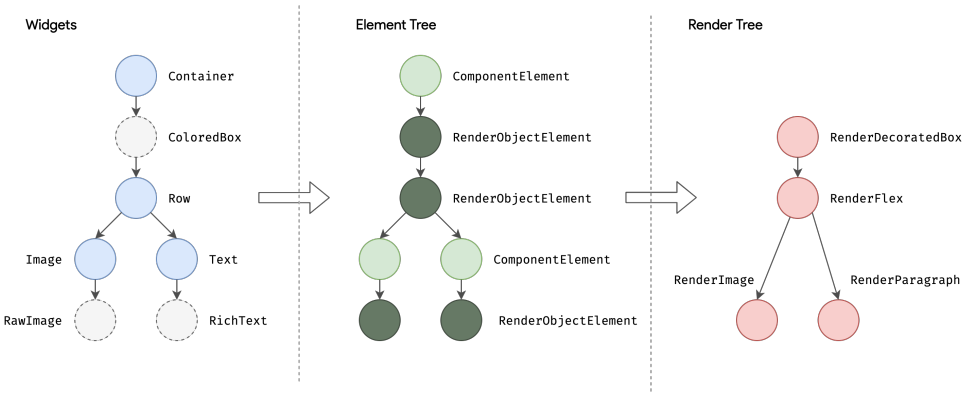


Figura A.1: Flutter usa tre alberi logici come rappresentazione delle proprie strutture: uno per i *widget*, uno per gli *element* e infine quello di *render*.[1]

L'albero di *render* è una delle tre strutture logiche che Flutter usa per descrivere i propri oggetti e le proprie interfacce grafiche: *widget tree*, *element tree* e *render tree*. Il *widget tree* rappresenta gli elementi di un *widget* come uno *stack* di *widget*, messi in relazione con il loro elemento padre e i loro elementi figli. Quello mostrato in figura A.1, ad esempio, rappresenta un `Containter` che contiene un `ColoredBox` che a sua volta contiene una `Row` e via dicendo. I *widget* sono per definizione immutabili, quindi una modifica richiede necessariamente una distruzione e successiva ricreazione del *widget* interessato.
Vediamo un esempio più specifico di *widget tree*:

```
1  Widget build(BuildContext context) {
2      return Scaffold(
3        appBar: AppBar(
4          title: Text(widget.title),
5        ),
6        body: Center(
7          child: Column(
8            mainAxisAlignment: MainAxisAlignment.center,
9            children: <Widget>[
10             const Text(
11               'You have pushed the button this many times:',
```

[1] Fonte: https://docs.flutter.dev/resources/architectural-overview

```
12              ),
13              Text(
14                '$_counter',
15                style: Theme.of(context).textTheme.headline4,
16              ),
17            ],
18          ),
19        ),
20        floatingActionButton: FloatingActionButton(
21          onPressed: _incrementCounter,
22          tooltip: 'Increment',
23          child: const Icon(Icons.add),
24        ),
25      );
26  }
```

Listing A.8: Codice rilevante per la parte grafica dell'*app* di esempio di Flutter

Il codice precedente produce il seguente risultato (e il seguente *widget tree*):

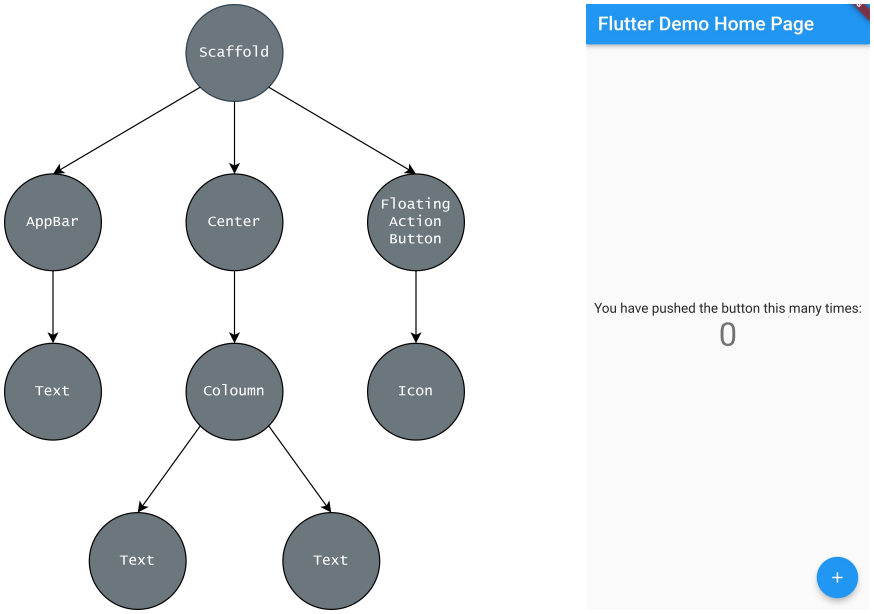


Figura A.2: Il *widget tree* dell'applicazione di esempio di Flutter e il suo risultato su Android

Per quanto riguarda le dimensioni delle varie componenti, Flutter ragiona come segue: i padri passano ai figli (dall'altro verso il basso nel *widget tree*) dei *"constraints"*, ovvero le dimensioni massime oltre le quali non possono cresce, e i figli passano quindi ai padri (dal basso verso l'alto nel *widget tree*) le proprie dimensioni.
Il *widget tree* è quello più importante da tenere a mente, e che più direttamente si interfaccia con il codice scritto dal programmatore.
Torniamo però alla figura A.1 per parlare brevemente degli altri due alberi: l'*element tree* rappresenta gli elementi che, a differenza dei *widget*, sono mutabili e vengono definiti come "un'istanza di un *widget* in una particolare posizione dell'albero". Gli elementi sono responsabili di aggiornare l'interfaccia utente e fungono da ponte tra `Widget` e `Render Object`.
Il *render tree* rappresenta i `Render Object` e viene interpellato dal *framework* per

disegnare i componenti dell'interfaccia grafica. Una singola istanza di `Render Object` contiene tutte le informazioni riguardanti dimensioni, colori e logiche di *layout* di un `Widget`.
Abbiamo accennato che i *widget* quando cambiano stato richiamano il proprio metodo `build` che provoca quindi una ricostruzione del *widget* stesso. Questi *widget* si chiamano "*stateful*", e sono in diretta contrapposizione con quelli incapaci di modificarsi durante l'esecuzione, chiamati "*stateless*".
Vediamo un esempio di *stateful widget*:

```
class MyApp extends StatelessWidget {
  //stateless widget che fa da radice per l'applicazione
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: const MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}

class MyHomePage extends StatefulWidget {
  //qui passiamo allo stateful widget
  const MyHomePage({super.key, required this.title});
  final String title;

  @override
  //e qui creiamo lo stato vero e proprio
  State<MyHomePage> createState() => _MyHomePageState();
}


class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    //quando viene chiamata questa funzione viene
    //richiamato il metodo build, di fatto cambiando lo stato
    //del widget e provocando un re-render dell'applicazione
    setState(() {
      _counter++;
    });
  }

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

Listing A.9: Creazione *stateless widget*

Si nota immediatamente che l'uso degli *stateful widget* provoca scrittura di codice ridondante e si è quindi deciso di arginare questo problema, aumentando al contempo la condivisione di codice tra i *widget*, introducendo i Flutter Hooks (ispirati dagli *hooks* di React). Vediamo quindi la riscrittura del codice mostrato nel frammento A.9 opportunamente modificato usando un `HookWidget`:

```
class MyApp extends StatelessWidget {
  //stateless widget che fa da radice per l'applicazione
  const MyApp({super.key});

  @override
```

```
  Widget build(BuildContext context) {
    return MaterialApp(
        home: const MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}

class MyHomePage extends HookWidget {
  //basta questa classe per fare tutti gli step
  //necessari al cambio di stato

  const MyHomePage({super.key, required this.title});
  final String title;

  Widget  build (BuildContext context){
    //_counter viene inizializzato a zero e reso
    //l'hook per il cambio di stato, quindi quando
    //il suo valore cambia viene provocato un rebuild
    final _counter = useState(0)

    return Scaffold(
      appBar: AppBar(),
      body: Center(),
      floatingActionButton: FloatingActionButton(
        //qui il valore di _counter cambia
        //provocando una rebuild
        onPressed: () => _counter.value++,
      )
    )
  }
}
```

Listing A.10: Creazione *hook widget*

A.3 Method channels

I *method channel* nascono per uno scopo: utilizzare codice nativo tramite Flutter, ad esempio per chiamare *application programming interfaces platform-specific* in Kotlin o Java per Android, in Swift per iOS o in C++ per Windows.
Flutter non usa queste funzionalità tramite generazione di codice e preferisce un approccio basato sullo scambio di messaggi. Vediamone quindi un esempio per ottenere, tramite Java, il valore di carica della batteria di un dispositivo Android.
Per prima cosa è necessario agire sull'Android *manifest* inserendo il nome dell'applicazione di Flutter, in questo caso `batterylevel` (l'applicazione si può creare con il comando: `flutter create nome_applicazione`).

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.batterylevel">
```

Listing A.11: Android *manifest*

A questo punto si vanno a costruire gli "agganci" per il *method channel* in codice nativo, in questo caso quindi nell'attività principale di Java:

```
public class MainActivity extends FlutterActivity {
  //"samples.flutter.dev/battery" e' il nome del method channel
  //come definito in main.dart
  private static final String CHANNEL = "samples.flutter.dev/battery";
```

```
5
6   @Override
7   public void configureFlutterEngine(@NonNull FlutterEngine
      flutterEngine) {
8     super.configureFlutterEngine(flutterEngine);
9     new MethodChannel(flutterEngine.getDartExecutor()
10                                   .getBinaryMessenger(), CHANNEL)
11            .setMethodCallHandler(
12                    (call, result) -> {
13                        //Questo e' il metodo invocato nel main
14                        //sfruttiamo "getBatteryLevel" per invocarlo
15                        if (call.method.equals("getBatteryLevel")) {
16                            //il metodo vero e proprio in Java
17                            int batteryLevel = getBatteryLevel();
18                            if (batteryLevel != -1) {
19                                result.success(batteryLevel);
20                            } else {
21                                result.error("UNAVAILABLE",
22                                "Battery level not available.", null);
23                            }
24                        } else {
25                            result.notImplemented();
26                        }
27                    }
28            );
29  }
30
31  private int getBatteryLevel() {
32    //metodo che ritorna il valore di batteria del dispositivo
33  }
```

Listing A.12: Java *main activity*

Infine ci occupiamo di lanciare il messaggio da Flutter invocando il metodo:

```
1  class MyHomePage extends HookWidget {
2  const MyHomePage({super.key});
3  //creo e definisco il method channel
4  static const platform = MethodChannel('samples.flutter.dev/battery');
5
6    @override
7    Widget build(BuildContext context) {
8      var batteryLevel = useState('Unknown battery level.');
9
10     return Material(
11       child: Center(
12         child: Column(
13           mainAxisAlignment: MainAxisAlignment.spaceEvenly,
14           children: [
15             ElevatedButton(
16               onPressed: () async {
17                 try {
18                   //qui avviene l'effettiva invocazione del
19                   // method channel che provoca una chiamata in Java
20                   final int result =
21                       await platform.invokeMethod('getBatteryLevel');
22                   batteryLevel.value = 'Battery level at $result % .';
23                 } on PlatformException catch (e) {
24                   batteryLevel.value =
25                       "Failed to get battery level: '${e.message}'.";
26                 }
27               },
28               child: const Text('Get Battery Level'),
```

```
29              ),
30              Text(batteryLevel.value),
31            ],
32          ),
33        ),
34      );
35    }
36  }
```

Listing A.13: Dart *main*