

RELAZIONE DI PROGETTO DI PERVASIVE COMPUTING

Intelligent Backpack

Componenti del gruppo:

- *Lazzari Matteo*

Indice

1	Introduzione	3
2	Analisi dei requisiti	5
2.1	Benchmarking	9
3	Progettazione	10
3.1	Domain Storytelling	10
3.2	Casi d'uso	14
3.3	Progettazione tecnologica e architetturale	16
4	Implementazione	19
4.1	Gestione dispositivi Raspberry	20
4.2	Applicazione mobile	30
4.2.1	Gestione NFC Android	32
4.2.2	Camera	35
4.2.3	Firebase Realtime database	38
4.3	Microservizi	39
4.4	Database	43
5	Testing e performance	47
5.1	Raspberry Pi 4 e IoT Hub	47
5.2	Applicazione Smartphone	49
5.2.1	Testing	49
5.3	Performance	49
5.4	Microservizi	49
5.4.1	Testing	49
5.4.2	Performance	50
6	Analisi di deployment su larga scala	52
6.1	Raspberry Pi 4	52
6.2	Applicazione smartphone	52
6.3	Microservizi	53
7	Timeline di lavoro	54

1 Introduzione

Lo scopo finale del progetto è lo sviluppo di un sistema per la gestione di uno zaino smart, ovvero intelligente, e la sua interazione con il calendario scolastico, in modo da fornire agli studenti uno strumento per poter gestire gli oggetti e/o libri che devono essere inseriti nello zaino per le lezioni del giorno successivo e sapere sempre cosa si è inserito all'interno. Questo prodotto è considerabile come dispositivo Wearable IoT, in quanto unisce un oggetto indossabile con intelligenza, potenza di calcolo e connettività, oltre all'interazione rivista in ottica tecnologica. Il problema alla base del progetto è quello di ottimizzare la gestione del materiale scolastico da portare, in quanto ci possono essere errori di comunicazione tra professore/studenti, oltre alla possibilità che uno studente potrebbe dimenticarsi del materiale a casa.

Il sistema sviluppato sarà composto da 3 principali componenti per l'invio dei dati:

- *Servizi in cloud*: Tutta la parte che riguarda il server e il database sarà gestita in cloud, tramite l'utilizzo della piattaforma Microsoft Azure Portal;
- *Applicazione smartphone*: L'applicazione per dispositivi mobili offre all'utente un'interfaccia che permette di ottenere tutte le informazioni (a richiesta) del calendario, delle successive lezioni e degli oggetti/libri che sono stati inseriti all'interno dello zaino;
- *Raspberry PI 4*: Componente hardware general purpose del progetto che viene utilizzato per la lettura del codice RFID di un libro/oggetto inserito nello zaino dall'utente, aggiornare in tempo reale il database remoto con gli oggetti inseriti/rimossi e rimanere in ascolto di eventi inviati da Hub-Iot di Azure Portal.

Per quanto riguarda la parte dei servizi gestiti in cloud, si è deciso di optare per lo sviluppo di un'architettura basata su microservizi, in quanto dispone di una miglior scalabilità dei singoli servizi, se necessario (miglior gestione del carico di lavoro).

Il database è stato hostato su un server usando un database relazionale, in particolare, usando un server SQL visto che è perfetto per la gestione di tutte le informazioni che sono collegate tra di loro.

Per quanto riguarda l'ambito tecnologico, il Raspberry utilizzato è stato collegato ad un lettore RFID, che viene utilizzato per la lettura del tag del libro e un modulo WiFi, che permette la comunicazione del componente hardware con il database remoto per informarlo sull'aggiunta/rimozione di libri (la comunicazione verrà poi mostrata successivamente nella sezione dedicata alla Progettazione e Implementazione).

2 Analisi dei requisiti

Prima di partire con lo sviluppo del progetto nelle sue diverse parti, tramite l'utilizzo di interviste e user stories, sono stati dapprima individuati i principali requisiti che devono essere rispettati per avere un'applicazione conforme con le richieste dell'utente. I principali requisiti e funzionalità richieste sono state poi discusse all'interno del team di sviluppo e progettazione per individuare quali di esse fossero importanti ed essenziali da aggiungere e quali potessero essere di minor interesse, dando una priorità ai macrotask individuati.

Alla fine della prima fase di *brainstorming*, sono stati individuati i principali requisiti che devono essere tenuti in considerazione e poi suddivisi nelle seguenti categorie:

- *Requisiti funzionali*: si riferiscono alle specifiche funzionali del prodotto che deve essere sviluppato. Questi indicano cosa deve esser fatto dal prodotto e quali funzionalità deve fornire. In particolare, in questa sezione troviamo le seguenti funzionalità:
 - Il sistema deve permettere all'utente di registrare nuovi RFID associati a determinati libri/oggetti;
 - Il sistema deve permettere di aggiornare correttamente i libri/oggetti aggiunti/rimossi dallo zaino da parte dell'utente;
 - Gestione dei calendari: dovrà esser possibile creare un calendario per un determinato anno scolastico, istituto e classe
 - Il sistema deve permettere agli insegnanti di gestire ogni aspetto delle lezioni: cambiamenti di orario di una lezione, cambiamenti di libri/oggetti da portare in una determinata lezione, creazione e rimozione di lezioni;
 - Gestione dei libri: memorizzazione dei vari codici ISBN dei libri e delle loro copie
 - Il sistema deve permettere agli utenti di visualizzare, tramite applicazione mobile, tutte le informazioni che riguardano lo zaino (libri/oggetti attualmente inseriti), materiale mancante da inserire e i dati relativi alle lezioni/calendario.

- *Requisiti di performance*: si riferiscono alle specifiche delle prestazioni del prodotto. Questi requisiti indicano la velocità, la capacità e la scalabilità del prodotto. In particolare, il sistema deve rispondere in modo veloce e consistente alle richieste che vengono effettuate, entro limiti di tempo che sono stati definiti all'inizio del progetto:

- Il sistema deve gestire l'inserimento di una risorsa nel database remoto in un tempo inferiore ai 5 secondi (tempo stimato di pausa tra l'inserimento di un libro e un altro);
- Il sistema deve verificare le credenziali di login dell'utente con un tempo inferiore ai 2 secondi;
- Il sistema deve processare le chiamate fatte dal professore in un tempo inferiore ai 5 secondi;
- Le restanti funzionalità del sistema non devono eccedere ad un tempo di risposta di 10 secondi.
- Si utilizzerà un database remoto realtime per poter facilitare e velocizzare l'accesso ai dati dei propri zaini e il relativo materiale inserito al suo interno

Come vedremo nelle sezioni successive, il prototipo realizzato non consente di soddisfare a pieno i requisiti di performance, in quanto non sono stati scelti piani tariffari ad alte prestazioni per ospitare i servizi, i database e servire le richieste.

- *Requisiti architetturali*: si riferiscono alla struttura del prodotto, alla sua architettura e alla modalità di distribuzione delle varie funzioni. In particolare, per il progetto sono stati evidenziati i successivi requisiti:
- Il servizio deve essere gestito in cloud, in quanto il cliente non ha la disponibilità di hostare un servizio su un proprio server personale;
- L'architettura cloud sarà composta da una moltitudine di microservizi indipendenti tra di loro e un unico database, con viste specifiche per ogni microservizio, in modo da contenere i costi per la fase prototipale

- Il servizio deve essere scalabile, garantendo che aree con maggior carico di lavoro possano essere potenziate aggiungendo nuovi server oppure aumentando la potenza di quelli disponibili (scale up / scale out);
 - Il database deve essere di tipo SQL, in quanto è in grado di garantire un link tra le tabelle mediante le chiavi esterne, aumentando la consistenza dei dati rispetto ad un database no-SQL.
 - Tutto il sistema e le varie componenti saranno progettate seguendo uno sviluppo di tipo Domain Driven, partendo dalla definizione del dominio per definire il piano di progettazione e sviluppo
 - Si è scelto un approccio di tipo modulare, per poter meglio gestire il software di tutti i componenti, la sua manutenzione e il suo futuro upgrade per possibili prossimi lavori
 - Per quanto riguarda i dispositivi Raspberry Pi, essi avranno una loro rappresentazione in cloud, seguendo un approccio simile a quello del Digital Twin
- *Requisiti di sicurezza:* questi requisiti specificano il livello di sicurezza del prodotto da raggiungere. Questi requisiti sono molto importanti poiché un prodotto non sicuro può provocare danni a persone o proprietà.
 - In particolare, nel database gli utenti possono registrarsi e la password deve essere salvata: quindi come requisito è stato richiesto che la password fosse criptata usando algoritmi ritenuti *strong* dal punto di vista della resistenza, mediante l'utilizzo di hash functions.
 - Per quanto riguarda chiavi segrete di collegamento ai servizi e alle rappresentazioni virtuali dei dispositivi fisici, è necessario adottare tecniche di hashing ed evitare che informazioni sensibili girino in rete
 - *Requisiti di usabilità:* questi requisiti si riferiscono alla facilità d'uso del prodotto. Questi requisiti specificano come l'utente utilizzerà il prodotto e se l'interfaccia utente è facilmente comprensibile. In particolare, essendo che il sistema deve essere gestito da una grande platea di persone (da ragazzi a persone adulte), è necessario che l'usabilità sia il più semplice possibile, ma allo stesso tempo efficace.

- Il lettore dei tag RFID deve essere posizionato in un punto visibile e vicino alla tasca dello zaino, per rendere naturale l’inserimento del libro e non forzare un meccanismo non comodo per le persone;
 - Comunicazione tra zaino e applicazione smartphone sfruttando i microservizi in cloud e database remoto, senza richiedere un’ulteriore collegamento diretto col dispositivo
 - La UI dell’applicazione deve essere semplice e intuitiva, con una serie di tab per poter differenziare le varie funzionalità messe a disposizione dall’applicazione.
- *Requisiti di manutenzione:* questi requisiti indicano la necessità di manutenzione futura del prodotto. Questi requisiti sono importanti per garantire che il prodotto continui a funzionare come previsto nel tempo. In particolare, sono stati richiesti:
 - Buona leggibilità del codice, aggiungendo commenti dove necessario, per facilitare la comprensione dei meccanismi implementati;
 - Modularità nel codice, per permettere la manutenzione delle singole componenti, funzioni e servizi nel modo più indipendente e agevole possibile.
 - Documentazione obbligatoria per ogni componente del sistema
 - Build Automation sviluppata ad hoc per poter gestire il versioning, il deploy, test automatici su più sistemi e update automatici delle dipendenze
 - Continuous Integration e Continuous Deploy per velocizzare e rendere più efficace il processi di sviluppo Dev/Ops
 - *Requisiti hardware:* questi requisiti indicano una soglia minima di potenza di calcolo per poter offrire una gestione performante e allo stesso tempo non troppo invasiva delle funzionalità richieste all’interno dello zaino. In particolare, sono stati richiesti:
 - Dimensioni del dispositivo da integrare all’interno dello zaino non eccessiva, con peso ridotto per non influire negativamente sul fisico dell’utente

- Batteria non ingombrante, con possibilità di ricarica (e.g. powerbank)
- Utilizzo della sola componentistica aggiuntiva necessaria, cercando di sfruttare una scheda programmabile con moduli di connessione come WiFi già integrati onboard
- Potenza di calcolo per poter gestire programmi multithreaded

2.1 Benchmarking

I requisiti proposti verranno soddisfatti effettuando un confronto con le best practice utilizzate sul mercato, per poter gestire al meglio lo sviluppo del codice, la gestione delle comunicazioni e di tutto ciò che concerne le *Dev/Ops Practices*. In particolare, come vedremo nella sezione di progettazione e implementazione, sono stati adottati approcci e tecnologie massivamente utilizzate in prodotti già esistenti sul mercato, quali:

- Protocolli di comunicazione standard, come HTTP, ProtoBuf e RESTFul
- Utilizzo di GitHub come DVCS per versionare il codice e GitHub Actions per CI/CD
- Utilizzo di database relazionali e realtime in base alle specifiche esigenze
- Approccio di progettazione e sviluppo Domain Driven

3 Progettazione

Aderendo all'approccio di design e sviluppo Domain Driven, la fase di progettazione prevede una prima analisi del dominio in questione, in questo caso con il focus orientato esclusivamente su Use Case e User Story, ovvero i procedimenti di maggior interesse per questo progetto.

3.1 Domain Storytelling

Una delle prime analisi effettuate è stata la raccolta delle User Stories, che sono uno strumento utilizzato nello sviluppo per descrivere le esigenze degli utenti in modo semplice, rappresentando una descrizione di quello che gli utenti vogliono ottenere dall'utilizzo del sistema, il tutto in una sessione di riunione in cui si discutono gli aspetti fondamentali del sistema, in linguaggio informale ma preciso.

Sfruttando una serie di riunioni, abbiamo quindi avviato numerose discussioni che hanno portato alla stesura delle seguenti User Stories, che descrivono ad alto livello delle azioni eseguite sul sistema, in modo informale, schematico e seguendo una logica di storytelling. Innanzitutto ci siamo focalizzati sulla questione della creazione di un utente, quali azioni devono essere intraprese e quali entità partecipano.

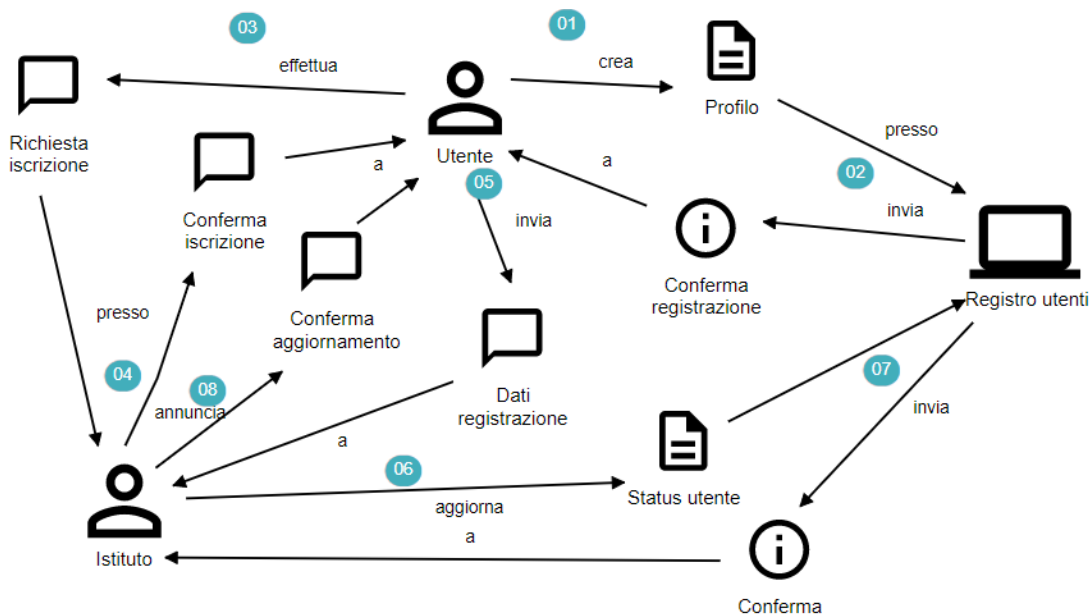


Figura 1: Creazione di un utente

Per realizzare questi schemi è stato utilizzato il tool online Egon, il quale offre un modo semplice e molto autoesplicativo per rappresentare lo storytelling a livello di dominio. Come possiamo vedere nell'immagine 1, sono rappresentate più specifiche:

- Un'utente crea un profilo presso il registro utenti, che conferma la registrazione all'utente
- Un utente effettua una richiesta di iscrizione presso un istituto, il quale annuncia la conferma di iscrizione all'utente
- L'utente invia i dati di registrazione all'istituto, il quale aggiorna lo status utente presso il registro utenti, che infine invia la conferma all'istituto e a sua volta conferma l'aggiornamento all'utente

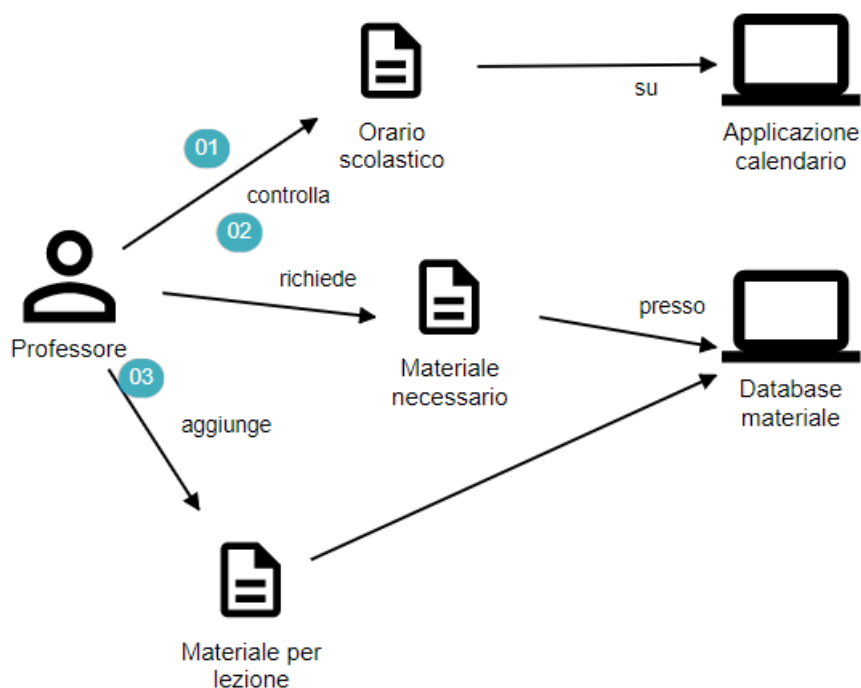


Figura 2: Creazione di un utente

Come possiamo vedere invece la user story in figura 2:

-
- ```
graph TD
 Utente((02 Utente)) -- "01 controllo" --> OrarioScolastico[Orario scolastico]
 OrarioScolastico -- "su" --> ApplicazioneCalendario[Applicazione calendario]
 ApplicazioneCalendario -- "su" --> OrarioScolastico2[Orario scolastico]
 OrarioScolastico2 -- "05 Controlla" --> SistemaControlloMateriale[06 Sistema controllo materiale]
 SistemaControlloMateriale -- "07 informa" --> ElementiMancanti((07 Elementi mancanti))
 ElementiMancanti -- "a" --> Utente
 Utente -- "04 controllo" --> Materiale[04 materiale]
 Materiale -- "aggiungi" --> Utente
 Utente -- "08 aggiungi" --> MaterialeMancante[materiale mancante]
 MaterialeMancante -- "dentro" --> Zaino[Zaino]
 Materiale -- "dentro" --> Zaino
 Zaino -- "03 notifica" --> NuovoMateriale[nuovo materiale]
 NuovoMateriale -- "dentro" --> DatabaseContenutoZaino[Database contenuto zaino]
 DatabaseContenutoZaino -- "dentro" --> MaterialeInserito[materiale inserito]
 MaterialeInserito -- "06 controlla" --> SistemaControlloMateriale
 MaterialeInserito -- "05 Controlla" --> OrarioScolastico2
 DatabaseContenutoZaino -- "inserito dentro" --> Materiale
 Materiale -- "aggiungi" --> Utente
```

Analizzando invece la figura 3, apprendiamo l'importanza di altre specifiche ad alto livello:

- 12

- Il sistema di controllo dell'applicazione controlla l'orario scolastico e il materiale inserito all'interno dello zaino, informando eventualmente l'utente degli elementi mancanti

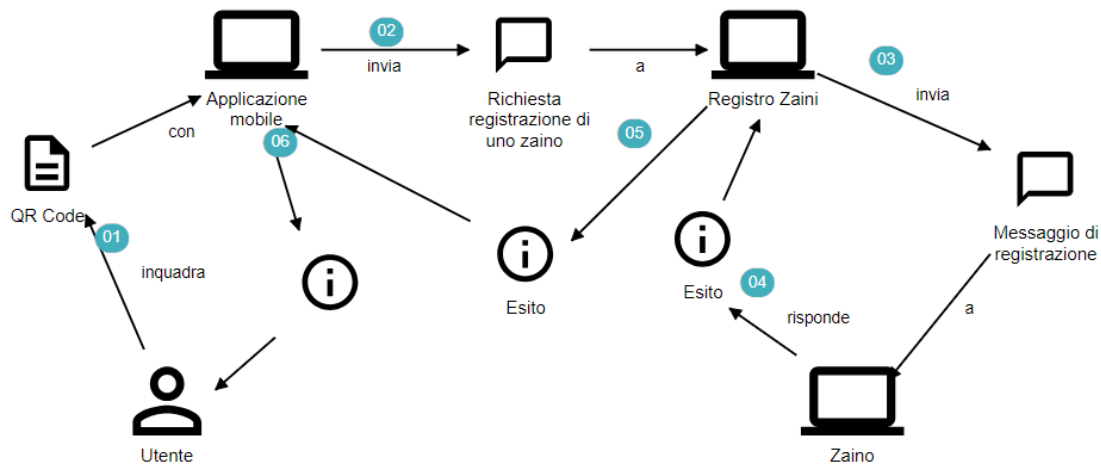


Figura 4: Registrazione di uno zaino

Infine, l'immagine 4 mostra la User Story per quanto riguarda il procedimento di registrazione di uno zaino:

1. Prima di tutto, l'utente inquadra, con l'app mobile, il QR Code presente all'interno dello zaino, accoppiato al dispositivo
2. L'applicazione mobile invia a questo punto una richiesta di registrazione al Registro Zaini, specificando il codice letto per riferirsi al preciso dispositivo in possesso
3. Il registro degli zaini, ricevuta la richiesta, invierà un messaggio di registrazione allo zaino (non sono mostrati casi particolari in cui lo zaino è già registrato o non esistente)
4. Infine, una serie di feedback in cascata riporta l'esito della registrazione all'utente, visibile con una notifica sull'applicazione smartphone

## 3.2 Casi d'uso

Per lo sviluppo degli use cases (casi d'uso) è stato utilizzato il tool online *Lucidchart* per raggruppare e schematizzare quali sono le azioni dei singoli utenti e quali attori devono interagire durante l'esecuzione di determinate azioni. L'importanza del loro utilizzo sta nell'identificare (a caratteri generali) quali attori sono coinvolti nelle varie operazioni e quali sono le principali azioni che devono essere sviluppate e/o sulle quali avere maggior riguardo.

Iniziamo con l'analizzare i casi d'uso del sistema che si vuole progettare. Gli attori che dovranno interagire con il sistema sono da ricercare tra i seguenti:

- User: colui che possiede uno zaino intelligente (Smart Backpack) e che effettua le operazioni che ne cambiano lo stato, quali aggiunta e rimozione di un oggetto
- Intelligent Backpack: questo attore rappresenta lo zaino, la sua componente smart, la quale è aperto a interazioni con l'utente esterno e possiede capacità di connessione per comunicare con l'esterno
- Istituto: entità che rappresenta un istituto scolastico, il quale detiene la facoltà di eseguire operazioni quali la possibilità di aggiungere/rimuovere e modificare il ruolo dei singoli utenti
- Vendor: entità che può aggiungere libri con il suo codice ISBN e informazioni standard come titolo
- Professore: entità che può gestire più classi, calendari, materie
- Studente: oltre ad utilizzare lo zaino, può accedere alle materie, calendario e libri necessari per una specifica lezione
- Applicazione mobile: parte del sistema che si interfaccia con l'utente, in particolare colui che ricorda l'utente riguardo qualche eventuale oggetto non inserito nello zaino ma previsto per il giorno successivo

Uno dei principali requisiti che sono scaturiti dagli use cases è la necessità di avere un'utente generale non assegnato a nessun ruolo. Infatti, tutte le azioni del professore e dello studente, possono essere generalizzate e messe sotto permessi

speciali, tramite l'utilizzo di un'utente generico che le accomuna. Saranno poi gli istituti, mediante i loro permessi, a modificare i ruoli degli utenti, assegnando studente o professore a seconda di quello che l'utente deve rappresentare.

In particolare, sono stati individuati 5 tipi di utenti diversi con diversi gradi di importanza all'interno del sistema:

1. *User*: Rappresenta l'utente generico, non specializzato e che non appartiene a nessun istituto, rappresenta il ruolo minimo all'interno del sistema;
2. *Studente*: Rappresenta un utente che studia presso un istituto;
3. *Professore*: Rappresenta un utente che lavora come professore presso un istituto. Seguendo i requisiti del problema, non è stato definito alcun vincolo riguardante il professore che può partecipare solamente ad un istituto;
4. *Istituto*: Rappresentazione di una entità in grado di gestire tutti i ruoli e azioni di alto livello sugli utenti che sono registrati al sistema;
5. *Owner*: Entità che rappresenta il proprietario del sistema o tutte le persone che ci possono accedere con i permessi di admin.



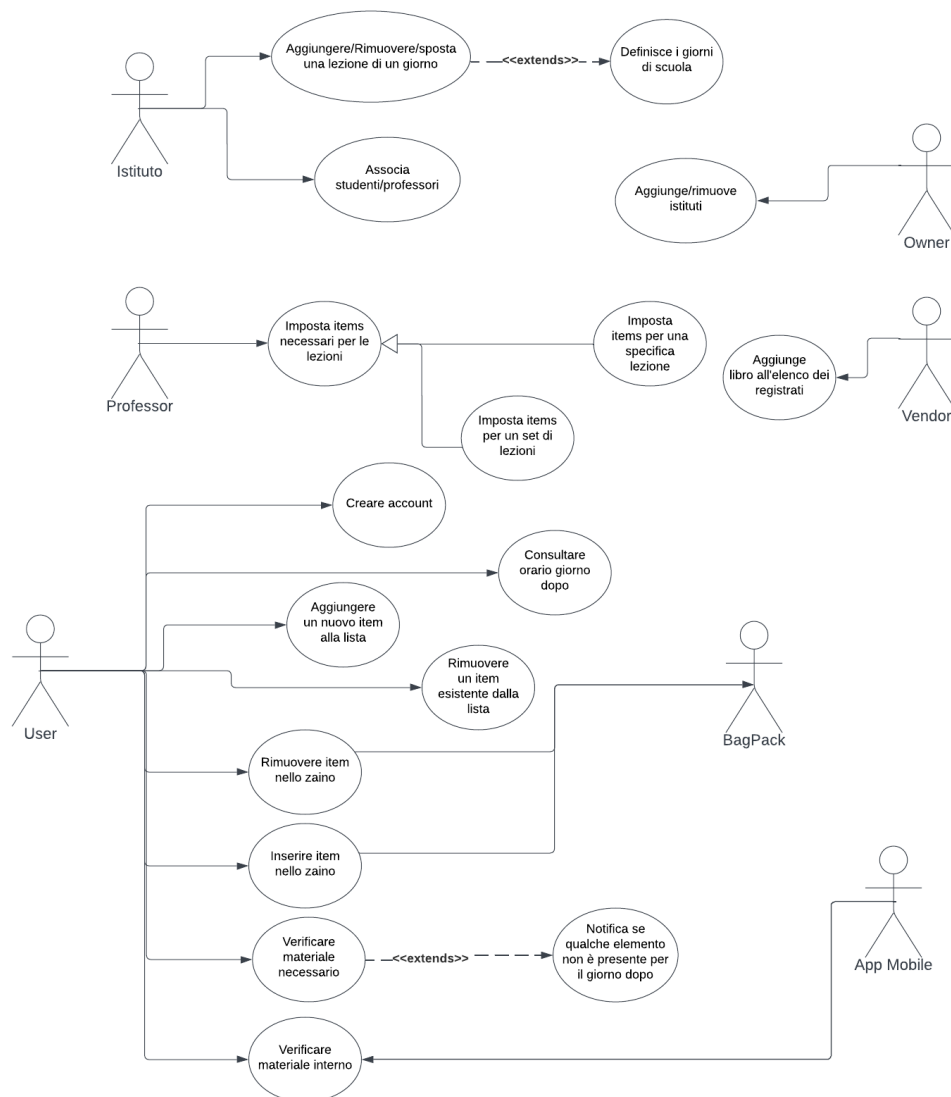


Figura 5: Esempio di un use case

### 3.3 Progettazione tecnologica e architetturale

Una volta terminata l'analisi preliminare di tutte le funzionalità del sistema, si è passati ad individuare le possibili soluzioni tecnologiche e architetture da adottare successivamente per lo sviluppo del sistema. Si sono pensate due possibili soluzioni:

1. Utilizzare un'applicazione mobile e un sistema cloud per gestire tutte le in-

terazioni. In particolare, l'utente avrebbe dovuto scansionare i libri e poi inserirli nello zaino. Questo però avrebbe provocato un'azione non familiare all'utente, con il rischio che quest'ultimo non esegua la procedura corretta e andando ad invalidare il sistema;

2. Utilizzare un modulo integrato all'interno dello zaino, che automaticamente legge il codice del libro. Questo poi sarà memorizzato per tenere traccia dei libri inseriti. Sarà presente anche un'applicazione, per visualizzare tutte le informazioni sulle lezioni e libri inseriti, oltre che alla presenza di un sistema in cloud per la comunicazione/gestione delle informazioni.

In particolare, la soluzione 2 è quella che è stata poi successivamente adottata in quanto non va ad aggiungere degli step intermedi (soprattutto di carattere umano) nel funzionamento dell'intero sistema. Infatti, per natura umana, se una persona è abituata a svolgere un determinato compito (in questo caso, inserire nello zaino i libri) in un determinato modo, andare ad aggiungere altri step può causare malcontento e/o errori, andando a compromettere l'utilizzabilità del sistema e anche il suo utilizzo, in quanto una persona non è propensa ad utilizzare lo zaino, se deve svolgere step non essenziali.

Successivamente, si è discusso su quali tecnologie utilizzare per la gestione dei vari componenti. Per la gestione del modulo integrato all'interno dello zaino si è deciso di utilizzare il componente hardware Raspberry Pi 4, in quanto già a disposizione per lo sviluppo e la sua capacità di potenza di calcolo (oltre all'addattabilità grazie ai suoi numerosi moduli esterni). Per la gestione del componente smartphone, si è deciso di sviluppare un'applicazione, seguendo i requisiti descritti precedentemente. Per la gestione dei sistemi in cloud, si è discusso su quale piattaforma utilizzare e sul tipo di sistema si volesse sviluppare. Al termine dell'analisi delle piattaforme, si è deciso di utilizzare Microsoft Azure, in quanto è facilmente collegabile al servizio Microsoft Azure Pipeline per la CI/CD del progetto (oltre che a GitHub Actions), e oltre al fatto che la stessa piattaforma offre anche servizi per il database online, andando anche a risolvere il problema di dove memorizzare i dati.

Oltre alla discussione sulle tecnologie da utilizzare, si è discusso su quale tipologia architetturale applicare per lo sviluppo del server. Le possibili opzioni discusse sono:

1. Utilizzo di un server unico (monolite), che esegue tutte le richieste dei client;
2. Utilizzo di un sistema di più server, ognuno con la propria porzione del database e funzionalità, che comunicano tra di loro per eseguire operazioni complesse (microservizi).

In particolare, secondo i requisiti, il sistema deve essere scalabile e di facile manutenzione futura, perciò la scelta ottima risulta nell'utilizzo di un sistema basato su microservizi.

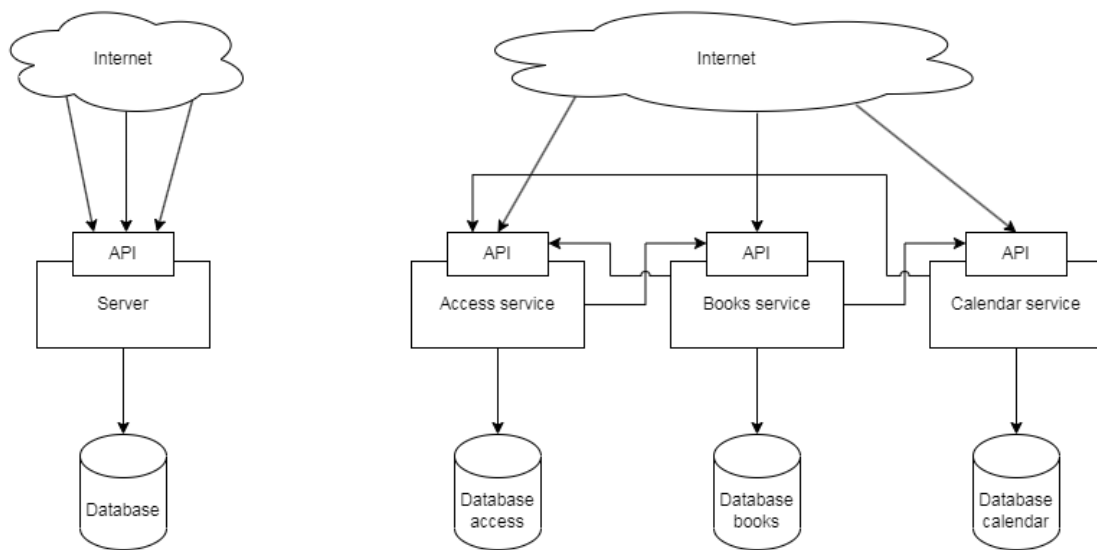


Figura 6: Differenza tra la struttura con solo 1 server (a sinistra) e a microservizi (a destra)

Per quanto riguarda il database, come vedremo nel seguito, si è preferito utilizzare un unico database in cloud, condiviso da tutti i microservizi, invece di avere un database per ogni microservizio, puramente per ragioni di costi.

## 4 Implementazione

Uno dei problemi principali riscontrati fin dall'inizio è stato come permettere la comunicazione tra i vari componenti, in quanto ognuno di essi è stato sviluppato con un linguaggio di programmazione diverso (successivamente nella sezione saranno spiegati in modo dettagliato i vari aspetti). Si è pensato di utilizzare diverse soluzioni, quali l'utilizzo di file JSON (o XML), ma alla fine si è optato per utilizzare protobuf (Protocol Buffers) di Google.

Protobuf è un formato di dati multiplatforma open source che viene utilizzato per serializzare i dati strutturati. È utile nello sviluppo di programmi che devono comunicare tra di loro sulla rete, in quanto standardizza il metodo di comunicazione tra le varie parti. I dati di protobuf (strutture dati che prendono il nome di message) sono descritti in un file di definizione *.proto* e poi compilati tramite il comando *protoc* che andrà a generare un codice che può essere utilizzato dallo sviluppatore.

Grazie all'utilizzo di protobuf, si è potuta avere una comunicazione strutturata e ben definita per ogni funzionalità richiesta. L'utilizzo di file JSON (o XML) avrebbe portato solamente ad un incremento della complessità, nonché alla necessità di avere una documentazione più difficile da comprendere in quanto devono essere specificati tutti i file JSON/XML e verificare, alla ricezione/invio dei messaggi, che il file JSON/XML sia ben strutturato e conforme con le specifiche richieste. Questo problema non sorge grazie a protobuf, in quanto è solamente necessario controllare sulla documentazione quale messaggio è richiesto.

Qui sotto sono riportati alcuni esempi di message proto che sono stati utilizzati per la comunicazione con il server.

```
message Lesson {
 string Nome_lezione = 2;
 int32 Materia = 3;
 string Professore = 4;
 string Ora_inizio = 5;
 string Ora_fine = 6;
 string Data_Inizio = 7;
 string Data_Fine = 8;
 string Giorno = 9;
 int32 ID_Calendarario = 10;
}
```

```
message UserInformations {
 string email_user = 1;
 repeated string classes = 2;
 repeated string subjects = 3;
 repeated string insitutes = 4;
}
```

```
message User {
 string email = 1;
 string password = 2;
 string nome = 3;
 string cognome = 4;
 Istituto istituto = 5;
 Role role = 6;
 string classe = 7;
}
```

Sotto saranno discussi nello specifico le implementazioni per le relative componenti principali, le difficoltà riscontrate e le soluzioni adottate.

## 4.1 Gestione dispositivi Raspberry

Per quanto riguarda la realizzazione dello zaino intelligente è stato utilizzato, come anticipato, un dispositivo Raspberry Pi 4 per la versione prototipale, in quanto offre un buon *trade off* tra costo, dimensione e performance. Per quanto riguarda il linguaggio scelto per produrre il firmware da eseguire al suo interno, è stato scelto Python, data la vasta pletora di librerie esistenti per gestire al meglio le funzionalità offerte dal Raspberry, come ad esempio il modulo RFID. Per quest'ultimo, invece, è stato utilizzato nel prototipo un comune modulo RC-522, interfacciabile sia con Raspberry che Arduino. Di seguito sono riportate le macro funzionalità che il Raspberry deve poter gestire in modo da soddisfare i requisiti:

- Connessione a Internet: deve poter ricevere e inviare dati da e verso la rete, tenendo conto che la connessione non sempre potrebbe essere disponibile
- Lettura di tag RFID: deve poter gestire il modulo RFID e leggere gli opportuni tag quando avvicinati al sensore

- Database interno: deve poter gestire un database interno nel quale memorizzare informazioni chiave come i dati dell'utente possessore del dispositivo e i libri inseriti, da sincronizzare *on the fly* con il database remoto realtime
- Connessione ad Azure IoT Hub per poter ricevere messaggi di registrazione e/o configurazione inviati dal cloud

Per produrre un firmware il più modulare possibile, l'architettura sfrutta moduli thread che possono arbitrariamente esser messi in esecuzione, richiedendo una certa interazione per poter eseguire correttamente. Per quanto riguarda la progettazione, il codice è stato organizzato in modo tale da rispecchiare il dominio analizzato e seguendo un approccio Domain Driven, quindi sfruttando entità, value objects, servizi di dominio (business logic) e infrastrutturali (database, moduli hardware).

### **Collegamento hardware modulo RFID**

Il modulo RFID utilizzato è un **RFID-RC522 Funduino** con 8 pin, compatibile con qualsiasi Raspberry Pi P1 Header. Lo schema di collegamento è mostrato nella successiva immagine.

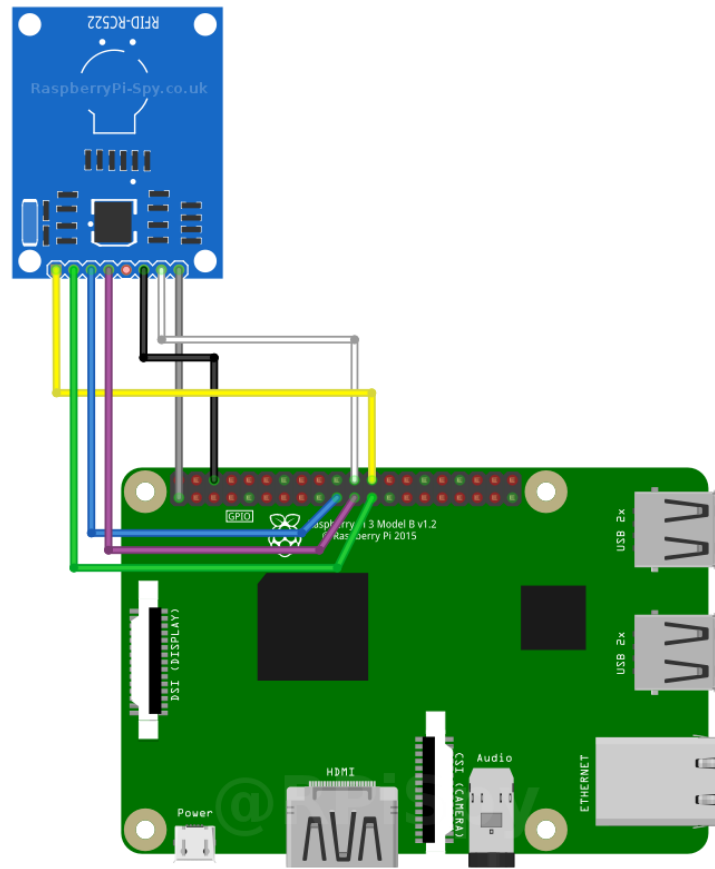


Figura 7: Collegamento pin su Raspberry Pi 4

Più precisamente, nella seguente tabella sono riportati i vari pin del modulo e il loro collegamento corretto sull'header del Raspberry.

| Pin RC522 | Pi Header    |
|-----------|--------------|
| 3.3V      | 1            |
| RST       | 22           |
| GND       | 6            |
| IRQ       | Non connesso |
| MISO      | 21           |
| MOSI      | 19           |
| SCK       | 23           |
| NSS o SDA | 24           |

La libreria per Python utilizzata, ovvero MFRC522, è compatibile con tag RFID di tipo Mifare Classic, i quali altro non sono che card contactless, mentre non opera correttamente con tag più moderni come ad esempio gli NTAG (test effettuato con un NTAG216), ovvero tag NFC più avanzati con maggior memoria disponibile e UID di 7 byte. Più precisamente, sono stati utilizzati tag Mifare Classic 1K, con 1 kB di memoria (16 settori di 4 blocchi), con un piccolo microchip e una piccola antenna, alimentato dal campo magnetico generato dal ricevitore che permette quindi di inviare i segnali, con una frequenza di 13,56 MHz. Tutto ciò è stato standardizzato con le specifiche ISO/IEC 14443, che definisce uno standard per le smart card contactless. I tag RFID portano con sé un UID e un payload testuale: nel progetto verrà usato l'UID univoco per identificare il tag stesso, e quindi l'oggetto ad esso associato.

## Integrazione moduli di rete e RFID

Innanzitutto analizziamo com'è stato definito il modulo di rete. Si è cercato di mantenere il codice il più semplice possibile seguendo il principio KISS, pur cercando di soddisfare i requisiti imposti dal progetto. Innanzitutto, come detto, questo modulo si occupa di inviare richieste HTTP in rete verso i microservizi, di fatto rappresenta il punto d'accesso alla rete.

```
def run(self):
 print("Thread " + self.name + " avviato")
 while (True):
 request = self.queue.get()
```



```

element = json.loads(json.dumps(request))
execute_calls(element["type"], element["url"], element["
 payload"])

```

Nella porzione di codice sopra rappresentato è presente la funzione eseguita dal thread di gestione della rete quando avviato: in un loop senza condizione di uscita troviamo una chiamata che mette in attesa il thread di ricevere un messaggio nella coda sincronizzata, strumento tramite il quale qualsiasi componente può inviare una richiesta di invio di un messaggio HTTP, sfruttando l'apposita coda per aggiungere richieste da servire. Una volta ricevuta una richiesta da inviare, vengono estrapolati il tipo di richiesta, l'url e il payload da inviare e viene inviata la richiesta HTTP apposita, come possiamo vedere nella porzione di codice seguente (funzione *executecalls*).

```

terminated = False
while not (terminated):
 try:
 headers = {'content-type': 'application/json'}
 if type == "PATCH":
 requests.patch(url, json.dumps(payload), headers=
 headers)
 elif type == "DELETE":
 requests.delete(url, headers=headers)
 terminated = True
 time.sleep(5)
 except Exception as e:
 print("Error")
 print(e)
 time.sleep(10)

```

Qui vediamo un altro ciclo, il quale tenta di eseguire, a cadenza regolare, la richiesta da servire, in modo da mantenere la richiesta in stato di *pending* quando la connessione alla rete non è disponibile.

Per quanto riguarda invece il modulo RFID, viene utilizzata un'altra coda sincronizzata, nella quale vengono inserite le richieste di tag RFID letti e da processare. Come possiamo vedere nella porzione di codice seguente, viene letto l'id del tag in decimale, per poi esser convertito in esadecimale e successivamente in stringa, per poter essere infine gestito a livello di dominio:

```

while True:
 print("Hold a tag near the reader")
 id, text = self.reader.read()
 print("ID: %s\nText: %s" % (id, text))
 hex_value = str(hex(id))[2:10].upper()
 value_to_send = ":".join([hex_value[i:i+2] for i in range(0,
 len(hex_value), 2)])

 message_to_send = {
 "type": "TAG_READ",
 "payload": value_to_send
 }
 self.queue_messages.put(message_to_send)
 sleep(5)

```

L'oggetto *reader* è un'istanza di tipo *SimpleMFRC522*, presente nella libreria già citata. Per l'integrazione dei due Thread, vengono sfruttate le relative code per ricevere e inviare messaggi all'entità controller: tutte le code sincronizzate sono ottenute mediante *dependency injection*. Il seguente codice mostra il controller e l'utilizzo del pattern Service Locator per semplificare l'accesso alle dipendenze; inoltre rimane in ascolto di messaggi ricevuti da tutti i moduli thread messi in esecuzione, agendo di conseguenza in base al tipo di richiesta inviata, mediando tra i vari moduli e la *business logic*, la quale si occupa di gestire il dominio globale dello zaino, compreso repository (sia remoto sia in locale).

```

if __name__ == "__main__":

 serviceLocator = ServiceLocator()
 queue_messages = serviceLocator.get_messages_queue()
 domainLogic = BackpackLogicService(serviceLocator.repository)
 domainLogic.register(serviceLocator.get_username())

 try:

 for module in serviceLocator.get_modules():
 module.start()

 while True:
 request = queue_messages.get()
 if request == "EXIT":

```

```

 raise KeyboardInterrupt()
 elif request["type"] is not None:
 if request["type"] == "REGISTER":
 print("REGISTRATO")
 name = request["payload"]["email"]
 write_username(CONFIG_FILE_PATH, name)
 domainLogic.register(name)
 if request["type"] == "UNREGISTER":
 write_username(CONFIG_FILE_PATH, "")
 domainLogic.unregister()
 if request["type"] == "TAG_READ":
 domainLogic.manage_element("", request["payload"])

```

Per quanto riguarda il *domain core* del software, come detto, si è seguito un approccio DDD, individuando quindi tutti i servizi di dominio, value objects, entità e policy di interesse.

## Modulo per Firebase Realtime Database

Per il database remoto si è scelto di utilizzare Firebase Realtime Database, ovvero un database NoSQL accessibile mediante diverse tecnologie: nel nostro caso l'applicazione mobile sfrutterà il framework di Android e il Raspberry le API Rest. Nel seguente frammento di codice (contenuto nella classe **RemoteRepository**) è riportata la funzione di aggiunta di un elemento all'interno del database, per conto di uno specifico utente e all'interno di uno specifico zaino, identificato da un *hash code*.

```

def add_element(self, user, value):
 if user == "":
 return
 user = user.replace(".", "-")
 new_request = {
 "type": "PATCH",
 "url": self.service_url + "/" + user + "/" + self.hash + ".json",
 "payload": {
 str(value): "true"
 }
 }

```

```
self.request_queue.put(new_request)
```

In particolare, è possibile notare la creazione dell'oggetto di richiesta HTTP e l'utilizzo della coda precedentemente accennata all'interno del modulo di rete: essa è infatti la coda di richieste che il thread di rete deve inviare.

## Azure Iot Hub

Per quanto riguarda la parte cloud, si è deciso di utilizzare la piattaforma Azure Iot Hub, la quale offre un backend che permette di gestire una rappresentazione cloud dei dispositivi IoT fisici, fornendo quindi uno strumento semplice e potente per gestire e mandare messaggi dal cloud verso i dispositivi e viceversa e creare una rappresentazione virtuale in cloud degli zaini. L'approccio è chiamato Device Twin, molto simile al modello Digital Twin ma più orientato a mantenere solo lo stato e la configurazione del dispositivo, non una rappresentazione digitale completa, che prevede ad esempio anche il punto di vista funzionale.

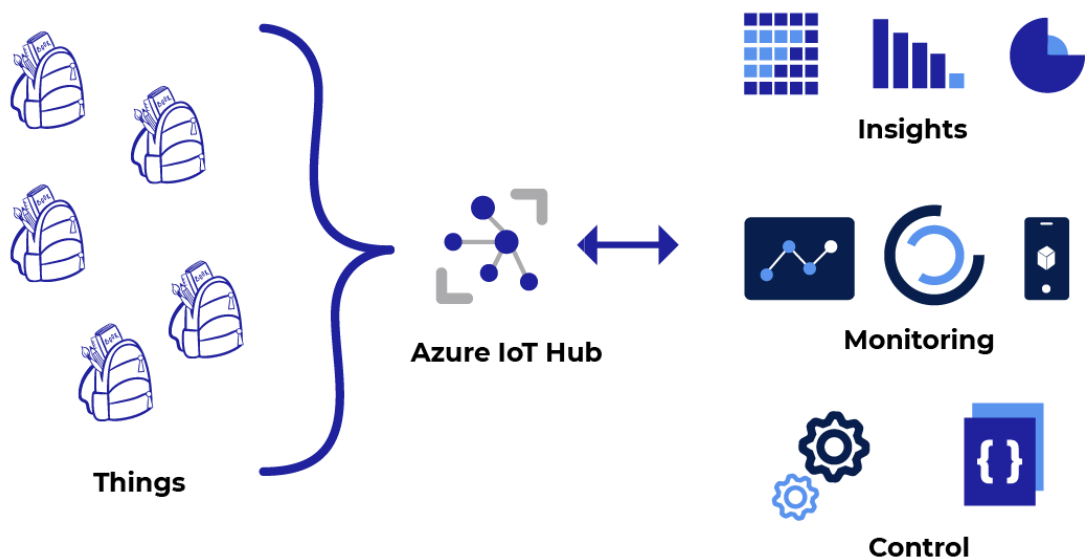


Figura 8: Potenzialità di Azure IoT Hub

Per poter registrare e gestire i vari dispositivi degli zaini è stato sviluppato un microservizio cloud, il quale opera da ponte tra l'applicazione mobile ed Azure Iot Hub, fornendo le API per una serie di funzionalità:

- Registrazione di un dispositivo: tramite l'applicazione, l'utente può inquadrare il QR Code dello zaino non registrato e registrarlo mediante una semplice chiamata HTTP Rest
- Aggiunta di un nuovo dispositivo virtuale in Azure Iot Hub
- Richiesta informazioni sui dispositivi esistenti
- Dissociazione di un dispositivo dal proprio utente

In particolare il servizio si interfaccia al database in cloud per ottenere informazioni sui dispositivi attualmente esistenti e registrarli, e alla piattaforma Azure Iot Hub per poter inviare messaggi ai dispositivi fisici, quali registrazione (specificando la mail) e dissociazione.

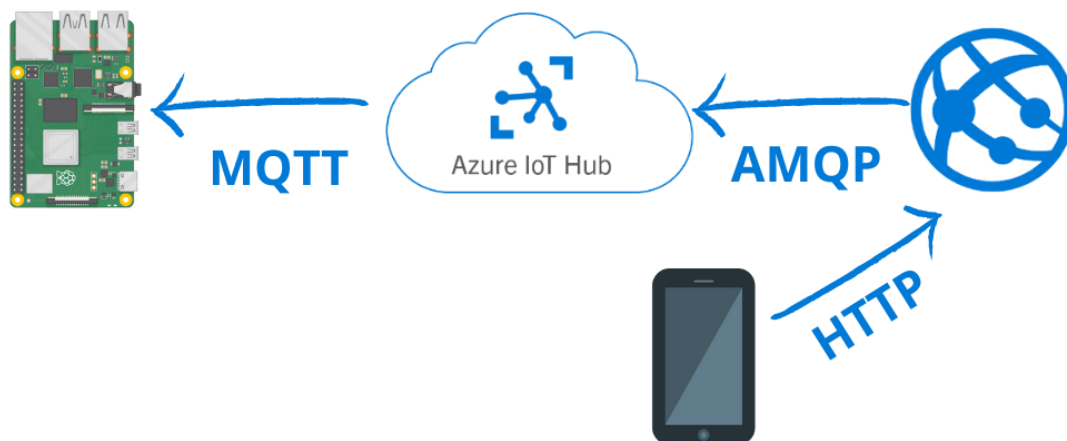


Figura 9: Comunicazione smartphone e zaino

Ogni dispositivo è accessibile dal microservizio con una connessione alla piattaforma IoT, mediante apposita stringa di connessione, specificando il dispositivo destinatario con il relativo DeviceId. Come possiamo notare dall'immagine precedente, l'utente con il proprio smartphone invierà un messaggio di registrazione al microservizio, specificando la propria mail e il DeviceId dello zaino da registrare, mediante una semplice chiamata HTTP di tipo Post; dopodiché il microservizio contatta il dispositivo, passando per Azure IoT Hub, sfruttando due canali:

- Un canale AMQP per connettersi alla piattaforma Azure, la quale si occuperà poi di inoltrare il messaggio al dispositivo
- Un canale MQTT tra zaino e Hub, in modo da rimanere in ascolto di messaggi cloud, in quanto più orientato all'utilizzo IoT

La porzione di codice Typescript del microservizio, che crea la connessione con la piattaforma Azure IoT Hub, è riportata di seguito:

```
serviceClient.open((err) => {
 if (err) {
 console.error('Could not connect: ' + err.message);
 throw err;
 } else {
 console.log('Service client connected');
 serviceClient.getFeedbackReceiver((err, receiver) => {
 if (!receiver) {
 return;
 }
 receiver.on('message', (msg) => {
 console.log('Feedback message:')
 console.log(msg.getData().toString('utf-8'));
 });
 });
 var message = new Message(data);
 message.ack = 'full';
 message.messageId = "My Message ID";
 console.log('Sending message: ' + message.getData());

 serviceClient.send(deviceId, message);

 callback();
 }
});
```

Per quanto riguarda invece la ricezione di messaggi all'interno del Raspberry Pi 4, nella seguente porzione (semplificata) di codice possiamo vedere l'utilizzo della libreria *azure.iot.device*, la quale fornisce API per connettersi al rispettivo dispositivo digitale presente in cloud, all'interno di Hub IoT. La connessione viene effettuata sfruttando l'apposita stringa di connessione del Device Twin, composta dal DeviceId e una chiave.

```

async with IoTHubSession.from_connection_string(self.
connection_string) as session:
 print("Connected to IoT Hub")
 async with session.messages() as messages:
 async for message in messages:
 TOTAL_MESSAGES_RECEIVED += 1
 if "UNREGISTER" in message.payload:
 message_to_send = {
 "type": "UNREGISTER",
 "payload": ""
 }
 self.messages_queue.put(message_to_send)
 else:
 if "REGISTER" in message.payload:
 message_to_send = {
 "type": "REGISTER",
 "payload": {
 "email": message.payload.split(";")
 [1],
 "hash": message.payload.split(";") [2]
 }
 }
 self.messages_queue.put(message_to_send)

```

Possiamo notare la facilità di apertura del client e l'invio di un messaggio al dispositivo, identificato con **deviceId** e la gestione del feedback. Va fatta una nota per quanto riguarda il feedback del dispositivo al server che lo contatta: a causa di un aggiornamento delle policy di Azure IoT Hub, più specificamente quelle riguardanti l'utilizzo di un certificato DigiCert Global G2 sui dispositivi associati al servizio, non è possibile gestire messaggi *device to cloud*, in quanto non presente alcun certificato. Perciò, per poter ricevere feedback dal dispositivo Raspberry, è necessario disporre di un certificato appropriato: per questo motivo si è preferito non gestirlo, ma solo predisporlo ad un suo utilizzo futuro.

## 4.2 Applicazione mobile

Per quanto riguarda l'applicazione mobile si è scelto di sviluppare solo la versione Android (in particolare SDK 29 o superiore) in linguaggio nativo. La scelta di sviluppare solo in Android risulta dal fatto che è più facile interagire con i sensori

rispetto ad iOS. Lo sviluppo in nativo, in particolare con Kotlin (linguaggio di riferimento per Google per lo sviluppo Android), permette di ottenere prestazioni migliori rispetto all'approccio Web App. Nell'applicazione vengono usati il sensore NFC del telefono e la fotocamera. In particolare, l'applicazione deve permettere di:

- Creare un account e accederci
- Scannerizzare un QR Code con cui si potrà associare uno zaino all'account con cui si è fatto l'accesso
- Passare un tag NFC o RFID sul retro dello smartphone per associare la copia del libro a cui è attaccato il proprio utente
- Scannerizzare il codice a barre di un libro (che rappresenta il codice ISBN a 13 cifre) per associare in modo più rapido e con meno errori il libro al tag
- Salvare su un DB interno e sul cloud l'aggiunta di nuovi tag/copie
- Ricevere aggiornamenti dei libri presenti nello zaino tramite l'uso del servizio Realtime Database di Firebase.

Al fine di realizzare un'applicazione completa e modulare si è applicata un architettura MVVM con i principi della Clean Architecture. In particolare si è divisa l'applicazione in:

- app: gestisce l'interazione con l'utente, comprende quindi la gestione della user interface e dei sensori (camera e NFC)
- accessDomain: comprende il dominio per l'accesso dell'utente
- accessData: comprende le chiamate HTTP per Access Microservice e memorizza i dati nelle shared preferences di Android (dato che si deve memorizzare la password questa viene criptata)
- desktopDomain: comprende il dominio relativo al materiale scolastico (libri e zaino)
- desktopData: comprende le chiamate HTTP per Book Microservice e il realtime Database, inoltre usa Room (basato su SQLite) per salvare i dati



- schoolDomain: comprende il dominio relativo alla scuola e agli orari delle classi e dei professori
- schoolData: comprende le chiamate HTTP per Calendar Microservice e usa Room per salvare i dati
- reminderDomain: è il punto di congiunzione di desktopDomain e schoolDomain e permette di assegnare i libri alle lezioni, inoltre permette di controllare se risulta un oggetto mancante per una data
- reminderData: comprende le chiamate HTTP per Calendar Microservice e usa Room per salvare i dati

#### 4.2.1 Gestione NFC Android

Dato che ogni costruttore può montare il lettore NFC che desidera e che la gestione è in parte delegata al sistema operativo, Android fornisce delle api per gestire la lettura di un tag. In particolare questo viene fatto dichiarando nel manifest dell'applicazione che si usa l'hardware NFC e gli intent che si intende intercettare. Un intent è una descrizione astratta di un'operazione che viene svolta o deve essere svolta.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/
 android"
 xmlns:tools="http://schemas.android.com/tools">
 <uses-permission android:name="android.permission.NFC" >
 ...
 <uses-feature
 android:name="android.hardware.nfc"
 android:required="true" />

 <application
 ...>
 <activity
 android:name=".MainActivity"
 ...>
 ...
 <intent-filter>
```

```

 <action android:name="android.nfc.action.
 TAG_DISCOVERED" />
 <action android:name="android.nfc.action.
 TECH_DISCOVERED" />
 <category android:name="android.intent.category.
 DEFAULT" />
 </intent-filter>
</activity>
</application>
</manifest>

```

L'estratto del file manifest mostra la richiesta dei permessi per l'NFC, insieme all'obbligo della presenza di hardware per la lettura. Mostrerà anche i diversi intent che l'applicazione intercetta per leggere i dati del tag.

```

override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 this.nfcAdapter = NfcAdapter.getDefaultAdapter(this)
 nfcAdapter ?: run {
 Toast.makeText(
 this,
 "NO NFC Capabilities",
 Toast.LENGTH_SHORT,
).show()
 }
 val intent = Intent(this.applicationContext, this.
 javaClass)
 intent.flags = Intent.FLAG_ACTIVITY_SINGLE_TOP
 pendingIntent = PendingIntent.getActivity(
 this.applicationContext,
 0,
 intent,
 PendingIntent.FLAG_MUTABLE,
)
 ...
}

override fun onResume() {
 super.onResume()
 nfcAdapter?.enableForegroundDispatch(this, pendingIntent, null
 , null)
}

```

```

override fun onPause() {
 super.onPause()
 nfcAdapter?.disableForegroundDispatch(this)
}

```

Nel frammento di codice estratto dalla Main Activity, si cerca l'adapter del sensore NFC e si assegna all'activity la gestione degli intent, descritti nel manifest. La scansione della ricerca dei tag viene interrotta quando l'app va in Background e riprende quando torna in Foreground.

```

override fun onNewIntent(intent: Intent?) {
 super.onNewIntent(intent)
 setIntent(intent)
 intent?.let { resolveIntent(it) }
}

@Suppress("DEPRECATION")
private fun resolveIntent(intent: Intent) {
 val action = intent.action
 if (NfcAdapter.ACTION_TAG_DISCOVERED == action ||
 NfcAdapter.ACTION_TECH_DISCOVERED == action ||
 NfcAdapter.ACTION_NDEF_DISCOVERED == action
) {
 val tag: Tag? = if (Build.VERSION.SDK_INT >= Build.
 VERSION_CODES.TIRAMISU) {
 intent.getParcelableExtra(NfcAdapter.EXTRA_TAG, Tag::
 class.java)
 } else {
 intent.getParcelableExtra(NfcAdapter.EXTRA_TAG)
 }
 tag?.let { detectTagData(it) }
 .let {
 ...
 }
 }
}

```

In questo frammento, quando si riceve un nuovo intent, se l'azione che ha causato la sua esecuzione riguarda l'NFC, si estraggono i dati, l'api cambia in base alla versione di Android.

```
fun detectTagData(tag: Tag): NFCTag {
 val sb = StringBuilder()
 val id = tag.id
 val hexId = toHex(id)
 ...
 return NFCTag(hexId, data)
}
```

L'estratto di codice da NFCTag permette di vedere come viene letto l'id del Tag. Benchè l'app necessiti solo dell'ID del tag si è scelto comunque di implementare l'intera lettura dei dati del tag, che però possono non essere sempre presenti.

#### 4.2.2 Camera

Per gestire la scansione di codici a barre e QR (necessari per registrare lo zaino), l'app android richiede la presenza di una fotocamera nel file manifest.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/
 android"
 xmlns:tools="http://schemas.android.com/tools">
 <uses-permission android:name="android.permission.CAMERA"/>
 ...
 <uses-feature
 android:name="android.hardware.camera"
 android:required="true" />
 ...
</manifest>
```

Per gestire in modo più dinamico l'interazione con l'utente viene creata una preview che mostra quello che l'utente inquadra con la fotocamera. Tramite la libreria *mlkit barcode-scanning* viene cercato un barcode nell'immagine estratta dalla preview.

```
class BarcodeAnalyser(
 val onBarcodeFound: (String) -> Unit,
 val options: BarcodeScannerOptions = BarcodeScannerOptions.
 Builder()
 .setBarcodeFormats(Barcode.FORMAT_QR_CODE, Barcode.
 FORMAT_EAN_13)
 .build(),
) : ImageAnalysis.Analyzer {
```

```

override fun analyze(imageProxy: ImageProxy) {
 val scanner = BarcodeScanning.getClient(options)
 val mediaImage = imageProxy.image
 mediaImage?.let {
 val image = InputImage.fromMediaImage(mediaImage,
 imageProxy.imageInfo.rotationDegrees)

 scanner.process(image)
 .addOnSuccessListener { barcodes ->
 if (barcodes.size > 0) {
 onBarcodeFound(barcodes[0].rawValue ?: "")
 }
 }
 .addOnFailureListener {
 }
 }
 imageProxy.close()
}
}

```

Il precedente estratto di codice mostra come data un'immagine questa venga processata dall'analyzer.

```

AndroidView(
 { context ->
 val cameraExecutor = Executors.newSingleThreadExecutor()
 val cameraProviderFuture = ProcessCameraProvider.getInstance(context)
 cameraProviderFuture.addListener({
 cameraProvider = cameraProviderFuture.get()

 val preview = Preview.Builder()
 .build()
 .also {
 it.setSurfaceProvider(previewView.surfaceProvider)
 }

 ...

 val imageAnalyzer = ImageAnalysis.Builder()

```

```

 .setBackpressureStrategy(ImageAnalysis.
 STRATEGY_KEEP_ONLY_LATEST)
 .build()
 .apply {
 setAnalyzer(cameraExecutor,
 barcodeAnalyser)
 }

 val cameraSelector = CameraSelector.Builder()
 .requireLensFacing(CameraSelector.
 LENS_FACING_BACK)
 .build()

 try {
 // Unbind use cases before rebinding
 cameraProvider?.unbindAll()

 // Bind use cases to camera
 cameraProvider?.bindToLifecycle(
 lifecycleOwner,
 cameraSelector,
 preview,
 imageCapture,
 imageAnalyzer,
)
 } catch (exc: Exception) {
 Log.e("DEBUG", "Use case binding failed", exc)
 }
}, ContextCompat.getMainExecutor(context))
previewView
},

```

Il frammento di codice mostrato in precedenza mostra come l'analizzatore di immagini viene "legato" alla preview della camera del telefono. Questo viene fatto finchè la componente grafica è attiva, quando viene chiusa il processo viene cancellato. Per rispettare la privacy dell'utente, Android, impone che prima di usare la fotocamera l'utente conceda il permesso. Questa parte, benchè non richiasta esplicitamente nel corso, viene gestita tramite la richiesta dei permessi prima dell'uso della fotocamera. Se l'utente non concede il permesso, viene mostrato un messaggio di errore e poi richiesto. Se l'utente ha negato il permesso o non è stato possi-

bile mostrare la richiesta, viene mostrato un'errore e l'utente viene direttamente portato alle impostazioni dell'app.

#### 4.2.3 Firebase Realtime database

Firebase Realtime database è un NoSQL cloud database realizzato al fine di permettere di sincronizzare i client ad esso connesso e di rimanere disponibile se l'app perde la connessione a internet. In particolare i dati vengono salvati in formato json e possono essere letti, tramite HTTP, da tutti i client, ma la funzionalità che lo caratterizza è la possibilità di sincronizzare l'istanza di dati in realtime tra applicazioni Android, Apple e JavaScript SDKs.

```
private fun getUserBackpackFirebaseReference(user: User) =
 database.reference
 .child(user.email.replace(".", "-"))

fun subscribeToBackpackChanges(user: User, backpack: String) =
 callbackFlow<Result<Set<String>>> {
 val postListener = object : ValueEventListener {
 override fun onCancelled(error: DatabaseError) {
 runBlocking {
 this@callbackFlow.send(Result.failure(
 error.toException()))
 }
 }
 }

 override fun onDataChange(dataSnapshot:
 DataSnapshot) {
 val items = dataSnapshot.children.map { it.key
 }
 runBlocking {
 this@callbackFlow.send(
 Result.success(
 items
 .filterNotNull()
 .map { it.uppercase() }
 .filter { RFIDPolicy.isValid(
 it) }
 .toSet(),
),
)
 }
 }
```

```

)
 }
}

getUserBackpackFirebaseReference(user)
 .child(backpack)
 .addValueEventListener(postListener)

awaitClose {
 getUserBackpackFirebaseReference(user)
 .child(backpack)
 .removeEventListener(postListener)
}
}

```

Il frammento di codice, estratto da `RemoteDataSourceImpl` in `DesktopData`, permette di vedere la struttura di una connessione al database. Infatti, una volta selezionato in base a quale dati si vogliono avere aggiornamenti basta implementare il metodo `onDataChange` che contiene l'istantanea del database dopo ogni modifica. Si può accedere al db anche in semplice lettura tramite HTTP. Il protocollo di comunicazione usato del sistema è quello delle WebSockets che permettono di mantenere la connessione attiva tra client e server.

### 4.3 Microservizi

All'inizio dello sviluppo si è fatta una breve analisi per determinare quale tipologia di server utilizzare per il progetto, ovvero se sviluppare un solo server che gestiva tutti gli aspetti del progetto (poco scalabile) oppure applicare un'architettura basata su microservizi, dove ognuno di essi gestisce una parte di funzionalità (correlate) e di database (molto scalabile), che si è rivelata essere la scelta per il successivo sviluppo.



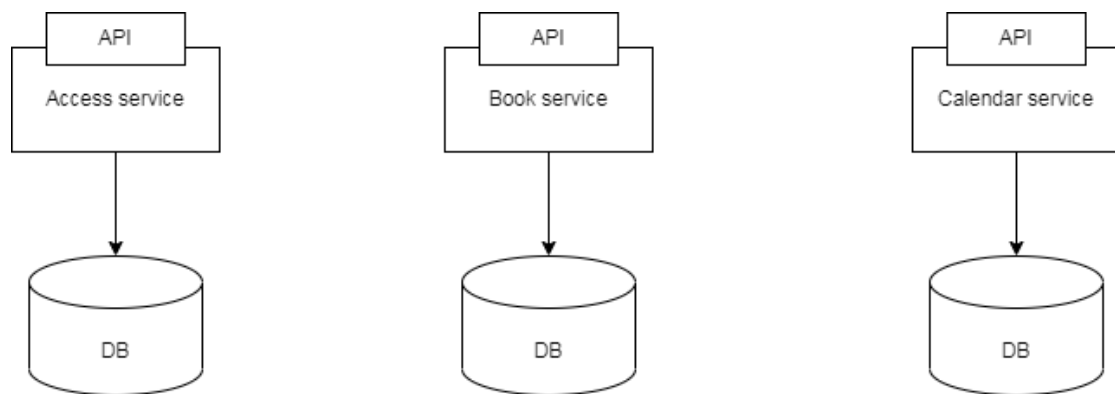


Figura 10: Rappresentazione della struttura dei microservizi

Uno dei primi problemi riscontrati nello sviluppo dei microservizi, è stata la necessità di trovare un servizio di hosting gratuito per i vari microservizi, in quanto si voleva testare fin dall'inizio il sistema nelle sue condizioni di lavoro a sviluppo completato (invece di effettuare dei deploy locali che avrebbero portato solamente a pensare in futuro a questo problema). Sono state pensate diverse soluzioni:

- Amazon AWS;
- Google Cloud Platform;
- Microsoft Azure Portal.

Al termine dell'analisi delle piattaforme, è stato deciso di risolvere il problema dell'hosting mediante l'uso della piattaforma Microsoft Azure Portal, in quanto alcuni membri del gruppo avevano già una precedente esperienza nell'uso di essa, oltre che la presenza di un account per studenti che ha fornito anche credito nel caso in cui se ne avesse avuto bisogno.

Successivamente, è stata decisa con abbastanza facilità l'utilizzo di NodeJS per i microservizi, in particolare utilizzando il linguaggio Typescript, che fornisce la tipizzazione dei dati (sicuramente più utile rispetto a JavaScript in quanto la comunicazione tramite Protobuf avviene mediante message con dati tipizzati). Oltre a ciò, si è deciso di sviluppare il server come servizio REST.

Infine, per quanto riguarda la gestione della CI/CD dei microservizi, si è provato ad usare Azure Pipeline, ma con notevoli difficoltà e problemi, quindi si è deciso di utilizzare le GitHub Actions.

Determinata la piattaforma, le tecnologie e l'architettura, si è iniziato lo sviluppo decidendo quale fosse il microservizio migliore da cui partire. In base alle necessità degli altri componenti del gruppo e all'importanza dei microservizi (oltre che alle funzionalità che erano richieste tra di essi), si è deciso di procedere con il seguente ordine di sviluppo:

1. *Access Microservice*: Microservizio che ha come scopo la gestione di tutti gli utenti, dei loro ruoli, nonché il login e la registrazione;
2. *Book Microservice*: Microservizio che ha come scopo la gestione di tutti i libri nel sistema, oltre che la gestione degli RFID degli utenti;
3. *Calendar Microservice*: Microservizio che ha come scopo la gestione del calendario scolastico, nonché tutte le lezioni dei professori e il materiale assegnato per ogni lezione.

Inoltre, durante lo sviluppo si è fatto uso della piattaforma GitHub per il versioning dei microservizi. Successivamente sono riportati e commentati alcuni frammenti di codice dei vari microservizi.

```
router.get('/getProfessorInformations', async (req, res) => {
 if(req.query.email == undefined) {
 res.status(400).send(new proto.BasicMessage({message: "You
 need to specify an email."}).toObject())
 return;
 }

 var serverResponse = await request(AccessMicroserviceURL).get(
 '/utility/emailExists').query({ email: req.query.email.
 toString()});
 if(serverResponse.statusCode != 200) {
 res.status(400).send(new proto.BasicMessage({message: "The
 professor specified does not exists"}).toObject())
 return;
 }

 const classes = await queryAsk.get_Classes_OfProfessor(req.
 query.email.toString());
 const subjects = await queryAsk.get_Subjects_OfProfessor(req.
 query.email.toString());
```

```

const institutes = await queryAsk.get_Institutes_OfProfessor(
 req.query.email.toString())
var institutesName: string[] = []
for(var val of institutes) {
 const serverResponse = await request(AccessMicroserviceURL
).get('/utility/get_istituto').query({id: val})
 institutesName.push(serverResponse.body.IstitutoNome+" / "
 +serverResponse.body.IstitutoCitta)
}

res.status(200).send(new proto.UserInformations({email_user:
 req.query.email.toString(), classes: classes, subjects:
 subjects, insitutes: institutesName}).toObject())
});

```

Il codice sopra riportato è una route del microservizio del calendario e viene utilizzato per ottenere tutte le informazioni di un professore. Questo però richiede che sia passata, come query, l'email del professore in questione. Alla fine, verranno poi ritornate tutte le informazioni riguardanti le classi, gli istituti e le materie del professore.

```

router.get('/verifyPrivileges_HIGH', async (req, res) => {
 if(req.query.email == undefined) {
 res.status(401).send(new proto.BasicMessage({ message: "Error"
 })).toObject()
 return;
 }
 if(await queryAsk.verifyPrivileges_HIGH(req.query.email.toString
 ())) {
 res.status(200).send(new proto.BasicMessage({ message: "High"
 })).toObject()
 return;
 }
 res.status(401).send(new proto.BasicMessage({ message: "Error"})
 .toObject())
});

```

VerifyPrivileges è uno dei metodi più importanti di tutti i microservizi. Questo metodo permette di verificare i permessi che ha un utente per poter accedere o meno a risorse dove non tutti gli utenti possono accedere. In questo caso, ven-

gono verificati i privilegi di tipo alto, ovvero che l'utente che esegue il metodo abbia i permessi di Istituto o Owner. Esiste anche una variante identica, chiamata `verifyPrivileges_LOW`, dove anche i professori hanno i permessi. Il metodo semplicemente controlla se è stata passata una email e se nel database, l'utente con l'email passata ha i determinati requisiti per poter accedere alla risorsa richiesta.

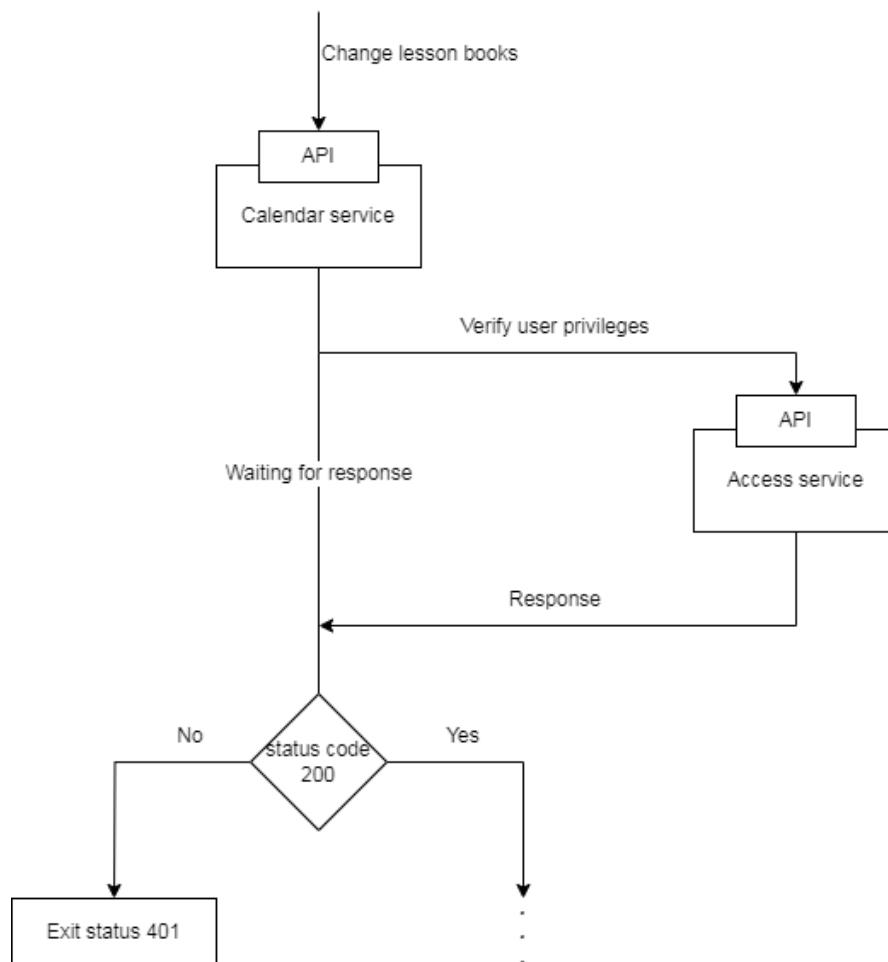


Figura 11: Esempio di chiamata tra microservizi durante l'esecuzione di una route

## 4.4 Database

Come per i microservizi, la prima parte dello sviluppo si è concentrata nel determinare quale sia la piattaforma migliore. Le scelte principali si sono concentrate su:

- Amazon AWS;
- Google Cloud Platform;
- Microsoft Azure Portal.

Alla fine, per questioni di conoscenza e di utilizzo, si è optato per utilizzare la piattaforma di Microsoft Azure Portal. Inoltre, grazie al credito fornito agli studenti, si è stati in grado di non avere spese per il database (in quanto il costo è di 5€/mese).

Per lo sviluppo del database, si è costruita una struttura all'inizio, in grado di gestire tutte le informazioni e casistiche generali. Nel caso che, durante lo sviluppo ci si trovasse con la mancanza di tabelle/colonne, veniva effettuata una breve analisi per definire quali fossero le modifiche da apportare al database per sistemare il problema e anche possibili futuri cambiamenti che sarebbero stati necessari (ma non previsti precedentemente) durante la fase di progettazione del database.

```
CREATE TABLE Lezione (
 ID integer NOT NULL IDENTITY(1,1),
 ID_Calendarario integer NOT NULL,
 Nome_lezione VARCHAR(255) NOT NULL,
 Materia integer NOT NULL,
 Professore VARCHAR(255) NOT NULL,
 Ora_inizio TIME NOT NULL,
 Ora_fine TIME NOT NULL,
 Giorno VARCHAR(20),
 PRIMARY KEY (ID))
```

```
CREATE TABLE Libro (
 ISBN varchar(17) NOT NULL,
 Titolo varchar(255) NOT NULL,
 Autore varchar(255) NOT NULL,
 Data_Pubblicazione DATE NOT NULL,
 PRIMARY KEY (ISBN))
```

```
CREATE TABLE LibroPerLezione (
 ID_lezione integer NOT NULL,
 ISBN VARCHAR(17) NOT NULL,
 PRIMARY KEY (Id_lezione, ISBN))
```

Qui sopra sono riportate le tabelle presenti nel database per la gestione delle lezioni, i libri e i libri assegnati ad una lezione.

Essendo che i microservizi possono comunicare solamente con una porzione del database, ogni microservizio dovrebbe avere il suo database interno, non condiviso con nessun altro microservizio. Per ragione di costi, questa scelta non è stata possibile adottarla per il progetto. Si è pertanto passati al creare un solo database, dove i microservizi salvano i propri dati, ma separato nella struttura. Ovvero, sono presenti tutte le tabelle, ma si hanno connessioni tra di loro solo se appartengono allo stesso microservizio. Ciò permette di simulare le interazioni tra le tabelle, anche se un microservizio avrebbe la possibilità di interrogare anche tabelle di non sua appartenenza (questa cosa è stata evitata durante lo sviluppo del database). Qui sotto è possibile visualizzare la struttura che si è adottata all'interno del database.

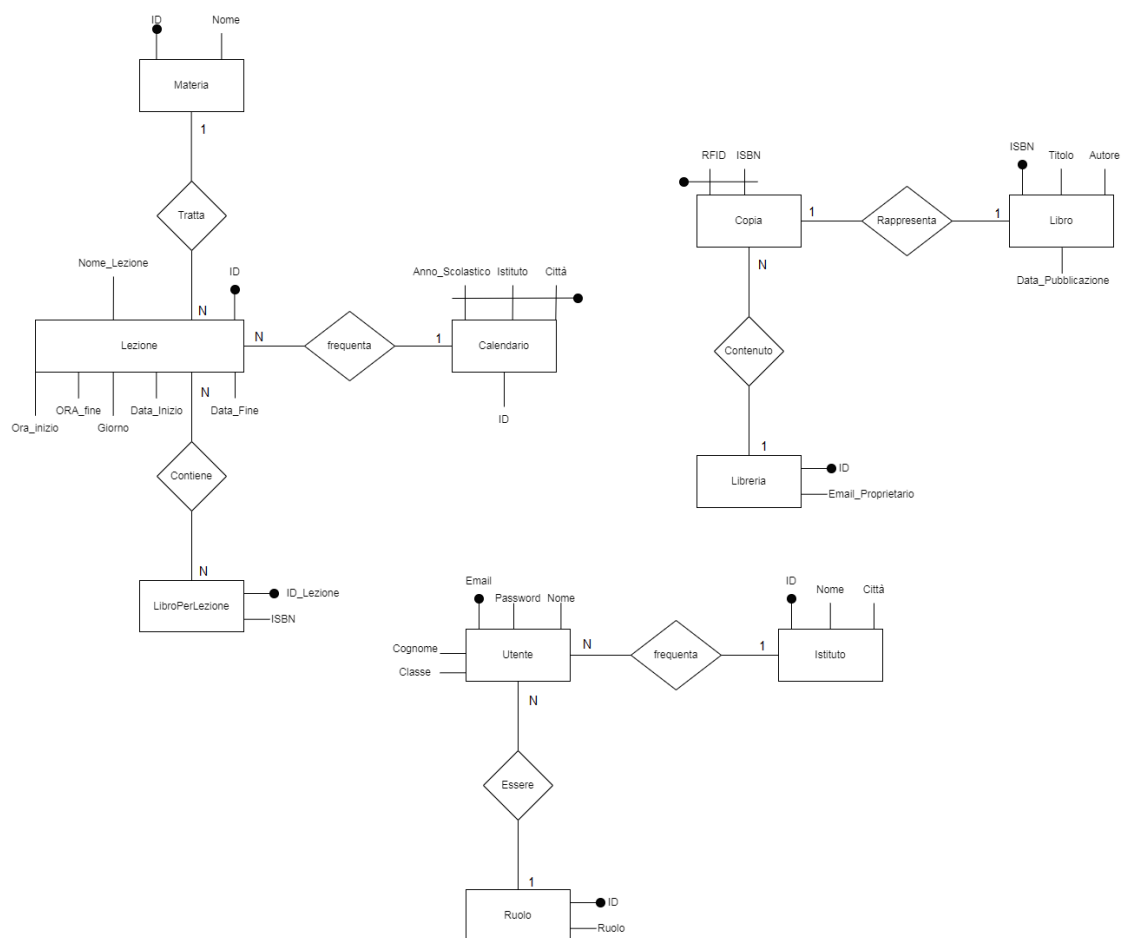


Figura 12: Schemi ER dei database

Come si può notare, si è applicata una divisione nelle 3 macroaree che coincidono con i micorservizi. Queste separazioni sono completamente indipendenti tra di loro, in quanto modellano database diversi. Quindi, le modifiche che vengono apportate ad informazioni su un database non vengono riportate automaticamente anche agli altri database, ma devono essere inviate le informazioni ai relativi servizi per far sì che modifichino il valore con il dato opportuno.

## 5 Testing e performance

Avendo adottato un approccio basato su CI/CD, durante tutta la fase di sviluppo sono stati sviluppati e adottati opportuni test per verificare che tutti i sistemi funzionassero correttamente e rispondessero nel modo opportuno. In particolare, ogni componente del gruppo è stato testato utilizzando sia localmente per verificarne il corretto funzionamento durante la fase di sviluppo, sia in remoto durante la fase di deploy/pubblicazione tramite GitHub Actions.

Ogni componente del gruppo parlerà successivamente delle tecniche di testing che ha implementato e delle relative performance ottenute, ma va precisato che le performance del sistema sono limitate dai piani a basso costo utilizzati per tutto ciò che concerne il Cloud, per rimanere nell'ambito prototipale.

### 5.1 Raspberry Pi 4 e IoT Hub

Per validare la parte di dominio e repository (unicamente database locale) sono stati realizzati test in Python ad hoc per provarne la correttezza e la funzionalità. Altri moduli, come quello relativo alla rete o alla lettura di tag RFID, sono stati testati manualmente sul dispositivo fisico, pertanto non è stato raggiunto un livello altissimo (circa 65%) di coverage. La parte comunicativa tra dispositivi e Raspberry è stata validata mediante l'utilizzo della dashboard di Azure Portal, che permette di inviare un messaggio qualsiasi ad un preciso dispositivo registrato. Prima di tutto avviamo il modulo del dispositivo che gestisce le comunicazioni con IoT Hub; dopodiché possiamo inviare un messaggio qualsiasi per verificarne l'effettiva ricezione e quindi la corretta comunicazione.



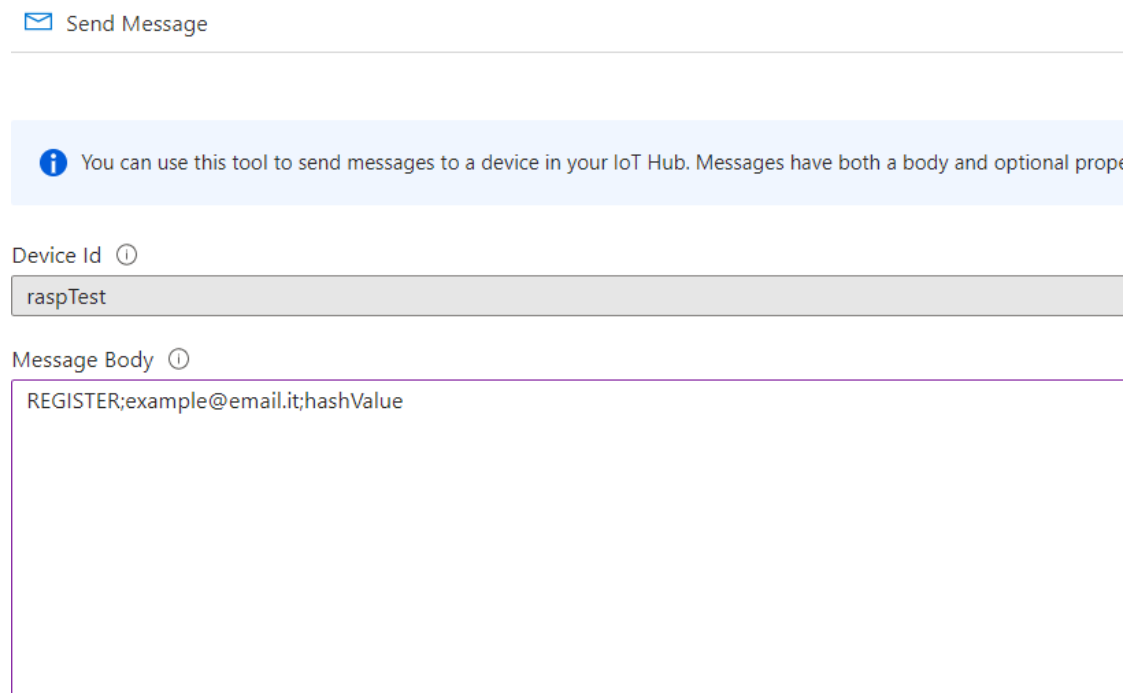


Figura 13: Invio di un messaggio tramite Azure Portal

Come possiamo vedere, nell'applicativo che esegue sul dispositivo è stato ricevuto un nuovo messaggio, ovvero quello inviato correttamente.

```
(python) D:\Universita\Magistrale\PROGETTONE\Python\IntelligentBackpack\src\main>python IntelligentBackpackService.py
Thread 'Thread#Network' avviato
Starting C2D sample
Press Ctrl-C to exit
Connecting to IoT Hub...
Connected to IoT Hub
Message received with payload: REGISTER;example@email.it;hashValue
REGISTRATO
```

Figura 14: Ricezione di un messaggio su dispositivo Raspberry

Non c'è limite al numero di dispositivi che possiamo aggiungere all'Hub IoT della piattaforma Azure, la quale permette di scalare le performance in base all'utilizzo scegliendo piani di fatturazione più costosi e performanti. Stessa cosa riguarda il microservizio che gestisce il registro degli zaini, il numero di richieste che possono essere soddisfatte nell'unità di tempo e velocità di risposta dipende fortemente dal piano tariffario scelto, scalabile nel momento in cui si mostra l'esigenza. Per

quanto riguarda l'accesso al database da parte del microservizio di gestione dei dispositivi, le latenze risultano abbastanza elevate, spaziando da pochi secondi ad un minuto, per via del piano tariffario non proprio ottimale dal punto di vista delle prestazioni.

## **5.2 Applicazione Smartphone**

### **5.2.1 Testing**

Per parte di dominio vengono creati degli unit test e, dove decessari, vengono creati mock delle classi per facilitare i test. La parte di gestione del sistema android viene testato tramite `instrumentTest` che permette di testare come vengono salvati i dati in modo permanente. La parte di comunicazione con il realtime Database è stata validata tramite modifiche online del database e verifica su emulatore o dispositivo reale. Il test del sensore NFC e della camera sono stati effettuati tramite dispositivo reale e carte NFC e barcode reali di libri. Infine, per testare l'intera applicazione, si è effettuato un test completo con i microservizi e il raspberry.

## **5.3 Performance**

Nel test delle performance, svolte manualmente, si riscontrano problemi di latenza con i microservizi (verrà analizzato in seguito), mentre per il recupero dei dati in locale si riscontrano dati accettabili. In particolare l'uso del servizio realtime Database permette di ottenere i cambiamenti in tempi realtime (minori di 200 millisecondi)

## **5.4 Microservizi**

### **5.4.1 Testing**

I test dei microservizi sono stati eseguiti tutti nello stesso modo, senza differenziare da microservizio a microservizio (ovviamente, i test sono diversi, ma la struttura e le tecnologie utilizzate sono le stesse).

I test sono stati programmati utilizzando il linguaggio di programmazione TypeScript, utilizzando un modulo esterno chiamato Jest, utilizzato per testare metodi e funzioni. Inoltre, Jest ha anche la funzionalità di eseguire la coverage dei test,

per verificare la percentuale di quanto codice è stato verificato mediante l'uso dei test (una buona soglia è circa 80%)

Essendo che Jest viene utilizzato per testare metodi, il suo utilizzo per testare un server HTTP è pressochè nullo, in quanto tutte le chiamate HTTP che esso riceve sono sviluppate in delle routes che non sono possibili da chiamare dall'interno del codice. Per risolvere questo problema, sono state pensate due possibili soluzioni:

1. Modificare le route, andando a rimuovere il codice dalla route ed inserendolo all'interno di una funzione dedicata, che avrebbe ricevuto gli stessi input della route ed avrebbe eseguito il codice presente originariamente nella route. Questa soluzione è stata scartata in quanto si andava a rimuovere parte del significato della route, oltre ad aggiungere metodi inutili che possono essere evitati usando le route come si dovrebbe;
2. Utilizzare un modulo esterno per il testing delle route (soluzione adottata).

Infatti, grazie al modulo Supertest, è possibile effettuare chiamate HTTP direttamente al server (nel nostro caso, il microservizio che veniva testato) eseguito in locale, senza avere la necessità di effettuare un deploy remoto precedente.

Mesiante questi 2 moduli, tutte le funzionalità sono state testate prima di effettuare il deploy, assicurandosi che il microservizio pubblicato sia conforme con le funzionalità progettate.

In particolare, dai test si è ottenuta anche un'alta coverage dei sistemi, come riportato dalla foto riposata qui sotto.

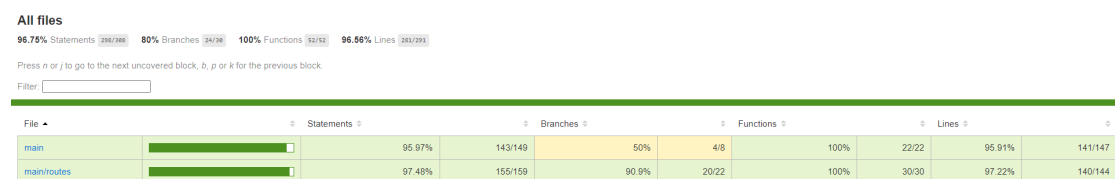


Figura 15: Coverage ottenuta sul microservizio di accesso

## 5.4.2 Performance

Le prestazioni sono state testate mediante test manuali. Infatti, tramite l'utilizzo di test automatici (sia locali, che remoti su GitHub), si è notata un'altissima latenza (alcuni test, per eseguire una query base, richiedevano anche tempi superiori

ai 30 secondi). Questo non si è capito se è dovuto dall'utilizzo di server gratuiti e un database poco potente, ma andando successivamente ad eseguire test manuali (utilizzando postman o eseguendo direttamente le query nel browser), si otteneva che le chiamate rispondevano nell'arco di millisecondi (anche le più complesse non hanno superato il secondo di delay).

Tramite questi test manuali, si sono potuti verificare anche i requisiti di performance richiesti per il progetto.

## 6 Analisi di deployment su larga scala

Una volta terminata la fase di sviluppo e realizzato il deploy di tutti i servizi, si è effettuato un colloquio con tutti i membri del gruppo per discutere di come sarebbe da gestire un deployment su larga scala e le sue possibili criticità. Successivamente saranno trattate le principali criticità, divise per componente.

### 6.1 Raspberry Pi 4

Qui sotto saranno elencate le principali criticità che devono essere risolte/discusse durante la fase di deploy su larga scala.

- *Portable*: Uno dei principali problemi è quello di inserire l'hardware nello zaino. Questo può essere fatto in 2 modi:
  1. Insieme ad una compagnia esistente, produrre uno zaino con la tecnologia già inserita all'interno delle cuciture. La scelta è particolarmente costosa e rischia di non avere comunque successo, in quanto non tutti gli utenti prenderebbero quel brand di zaini solo per la funzionalità smart (costo troppo elevato / preferenze di altre marche);
  2. Sviluppare un contenitore/modulo portatile, per far sì che la componente hardware sia contenuta all'interno di esso e si adatti facilmente agli zaini esistenti. Ciò permetterebbe l'utilizzo del proprio zaino preferito dall'utente e andando semplicemente ad inserire il modulo smart per rendere lo zaino connesso.

### 6.2 Applicazione smartphone

Qui sotto saranno elencate le principali criticità che devono essere risolte/discusse durante la fase di deploy su larga scala.

- *Multipiattaforma*: Essendo che l'applicazione è destinata a tutte le persone, l'applicazione deve essere portata anche su dispositivi iOS (originariamente è sviluppata solo per dispositivi Android). Per risolvere questa criticità la scelta migliore sarebbe riscrivere l'applicazione in Flutter in modo da gestire in modo efficiente le parti in comune ed eventualmente scrivere in nativo le interazioni hardware per cui non sono presenti librerie.

- *Compatibilità*: Deve essere verificato il corretto funzionamento con il più alto numero di dispositivi, cercando andando a modificare (dove possibile) l'applicazione per renderla il più retrocompatibile possibile. A questo fine l'applicazione contiene i componenti di Firebase crashlytics che segnala eventuali errori non gestiti. L'applicazione, con sdk 29 (android 10), raggiunge al momento compatibilità con il 78.5% dei dispositivi android.

### 6.3 Microservizi

Qui sotto saranno elencate le principali criticità che devono essere risolte/discusse durante la fase di deploy su larga scala.

- *Determinare se è necessario aumentare la capacità di computazione del database e dei server*: Come si è evidenziato nel paragrafo del testing e prestazioni dei microservizi, i tempo di risposta di alcuni microservizi erano eccessivamente alti durante i test automatici. Per questo bisognerebbe effettuare un'attenta analisi e determinare quali sono i fattori principali che determinano questo fattore (se sono problemi interni ai moduli jest/supertest utilizzati, oppure altre cause);
- *Aumento della capacità computazionale del database*: La potenza attuale del server è di 4DTU (unità di misura utilizzata da microsoft per rappresentare la misura dell'utilizzo della CPU, memoria e operazioni di read-write insieme), con una memoria totale di 2GB. Nel caso in cui si passi a vendere il prodotto su scala nazionale/internazionale, è necessario disporre di una quantità di memoria nettamente superiore a quella attualmente presente (basti pensare come, solamente utilizzando i dati dei test, si arrivava ad uno spazio occupato di 27MB);
- *Nuovi server*: Essendo che, nel caso di un deploy su larga scala, una grande quantità di utenti utilizzerebbero il servizio, è necessario aumentare la capacità dei server per poter gestire al meglio il crescente numero di richieste. La scelta migliore, per limitare anche i costi, sarebbe di avere un piano personalizzato per i server (nel caso l'hosting dell'applicazione sia in cloud), dove il costo dell'hosting varia in base all'utilizzo, in quanto si hanno orari di massimo utilizzo e orari dove non si ha tanta attività.



## 8 Conclusioni

In conclusione, il lavoro svolto ha rispettato i requisiti richiesti, andando a sviluppare un sistema in grado di gestire tutti gli aspetti di uno smart bag.

Anche i tempi di consegna sono stati rispettati, in quanto il progetto è stato richiesto su una base temporale di massimo 3 mesi totali (il progetto è stato completato in un lasso temporale di 2 mesi).

Al termine del progetto sono stati individuati alcuni aspetti che potrebbero essere migliorati per poter rendere ancora migliore il servizio (o che sono stati semplificati per via del tempo/budget a disposizione).

1. Inserimento ed utilizzo di un modulo GSM per avere connettività mobile indipendente e un modulo GPS per ottenere la posizione geolocalizzata del dispositivo;
2. Implementazione di un Reminder Engine, in grado di poter notificare l'utente oggetti non inseriti necessari alla lezione/evento del giorno successivo, in totale autonomia;
3. Implementazione di un sistema di aggiornamento automatico di tutti i dispositivi Raspberry nel momento di rilascio di una nuova versione del firmware;
4. Inserimento di un salt value negli utenti, per rendere ancora più efficace l'hash delle password (rende l'hash univoco, in modo tale che anche 2 password identiche hanno un valore di hash differente).