

Elaborato di Sistemi Distribuiti



Baiardi Alex

`alex.baiardi@studio.unibo.it`

Frattini Francesca

`francesca.frattini7@studio.unibo.it`

Per il progetto di Sistemi Distribuiti, si è pensato di realizzare un'applicazione che consenta di gestire una serra "smart", in modo da poter visualizzare un elenco di tutti i tipi di ortaggi coltivati e gestire il loro ambiente di sviluppo.

Per poter utilizzare l'applicativo, un utente deve registrarsi, in modo tale da avere la visualizzazione esclusiva della propria serra.

Ogni utente ha la possibilità di: visualizzare le piante presenti in serra, gestire l'irrigazione delle piante e monitorare i dati relativi alla qualità dell'aria all'interno della serra.

Il progetto comprende anche una componente di Internet of Things, in quanto l'applicazione comunica con "oggetti intelligenti" tramite la connessione di rete: nello specifico, sono presenti sensori per rilevare dati ambientali come temperatura, umidità, pressione e livello di CO₂ e un attuatore che controlla l'impianto di irrigazione.

Indice

1	Obiettivi/Requisiti	5
1.1	Scenarios	5
1.2	Metodi di autovalutazione	6
2	Analisi dei requisiti	7
2.1	Requisiti funzionali	7
2.2	Requisiti non funzionali	7
2.3	Analisi del dominio	8
3	Design	10
3.1	Struttura	10
3.1.1	Struttura del database	10
3.1.2	Struttura componenti IoT	11
3.1.3	Struttura del progetto: moduli e dipendenze	12
3.1.4	Struttura del modulo common	13
3.1.5	Struttura del modulo serverBackend	14
3.1.6	Struttura modulo mobileApp	16
3.1.7	Struttura del modulo sensors	20
3.1.8	Struttura modulo sensorDataService	22
3.1.9	Struttura modulo irrigationControlUnit	22
3.2	Comportamento	23
3.2.1	Comportamento del Server	23
3.2.2	Comportamento del Client Mobile	24
3.3	Interazione	26
4	Dettagli implementativi	30
4.1	Tecnologie utilizzate	30
4.2	Dettagli implementativi Server	32
4.2.1	Gestione validazione richieste HTTP	32
4.3	Dettagli implementativi Client Mobile	33
4.3.1	Gestione delle Activity e Fragment	33
4.3.2	Gestione e condivisione dei dati tra Fragment	34
4.3.3	Gestione invio delle richieste con Volley	35
4.4	Dettagli implementativi parte IoT	37
4.4.1	Utilizzo di RabbitMQ per la comunicazione delle statistiche	37
5	Autovalutazione/Validazione	40
5.1	Testing	40
5.1.1	Testing del Server	40
5.1.2	Testing del Client Mobile	41
5.2	Testing dei componenti IoT	41

5.3	Modalità di lavoro	42
6	Deployment	43
6.1	Containerizzazione	43
6.2	Istruzioni di deployment	44
6.2.1	Setup ed installazioni di base	44
6.2.2	Containerizzazione	44
6.2.3	Setup del Client Mobile	45
7	Esempi di utilizzo	46
8	Conclusioni	50
8.1	Lavori futuri	50
8.2	Cosa è stato appreso	51

Elenco delle figure

1	Diagramma dei casi d'uso	6
2	Entità del dominio e le loro interazioni	8
3	Macro-struttura del sistema	10
4	Dipendenze dei moduli del progetto	13
5	Diagramma delle classi: esempio con Serializer e Deserializer di User	14
6	Diagramma delle classi: UserRoutes(API) e UserController	16
7	Livelli principali dell'applicazione mobile	17
8	Struttura applicazione mobile	19
9	Diagramma delle classi: gestione lista piante	20
10	Schema dello scambio di messaggi tramite RabbitMQ	21
11	Architettura di gRPC	23
12	Diagramma degli stati del Server	23
13	Diagramma degli stati: comportamento del Client	25
14	Diagramma degli stati: stato composto Plant Gallery	26
15	Diagramma di sequenza: interazione tra le entità per la registrazione di un utente	27
16	Diagramma di sequenza: interazione tra le diverse entità per l'inse- rimento di una pianta	28
17	Schermata di registrazione	46
18	Pagina di login	46
19	Homepage dell'applicazione	47
20	Schermata di aggiunta pianta	47
21	Elenco settori	48
22	Galleria piante settore 3	48
23	Impostazioni pianta	48
24	Schermata statistiche serra	49
25	Pagina gestione irrigazione	49

1 Obiettivi/Requisiti

Si intende realizzare un'applicazione per la gestione di una serra all'interno della quale sono presenti coltivazioni per la produzione di verdure ed ortaggi.

L'idea nasce dalla necessità sempre più frequente di dover controllare l'ambiente di sviluppo delle piante e gestirne l'annaffiatura.

La serra è suddivisa in settori per una migliore gestione logistica e per gestire l'irrigazione delle colture.

Il servizio è utilizzabile mediante applicazione Android.

Ad ogni utente, dopo che ha effettuato la registrazione, viene associata una galleria che rappresenta la propria serra fisica, un sensore per la qualità dell'aria e un attuatore per la gestione dell'irrigazione delle piante.

Per ottimizzare l'irrigazione all'interno della serra, vengono individuati dei settori contenenti diverse specie di piante.

L'utente può aggiungere piante, indicando il settore in cui sono collocate; successivamente, è possibile modificare le piante o eliminarle definitivamente dalla galleria.

È presente una sezione dedicata al monitoraggio dell'ambiente della serra, dove un sensore comunica in tempo reale i dati relativi a temperatura, umidità, pressione e livelli di CO₂.

L'irrigazione viene gestita direttamente dall'utente, il quale può decidere in quali settori attivare gli irrigatori e disattivarli successivamente.

Infine, si intende realizzare un sistema che in grado di gestire le richieste provenienti dai diversi utenti; l'applicazione Android dovrà funzionare correttamente su diversi dispositivi con differenti caratteristiche hardware.

1.1 Scenarios

Il target di utenti per cui è pensata l'applicazione comprende sia la singola persona che vuole iniziare a coltivare in serra ed intende organizzarsi, sia un'azienda agricola che deve gestire serre di grandi dimensioni.

Supponiamo che un utente vuole utilizzare l'applicazione per organizzare la propria serra, gestendo l'ambiente all'interno di essa.

Una volta aperta l'applicazione, all'utente verrà richiesto di registrarsi oppure di effettuare il login in caso possieda già un account.

Una volta effettuato l'accesso, si aprirà un menù con diverse opzioni: visualizzare la galleria di piante, aggiungere una pianta alla serra, attivare/disattivare l'irrigazione e controllare i valori ambientali della serra.

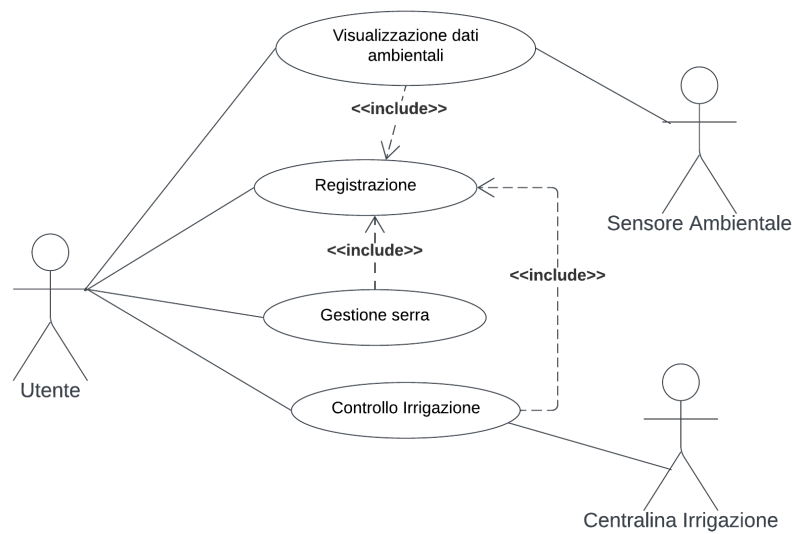


Figura 1: Diagramma dei casi d'uso

1.2 Metodi di autovalutazione

Per valutare l'efficacia e l'efficienza del software generato, si intendono utilizzare test automatici e manuali. In particolare, si pensa di testare la maggior parte delle funzionalità di backend mediante test automatici, mentre la reattività dell'interfaccia grafica dell'applicazione mobile verrà testata tramite test perlopiù manuali: dove possibile, saranno sfruttati anche test automatici.

I dettagli circa i test sono approfonditi nella sezione 5.

Per quanto riguarda il testing della qualità del servizio offerto, si verificherà se l'applicazione sia in grado di soddisfare i requisiti emersi dall'analisi e come essi vengono soddisfatti, e se gli obiettivi stabiliti sono stati rispettati e il sistema realizzato risulta essere efficiente.

2 Analisi dei requisiti

L'applicazione ha come obiettivo principale quello di ottimizzare la gestione di una serra dotata di tecnologie IoT, a tal proposito sono stati individuati i seguenti requisiti funzionali e non funzionali.

2.1 Requisiti funzionali

- All'avvio dell'applicazione, sarà visualizzata la schermata di login, attraverso la quale sarà possibile effettuare l'accesso con le proprie credenziali (email, password) oppure registrare un nuovo utente.
- Dopo aver effettuato l'accesso, sarà possibile:
 - Aggiungere una nuova pianta alla serra, fornendo le seguenti informazioni:
 - * Specie;
 - * Settore in cui è stata piantata;
 - * Data di semina (opzionale)
 - * Immagine della pianta (opzionale)
 - Visualizzare le piante presenti in serra, ordinate in base al settore della serra in cui sono state piantate;
 - Attivare/disattivare il sistema di irrigazione direttamente dall'applicazione: ogni settore ha un rubinetto dedicato che può essere attivato o disattivato autonomamente;
 - Visualizzare i valori relativi all'ambiente della serra (temperatura, umidità, pressione atmosferica, quantità di CO₂)

2.2 Requisiti non funzionali

- Il servizio deve essere in grado di consentire l'accesso all'applicazione a più utenti contemporaneamente;
- Il design deve essere scalabile e facilmente estendibile per modifiche future o eventuali funzionalità aggiunte;
- L'applicazione deve funzionare su dispositivi Android diversi, con caratteristiche hardware diverse, dotati di una versione di sistema minima pari alla 9.0 (*Pie*);
- L'interfaccia utente deve essere reattiva alle interazioni con l'utente;
- Per quanto riguarda l'usabilità: l'utilizzo dell'applicativo deve essere semplice e chiaro, in modo tale che qualsiasi utente possa essere in grado di utilizzarlo.

2.3 Analisi del dominio

Poichè l'applicazione consiste in un *Web Service* a disposizione degli utenti, il programma viene progettato per aderire al modello di servizi **RESTful**, che si compone di un'interfaccia uniforme riutilizzabile del servizio offerto.

Le entità principali del dominio che si intende modellare sono (vedi figura 2):

- Il **Client mobile**, che invierà richieste al Server specificando opportune rotte, le quali consentiranno l'esecuzione di operazioni;
- Il **Server**, che avrà il compito di occuparsi di esporre il servizio al Client, di definire le rotte ed implementare le operazioni effettuabili, mediante l'interazione con un database non relazionale;
- Un **database non relazionale**, incaricato di occuparsi delle informazioni relative a utenti, settori, piante, valori registrati ... e, su richiesta del Server, fornire questi dati per eseguire determinate operazioni

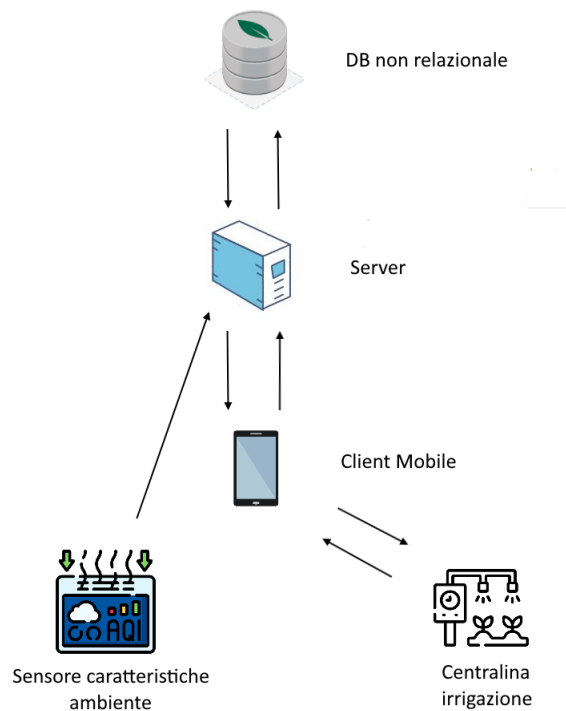


Figura 2: Entità del dominio e le loro interazioni

- Dei **dispositivi IoT**, che comprendono dei sensori per il rilevamento delle caratteristiche dell'ambiente (temperatura, umidità, pressione e livelli di CO₂) e una centralina che consente di attivare e disattivare da remoto l'irrigazione nella serra.

Per quanto riguarda i dispositivi IoT che integrano l'applicativo, i quali comprendono i sensori per il rilevamento dei valori ambientali e degli attuatori per l'irrigazione, essi saranno simulati tramite applicazioni dedicate, in quanto non si dispone fisicamente di essi.

3 Design

Dopo aver completato la raccolta di informazioni nelle varie fasi di analisi, si è proseguito con la fase di progettazione, che consiste nella definizione della struttura delle varie entità coinvolte e le loro interazioni, i cui dettagli sono trattati nelle prossime sezioni

3.1 Struttura

Il progetto definisce un sistema che è costituito da: un database non relazionale, Server Web, Client Mobile e componenti IoT per la gestione dei sensori e una centralina per la gestione dell'irrigazione. La macro-struttura del sistema è rappresentata dalla figura 3.

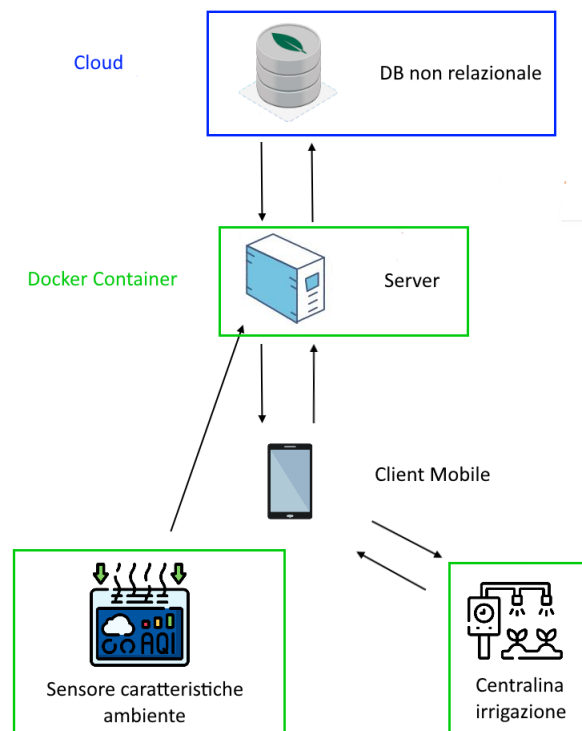


Figura 3: Macro-struttura del sistema

3.1.1 Struttura del database

Il database sul quale si appoggia l'applicativo è un database MongoDB, quindi non relazionale.

Le entità che lo compongono sono le seguenti:

- **User** - rappresenta gli utenti iscritti all'applicativo;

- **Plant** - rappresenta le piante presenti nelle serre dei vari utenti;
- **Stats** - rappresenta le statistiche relative alle varie rilevazioni effettuate dai dispositivi IoT per ogni utente,
- **Session** - entità auto-generata dalla libreria che gestisce il login e la registrazione, permette di conservare nel db i *cookies* e le informazioni relative alla sessione,
- **Irrigation Unit** - questa entità permette di tenere traccia dell'associazione tra un utente e la relativa centralina per l'irrigazione.

Questo tipo di database è conosciuto anche come *NoSQL* (Not only SQL), difatti le interrogazioni all'interno della base di dati vengono effettuate tramite query JSON.

MongoDB, in particolare, è di tipo documentale: la differenza principale con un database relazionale è che i dati non vengono memorizzati nel rigido modello di dati righe-colonne, bensì ogni record è registrato come un documento in formato JSON, dotato di determinate caratteristiche. Altri vantaggi dei database NoSQL sono la possibilità di salvare i dati in qualsiasi momento senza la necessità di effettuare verifiche e la scalabilità.

MongoDB Atlas, la piattaforma utilizzata per implementare la base di dati, è un servizio di database *multi-cloud* dotato dei massimi livelli di sicurezza e privacy, che permette di selezionare un provider di servizi cloud tra quelli disponibili (AWS, Azure e GCP)[1].

Il servizio *Database-as-a-Service* (DBaaS) consente di creare, implementare e scalare un database senza dover disporre fisicamente di hardware e software performanti, in quanto tutto ciò viene gestito dal *cloud provider* che fornisce all'utente una base di dati *ready-to-use*.

3.1.2 Struttura componenti IoT

Nella serra intelligente è possibile controllare in tempo reale le statistiche relative all'ambiente (temperatura, umidità, pressione e livelli di CO₂) e gestire da remoto l'innaffiatura dei diversi settori dove vengono coltivate le piante.

Per ragioni pratiche, queste componenti sono state simulate, non disponendo dei mezzi adeguati per testarli sul campo. Questo non ha impedito di pensarli come dispositivi indipendenti e raggiungibili da remoto, progettando per ciò gli opportuni meccanismi di comunicazione.

L'idea per la gestione di questa parte è quella per cui ogni utente che utilizza l'applicazione abbia in dotazione una centralina per la rilevazione dei dati ambientali e una per la gestione dell'irrigazione.

Per la gestione dei sensori si pensa ad un dispositivo che quando viene avviato periodicamente effettua le rilevazioni e le comunica per la memorizzazione nel

database. La comunicazione ha come controparte un servizio che sarà progettato appositamente per raccogliere i dati dalle varie centraline degli utenti e inviare questi dati al server web per la memorizzazione.

La centralina dell'irrigazione è pensata come un dispositivo che offre la possibilità di comandare i vari settori che gestisce da remoto connettendosi direttamente all'applicazione mobile.

3.1.3 Struttura del progetto: moduli e dipendenze

Il sistema è stato quindi progettato, secondo una logica di modularità e indipendenza dei vari componenti in cui ciascuno di essi è racchiuso in un modulo, condividendo ciò che viene utilizzato per definire gli aspetti comuni.

I moduli implementati sono i seguenti:

- **serverBackend** - contiene il servizio esposto dal Server nonché la definizione delle API del servizio stesso, specificando le rotte che permettono di manipolare le risorse all'interno del database;
- **mobileApp** - questo modulo, definisce il *Client Mobile* dell'applicazione. Il modulo comprende l'insieme dei componenti necessari alla creazione dell'applicazione Android;
- **sensors** - rappresenta la parte del progetto che simula il funzionamento della centralina che gestisce alcuni sensori. In particolare, si occupa di simulare le rilevazioni e pubblicare i dati su un servizio per la gestione di code di messaggi;
- **sensorDataService** - questo modulo contiene il servizio che ha il compito di raccogliere le informazioni inviate dal simulatore dei sensori per la qualità ambientale e provvede a comunicarli al web server affinché possa memorizzarli nel database;
- **irrigationControlUnit** - rappresenta il componente per la simulazione della centralina dell'irrigazione. Il suo compito è quello di rimanere in ascolto di richieste provenienti dal Client Mobile.
- **common** - in questo modulo sono presenti tutti i concetti utilizzati da più moduli differenti, in modo tale da non avere codice ridondante. In particolare, sono presenti le classi che definiscono le diverse entità e le strutture di serializzazione e deserializzazione necessarie per convertire gli oggetti nel loro formato JSON e viceversa.

Come si può osservare in figura 4, i moduli **sensors** e **sensorDataService** dipendono entrambi dal modulo **common**, mentre **mobileApp** sfrutta le classi contenute in **irrigationControlUnit** e **common** per il suo funzionamento.

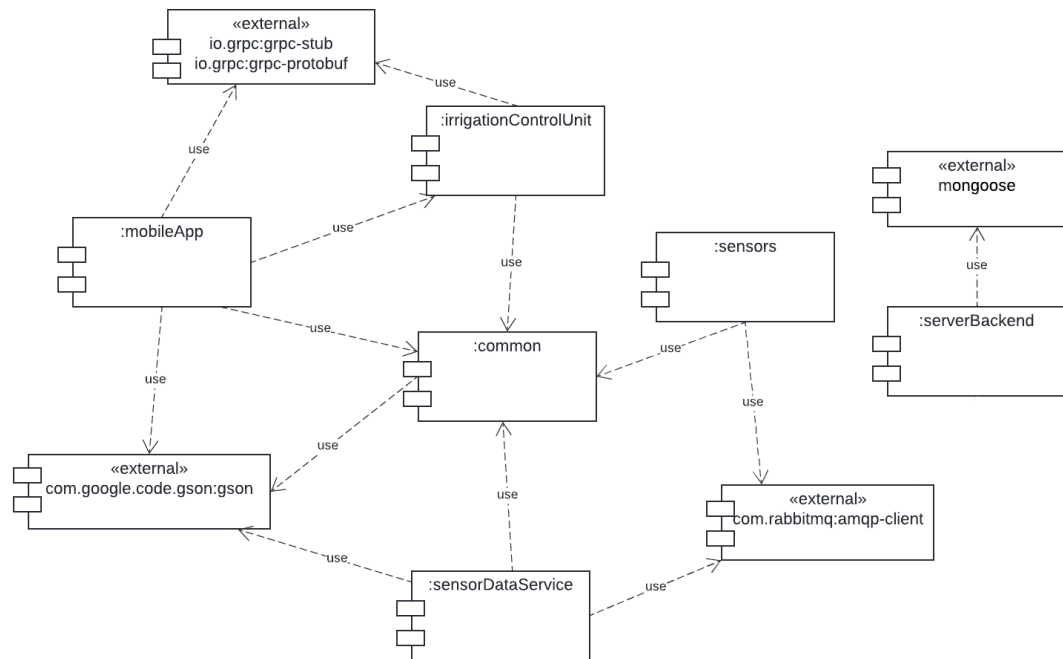


Figura 4: Dipendenze dei moduli del progetto

3.1.4 Struttura del modulo common

In questo modulo vengono definiti gli aspetti comuni alle varie entità del sistema ed in particolare si focalizza su due aspetti: la definizione degli elementi per la comunicazione tra esse e la modellazione degli elementi del dominio. Questi aspetti delineno quindi la struttura del modulo nei due rispettivi package **Presentation** e **Entities**.

Nella comunicazione tra le varie entità è necessario scambiare le informazioni in un formato comune e si è scelto di utilizzare **JSON**. Per poter convertire gli oggetti in questo formato lato mittente, è necessario effettuare un'operazione di serializzazione e allo stesso tempo lato ricevente l'operazione inversa di deserializzazione. Per svolgere questi compiti si è deciso di utilizzare come supporto la libreria **Gson**. All'interno del package **Presentation** sono presenti i **Serializer** e **Deserializer** dei vari oggetti che possono essere inviati nei pacchetti. Gli oggetti **Deserializer** implementano l'interfaccia **JsonDeserializer**, il metodo **deserialize** ed estendono la classe astratta **GeneralDeserializer**, la quale fornisce metodi di utilità, utilizzabili da tutti i **Deserializer**, che permettono di ottenere il valore di un campo di un oggetto in base al suo tipo. Gli oggetti **Serializer**, invece, implementano l'interfaccia **JsonSerializer** ed il metodo **serialize**.

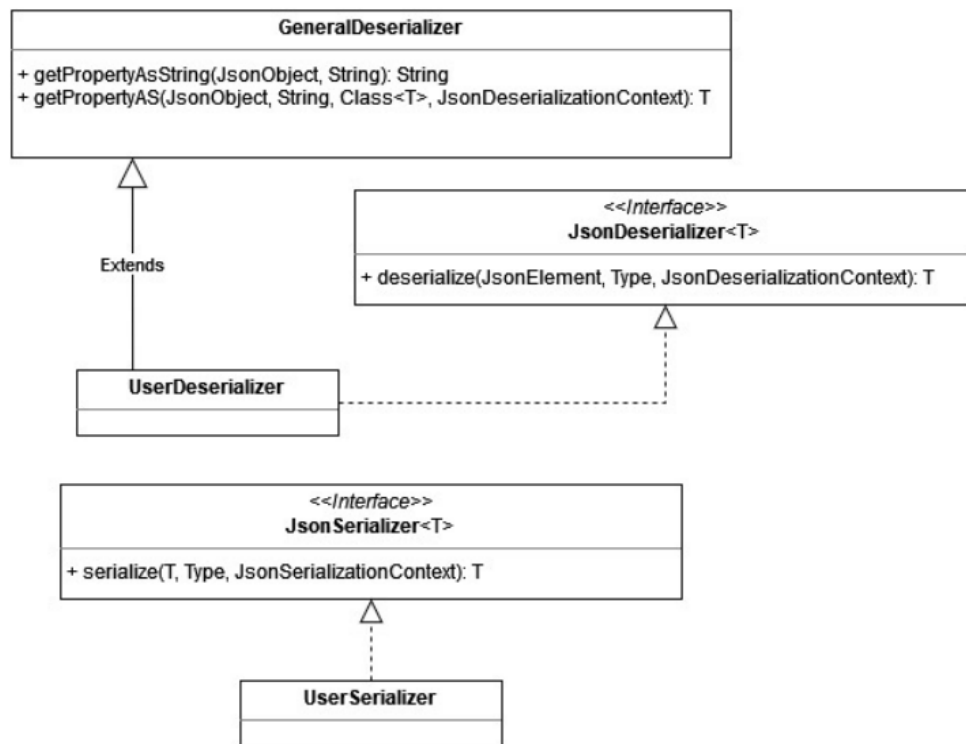


Figura 5: Diagramma delle classi: esempio con Serializer e Deserializer di User

Il package *Entities* comprende tutte le classi relative ai concetti di base del dominio che vengono utilizzate dai vari componenti del sistema mantenendo così una definizione omogenea ed evitando duplicazioni. Queste classi sono di fatto dei *POJO* (Plain Old Java Object), in quanto presentano solamente metodi *getter* e *setter* per i campi definiti al loro interno.

3.1.5 Struttura del modulo serverBackend

Il server web progettato ha il compito principale di interagire con un database non relazionale per effettuare le operazioni richieste, oltre a dover gestire le diverse richieste dei Client.

È stato scelto di realizzare uno **stateless Server**: questo permette che il Server non debba mantenere salvato in memoria alcuno stato dell'applicazione, come accade per le soluzioni **stateful**; di contro, però, ogni richiesta effettuata dai Client dovrà contenere un quantitativo di informazioni maggiore.

I vantaggi di questo approccio sono: la semplicità, in quanto la progettazione del Server risulta meno complicata rispetto ad una soluzione *stateful*; visibilità, non è necessario tenere traccia delle transazioni precedenti; scalabilità, il Server non deve preoccuparsi dell'allocazione di memoria o del suo svuotamento. Lo

svantaggio principale di questa soluzione è che Client e Server avranno un numero maggiore di interazioni con possibile trasmissione di dati già inviati in precedenza.

All'interno del modulo *serverBackend* vengono definite le *API RESTful*, che il servizio espone al Client.

La dichiarazione delle API secondo la specifica **OpenAPI**, realizzata con Swagger è disponibile a questo link.

Il funzionamento dell'applicazione è basato sul meccanismo di *Remote Procedure Call (RPC)*, in cui i Client inviano una richiesta al Server e si mettono in attesa di risposta. Le richieste sono indirizzate ad un URL e devono specificare un *metodo*, cioè quale operazione deve essere eseguita sul Server; le risposte, invece, restituiscono uno **status code** che descrive l'esito della richiesta: questo codice varia in base al protocollo utilizzato, in questo caso è l'HTTP. Questo protocollo definisce metodi per effettuare operazioni *CRUD* (Create, Read, Update and Delete), che sono:

- **POST**: consente di effettuare l'operazione Create;
- **GET**: richiama l'operazione Read;
- **PUT**: effettua l'operazione di Update/Replace;
- **PATCH**: effettua l'operazione Update/Modify;
- **DELETE**: permette di eseguire l'operazione Delete

Il Server permetterà di eseguire queste operazioni sulle diverse entità che popolano il database esponendo delle rotte, tramite cui si possono richiamare i vari metodi. Perciò, una rotta non è niente più di una possibile operazione che il servizio mette a disposizione dei Client per manipolare le risorse: difatti, l'API Web del servizio si compone nella dichiarazione delle sue rotte. Per descrivere formalmente l'API del Server è stato scelto di utilizzare lo standard OpenAPI ed il tool *Swagger*.

Anche intestazioni e parametri sono importanti nei metodi HTTP all'interno di una richiesta **HTTP API RESTful**, perché forniscono informazioni per identificare la richiesta, quali metadati, autorizzazione, Uniform Resource Identifier (URI), caching, cookie e altro. Esistono intestazioni della richiesta e intestazioni della risposta, ognuna con le proprie informazioni di connessione HTTP e codici di stato.

All'interno del modulo è presente il package **model**, il quale determina le interfacce delle diverse entità che popolano il database: in particolare, il subpackage **db** definisce, seguendo gli standard della libreria *mongoose*, i **Types**, **Schema** e **Projection** delle entità. Le definizioni di questi modelli saranno utilizzate per schematizzare i dati trasmessi per eseguire le operazioni sul database.

In questo modulo i due package principali sono quello relativo al **controller**, che specifica le operazioni possibili da richiamare sul database, ed il package **api**,

che espone l'API ed i metodi che si possono richiamare, definendo inoltre le rotte che i Client dovranno specificare per l'invio delle richieste; all'interno dei package è stata creata una classe per ogni entità presente nel database.

Nel primo package `api` troviamo la definizione dei metodi DELETE, GET, POST e PUT, i quali, una volta richiamati, invocheranno i metodi della relativa classe `controller` per l'esecuzione delle operazioni.

Il `controller` si occupa, inoltre, di ritornare la risposta al Client o eventualmente gestire le eccezioni che possono essersi verificate durante l'esecuzione dell'operazione, impostando la risposta con il relativo codice di errore.

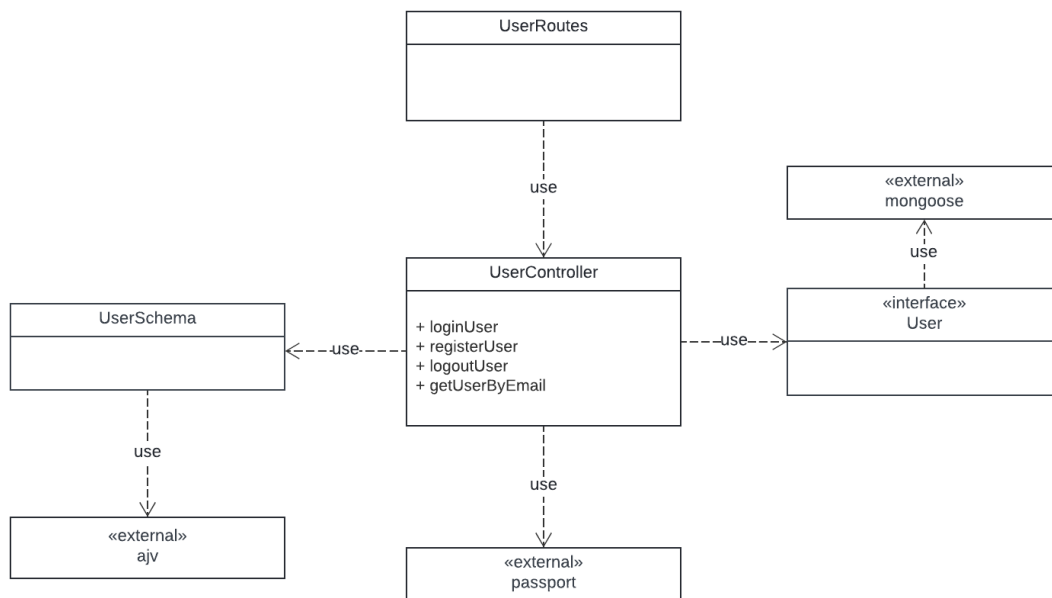


Figura 6: Diagramma delle classi: `UserRoutes`(API) e `UserController`

Abbiamo inoltre aggiunto un validatore, implementato nel package `validator`, per validare le richieste inviate all'API, che sfrutta la libreria `ajv`: quando giunge una richiesta al Server, prima di eseguire le operazioni sul Server, il validatore si occupa di confrontare i dati trasmessi con il modello predefinito dell'entità coinvolta nella richiesta. Se il corpo della richiesta corrisponde allo schema preimpostato, si procede con l'operazione selezionata, altrimenti la richiesta viene terminata con un codice di errore (400 - *Bad Request*).

3.1.6 Struttura modulo `mobileApp`

Il Client Mobile consiste in un'applicazione per dispositivi Android dotati di una versione del sistema superiore alla 9.0 (Android SDK 28 - Android Pie). Nella

realizzazione del Client si è cercato di garantire estendibilità per eventuali aggiunte future tramite l'adozione di scelte di design atte a raggiungere questo scopo.

Un'applicazione Android è formata da diversi componenti: Activity, Fragment, Service, Content Provider e Broadcast Receiver. Spesso gli utenti interagiscono in un breve tempo con più app, perciò il sistema operativo potrebbe distruggere i processi inutilizzati per liberare risorse e fare spazio a quelli nuovi. È possibile che i componenti dell'applicazione vengano avviati in ordine diverso o che possano essere distrutti dal sistema operativo in qualsiasi momento: perciò è buona norma non mantenere i dati e lo stato dell'applicazione all'interno dei componenti dell'applicazione. In base a quanto detto, bisogna progettare lo scheletro dell'applicazione in modo tale da separare gli aspetti dell'interfaccia dalla logica e gestione dei dati dell'applicazione.

Android Developers consiglia di separare i concetti in due livelli principali[2]: un primo livello **UI Layer** di interfaccia utente ed il livello **Data Layer** per i dati, come raffigurato nell'immagine 7. Il livello dell'interfaccia utente ha il compito di visualizzare a schermo i dati dell'applicazione e gestire l'interazione con l'utente; questo livello comprende:

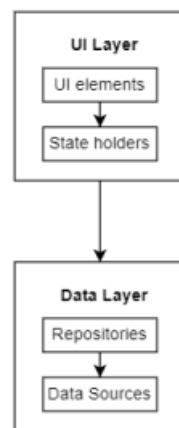


Figura 7: Livelli principali dell'applicazione mobile

- **UI elements**, tutto ciò che l'utente visualizza e con cui può interagire;
- **State holders**, mantengono stato e logica correlati all'interfaccia utente e richiedono il caricamento dei dati necessari per la visualizzazione.

Al contrario, il livello dei dati memorizza le informazioni dell'applicazione ed è composto da un insieme di *repositories* i quali, a loro volta, possono comprendere dei *data sources*:

- **Repositories**, hanno il compito di estrarre i dati dai *data sources* per poi esporli al resto dell'applicazione;

- **Data Sources**, rappresentano l'origine dei dati: questi ultimi possono essere in remoto, cioè richiesti in rete, oppure in locale, ad esempio memorizzati in un database locale. Queste classi sono utilizzate solo dei *repositories* per rendere i moduli indipendenti.

L'architettura dell'applicazione Mobile segue gli aspetti descritti in precedenza tramite l'uso del pattern architetturale **MVVM**, ovvero *Model-View-ViewModel*. Questa architettura è una variante del noto pattern MVC (Model-View-Controller): difatti è possibile separare gli aspetti dell'interfaccia grafica dalla logica del dominio, in aggiunta è stato ideato per semplificare la programmazione basata su eventi delle interfacce utente. In tal modo, l'interfaccia non include più la parte relativa alla logica e, grazie all'indipendenza delle varie componenti, si può modificare l'interfaccia stessa senza modificarne il comportamento. Le componenti principali di questa architettura sono:

- **Model**, rappresenta il dominio dell'applicazione nonché il punto di accesso ai dati, gestendo la logica dell'applicazione e dell'elaborazione dei dati. Il livello dei dati dei *repositories* e dei *data sources* si colloca qui;
- **View**, comprende gli *UI Elements*, nonché l'interfaccia utente con cui l'utente interagisce, si occupa della visualizzazione dei dati e dell'interazione con l'utente;
- **ViewModel**, rappresentato dagli *State holders*, i quali fungono da centro di comunicazione tra Model e View, interagendo con il primo per richiedere i dati necessari alla View.

In sintesi, la struttura utilizzata per la creazione dell'applicazione è raffigurata in figura 8.

L'applicazione per dispositivi mobili si compone di diverse pagine, risultando in più interfacce, ognuna di esse rappresentata da un **layout** definito in linguaggio XML e da una classe **UI Controller** associata. Il layout definisce il design dell'interfaccia, delineando tutti gli elementi View presenti nella pagina, mentre la UI Controller è rappresentata dalle Activity o dai Fragment. Un'Activity è l'*entry point* dell'interazione tra applicazione ed utente, è il componente associato all'interfaccia che permette di gestire gli elementi della pagina. Un'Activity può includere più interfacce, dette **Fragment**. Activity e Fragment hanno un proprio ciclo di vita, ed è sempre possibile sapere in che fase di questo ciclo di vita ci si trova: se il sistema ha creato l'attività, se la sta arrestando o riprendendo o se si sta distruggendo l'intero processo in cui l'attività risiede.

I **ViewModel** gestiscono i dati relativi all'interfaccia utente consapevolmente rispetto al ciclo di vita dell'applicazione. I ViewModel contengono i **LiveData**, cioè classi che possono essere *osservate* quando i dati contenuti vengono modificati: in questo modo, i componenti dell'interfaccia utente vengono notificati dell'avvenuta

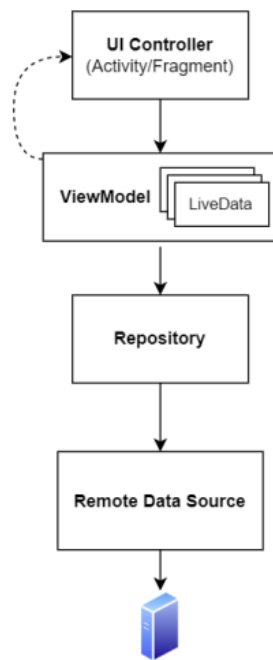


Figura 8: Struttura applicazione mobile

modifica e possono di conseguenza aggiornarsi. I LiveData gestiscono gli eventi in autonomia poichè sono consapevoli delle modifiche dello stato del ciclo di vita durante l'osservazione.

In seguito troviamo le classi **Repository**, che permettono di astrarre l'accesso ai dati: questo significa che i ViewModel si procurano i dati necessari alle interfacce tramite i Repository. Il ruolo dei Repository consiste nel richiedere i dati alle varie sorgenti, in particolare queste richieste avvengono in remoto, cioè in rete attraverso le rotte esposte dal Server. Ogni sorgente di dati (modificabile) è rappresentata con una entità **Remote Data Source**.

Per comprendere correttamente questi concetti e la struttura, procediamo analizzando il diagramma delle classi per la gestione delle piante presenti in serra, raffigurato in figura 9.

La classe **GalleryFragment** estende dal componente **Fragment** di Android e rappresenta la View della pagina che permette di visualizzare le piante presenti in un determinato settore della serra, selezionato in precedenza.

Per gestire una lista di elementi in modo semplice ed efficiente si è utilizzato il componente **RecyclerView** di Android: difatti, è sufficiente definire l'aspetto di ogni elemento e la libreria RecyclerView crea dinamicamente gli elementi alla necessità. Per la gestione del layout ed il comportamento specifico di ogni pianta dell'elenco è stata creata una classe **PlantAdapter**; questa classe estende dalla classe **Adapter** base di Android prendendo in input un **PlantViewHolder** che contiene

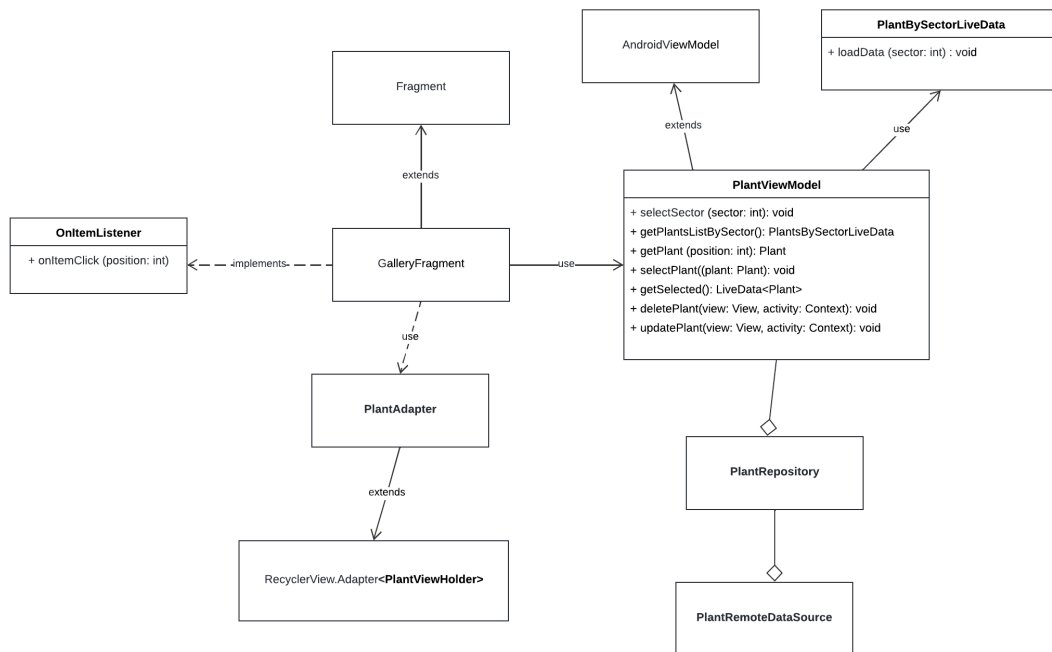


Figura 9: Diagramma delle classi: gestione lista piante

i riferimenti ai componenti della View. La classe `GalleryFragment` implementa anche l'interfaccia `OnItemClickListener` e presenta il metodo `onItemClickListener(position: int)` che permette di accedere alla pagina di modifica della pianta selezionata.

Per gestire la logica della View ed i dati relativi alle piante si è utilizzata la classe `PlantViewModel`: quest'ultima conserva il riferimento alla pianta selezionata e si occupa di richiedere i dati necessari alla View richiamando la classe `PlantRepository`. Tale classe, come già spiegato, si interpone tra le diverse origini dati ed il resto dell'applicazione. Perciò, per la gestione delle piante, la classe `PlantRepository` ha un'unica origine dati:

- `PlantRemoteDataSource`, si occupa di effettuare richieste alla rete per ottenere i dati relativi alle piante.

3.1.7 Struttura del modulo sensors

Questa sezione del progetto simula un sensore intelligente che misura i valori relativi a temperatura, umidità, pressione atmosferica e quantità di CO_2 in un certo ambiente.

Il modulo si compone di un piccolo applicativo definito da due classi:

- `ControlUnit`: rappresenta il concetto di centralina e simula le rilevazioni dei sensori.

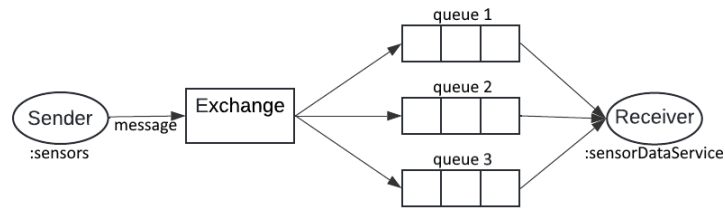


Figura 10: Schema dello scambio di messaggi tramite RabbitMQ

- **Sender**: che è progettato come *publisher* per RabbitMQ

La libreria **RabbitMQ** è un broker di messaggistica, che implementa il protocollo *Advanced Message Queuing Protocol* (AMQP). Lo scopo della libreria è quello di fornire una piattaforma in cui si ha la possibilità di inviare e ricevere messaggi. Il compito del broker è quindi quello di gestirne la coda dei messaggi in ingresso, garantendo memoria affidabile e la consegna dei messaggi. In questo modo è possibile gestire in modo asincrono lo scambio di messaggi. Il protocollo di comunicazione AMQP è composto dalle seguenti entità:

- **Publisher**, cioè l'inviante del messaggio;
- **Consumer**, il destinatario del messaggio
- **Queue**, la coda in cui i messaggi vengono conservati prima di essere consumati;
- **Exchange**, crea la comunicazione con la coda attraverso un legame detto *binding*.

Il paradigma di comunicazione utilizzato è quindi di tipo **publish-subscribe** in cui questo modulo costituisce il *publisher*. Nello specifico si è scelto di sfruttare la capacità di instradamento offerta dal protocollo definendo varie code. L'idea è quella per cui ogni utente del sistema abbia associata una coda in cui la propria centralina pubblica le rilevazioni, nella figura 10 è schematizzato come avviene lo scambio dei messaggi.

Per segnalare al consumatore dei messaggi che vi è una nuova centralina che pubblica i dati è stata riservata una particolare coda in cui periodicamente ogni produttore segnala la sua presenza indicando a quale indirizzo scriverà. I messaggi vengono inviati su quest'ultima periodicamente come meccanismo di **fault tolerance**, ovvero per far sì che nel caso in cui il ricevitore abbia dei problemi e perde la registrazione di una coda possa recuperare l'informazione alla notifica successiva.

3.1.8 Struttura modulo `sensorDataService`

Il modulo `sensorDataService` ha la responsabilità di raccogliere i dati prodotti dai sensori IoT dei vari utenti e inviarli al web server per la registrazione nel database, fungendo perciò da middleware. Il suo scopo è quindi quello di essere un *subscriber* per RabbitMQ.

Si è scelto di progettare come modulo autonomo, perché rappresenta un compito a se stante che se integrato all'interno di altri componenti andrebbe a renderli inutilmente più complessi. Inoltre in questo modo esso può essere distribuito.

La classe principale è `Receiver` che modella il concetto di un tipico *subscriber* asincrono, in cui a fronte di un evento, in questo caso un nuovo messaggio, esegue una funzione di *callback*. Seguendo quindi il meccanismo progettato per l'invio dei dati e in particolare la registrazione di nuovi produttori, il Receiver rimane quindi in attesa di messaggi che comunicano la registrazione di nuovi produttori, in seguito a questi si chiede al message broker di creare una nuova coda raggiungibile dall'indirizzo indicato nella richiesta dal produttore. Su quest'ultima coda viene poi registrata la callback per la gestione dei messaggi relativi alle rilevazioni dei sensori che registrerà i dati sul server. I messaggi scambiati sulle code sono codificati in formato JSON.

3.1.9 Struttura modulo `irrigationControlUnit`

Questo componente è progettato come un applicativo che gestisce la centralina per l'irrigazione, che in questo caso è simulata. Si tratta di un elemento che verrà istanziato per ogni utente e comunicherà direttamente con l'applicazione mobile. Per realizzare questa comunicazione si è pensato di utilizzare il framework **gRPC**, in quanto esso permette di gestire delle connessioni punto-punto di tipo request-response e ne facilita la progettazione e lo sviluppo (figura 11). Infatti, questo framework si basa sul concetto di definire un servizio attraverso una *Interface Definition Language* (IDL), ovvero un modo di descrivere un servizio indipendentemente dai linguaggi e dalle piattaforme utilizzate.

In questo modulo è stato definito quindi il protocollo per l'interazione con le centraline che in queste comunicazioni saranno i server in ascolto delle richieste provenienti dall'applicazione.

Il protocollo disegnato prevede le seguenti richieste:

- `openIrrigationSector`: per indicare di aprire l'irrigazione in un dato settore;
- `closeIrrigationSector`: per comandare la chiusura di un dato settore;
- `getSectorStatus`: per richiedere lo stato in cui si trovano i vari settori.

Per rispondere a queste esigenze il modulo presenta quindi una classe che rappresenta la centralina stessa e un'altra classe che delinea il servizio esposto.

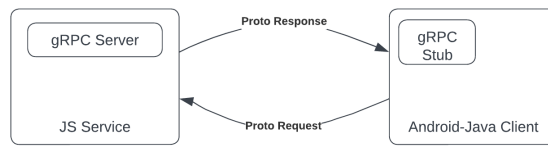


Figura 11: Architettura di gRPC

Per gestire l'accoppiamento tra una centralina e l'applicazione del suo utente proprietario si è pensato di memorizzare nel database tramite il server web l'indirizzo della centralina, che sarà quindi disponibile tramite l'apposita rotta API.

3.2 Comportamento

3.2.1 Comportamento del Server

Il comportamento del Server può essere descritto mediante il diagramma degli stati visibile in figura 12.

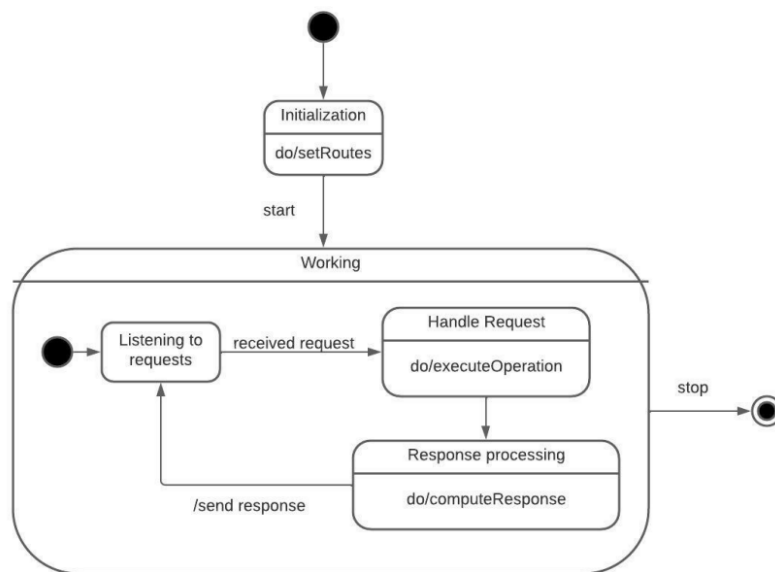


Figura 12: Diagramma degli stati del Server

Visualizzando il diagramma, si nota che, una volta avviato il Server, esso si trova in un primo stato di inizializzazione dove vengono impostate le rotte del servizio; successivamente quando il servizio viene attivato (evento *start*), si passa allo stato **Working**, che si tratta di uno stato composto da cui si può uscire solo quando si verifica l'evento *stop*, che provoca la terminazione del Server.

Sempre in figura 12, quando il Server si trova nello stato **Working**, si mette in ascolto di richieste provenienti dai Client e, quando arriva una nuova richiesta, passa allo stato **Handle Request**, nel quale il Server procede eseguendo l'operazione demandata dal Client. Una volta conclusasi l'esecuzione, si passa allo stato di **Response Processing**, dove il Server elabora la risposta, che può essere di successo o fallimento a seconda dell'esito dell'operazione. Una volta elaborata la risposta, essa viene inviata al Client ed il Server ritorna nello stato **Working**.

Da notare che, quando il Server si trova nello stato **Working**, possono essere gestite più richieste contemporaneamente, anche proveniente da diversi Client: per ogni richiesta pervenuta il comportamento mantenuto dal Server è lo stesso descritto in precedenza.

3.2.2 Comportamento del Client Mobile

Le principali funzioni compiute lato Client vengono innescate direttamente dall'utente: difatti, è l'utente che, mediante lo svolgimento di operazioni sull'applicazione, determinerà i cambiamenti di stato del Client. Inoltre, sempre secondo le operazioni svolte dall'utente, il Client si occupa di inviare le richieste al Server, ad esempio: una volta che l'utente compila correttamente il form di login per effettuare l'accesso all'applicazione, il Client invierà una richiesta al Server per l'autenticazione, quindi è possibile affermare che l'utente sia l'elemento cardine della parte Client-side.

Gli stati principali del Client, i quali possono essere mutati dall'utente stesso tramite un'interazione con il sistema, sono i seguenti:

- **Start:** stato iniziale dell'applicazione, da questo stato l'utente può effettuare il login o registrarsi. Se l'utente si registra, dovrà poi effettuare il login per proseguire nella pagina principale del sistema.
- **Main:** questo è lo stato principale dell'applicazione, che consiste in un menù da cui l'utente può decidere se creare una nuova pianta, visualizzare quelle in galleria, controllare le statistiche della serra o gestire l'impianto di irrigazione.
- **AddPlant:** se l'utente decide di aggiungere una nuova pianta alla serra, il Client entra nello stato addPlant; per ultimare la creazione effettiva della nuova pianta il Client invia una richiesta al Server e si mette in attesa di risposta, che se è di successo tornerà alla pagina principale; se invece la risposta contiene un codice d'errore si resterà nello stato corrente mostrando il messaggio d'errore all'utente.
- **PlantGallery:** il Client entra in questo stato quando l'utente decide di visualizzare le piante presenti in serra, si tratta di fatto di uno stato composto i cui dettagli sono visualizzabili nella figura 14.

- **StatsPage**: nel momento in cui l'utente decide di visualizzare le statistiche relative all'ambiente della serra, il Client entrerà nello stato StatsPage.
- **WateringPage**: se l'utente desidera gestire l'impianto di irrigazione da remoto, il Client giungerà nello stato WateringPage.

È giusto specificare che, una volta effettuato l'accesso, l'utente può navigare in tutte le pagine e compiere numerose azioni; allo stesso tempo, da qualsiasi stato in cui si l'utente trova, potrà tornare allo stato principale Main, ogni qual volta lo desideri. Infine, come visibile in figura 13, l'utente può continuare a navigare fra le rispettive pagine dell'applicativo finchè non decide di terminare l'applicazione.

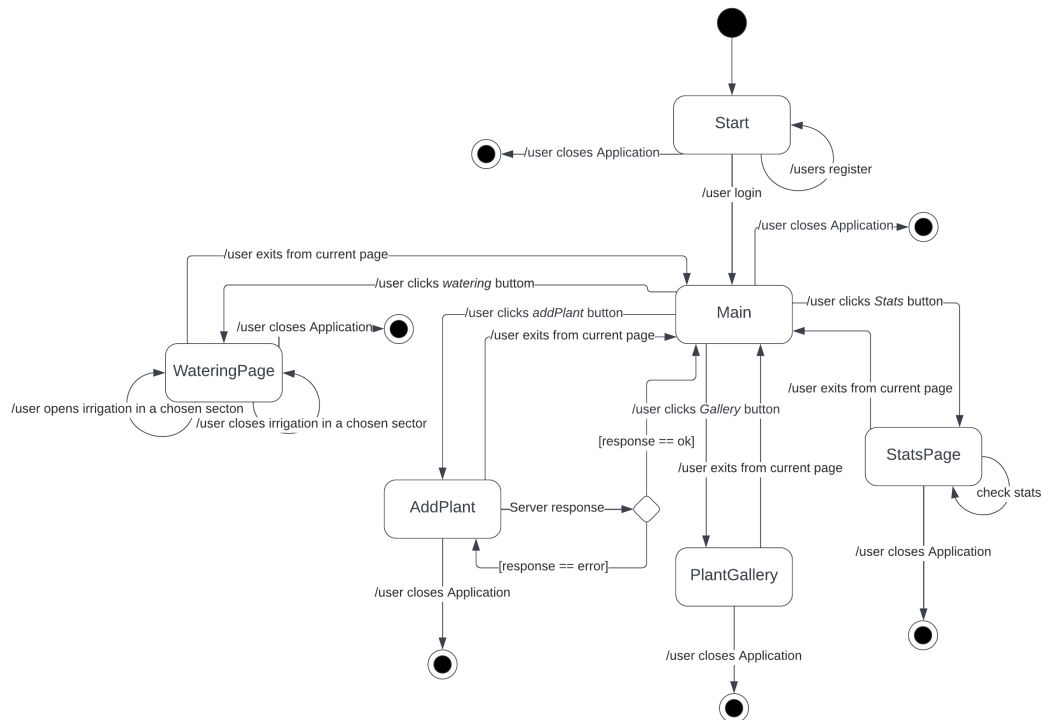


Figura 13: Diagramma degli stati: comportamento del Client

La figura 14 rappresenta la composizione dello stato PlantGallery, che mostra quali sono i possibili stati che il Client può raggiungere, quando un utente naviga all'interno della galleria.

Quando l'utente seleziona un particolare settore della serra, il Client entra nello stato **PlantList**, all'interno di questo stato viene mostrata la lista delle piante con le rispettive informazioni; per ogni pianta è possibile accedere alla scheda di modifica o eliminazione. Se si conferma di voler aggiornare le informazioni

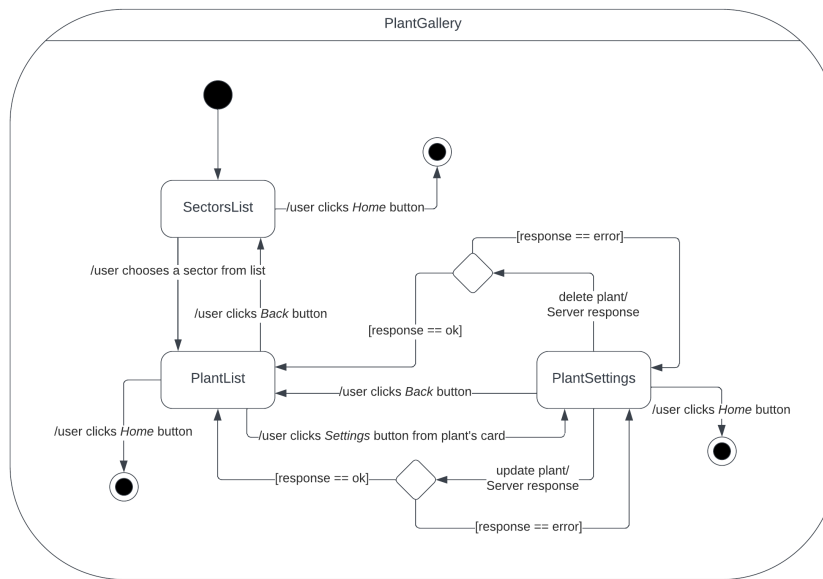


Figura 14: Diagramma degli stati: stato composto Plant Gallery

relative ad una pianta e il processo va a buon fine, si ritornerà all'elenco delle piante, altrimenti verrà visualizzato un messaggio con il codice di errore; se invece l'utente desidera eliminare la pianta, una volta terminata l'operazione si tornerà allo stato principale Main.

3.3 Interazione

Abbiamo definito che le entità del dominio sono essenzialmente quattro: Server, Client Mobile, database MongoDB e dispositivi IoT. Per comprendere come queste entità interagiscano tra loro è sufficiente guardare la figura 15, rappresentante un diagramma di sequenza, che mostra le interazioni tra le diverse entità durante il processo di registrazione di un nuovo utente.

Come si può osservare l'interazione iniziale parte dal Client, il quale invia una richiesta HTTP POST al Server, chiedendogli di creare un nuovo utente; il Server, prima di effettuare l'operazione di inserimento vera e propria, controlla che i dati inviati siano corretti tramite un validatore: nel caso in esame, viene dapprima controllato che siano stati inviate due stringhe per il nome ed il cognome dell'utente, una mail ed un'altra stringa per la password, poi si procede verificando che l'utente non sia già presente sul database. Se l'utente risulta essere già registrato, al Client giunge una risposta con un messaggio d'errore; in caso contrario, si procede all'inserimento del nuovo utente dove, sostanzialmente, il Server richiede al database di effettuare una query di aggiunta, con i dati passati dal Client.

Se l'operazione termina senza errori, il Server invia un messaggio di successo al Client (code 200: **Success**) senza restituire alcun dato.

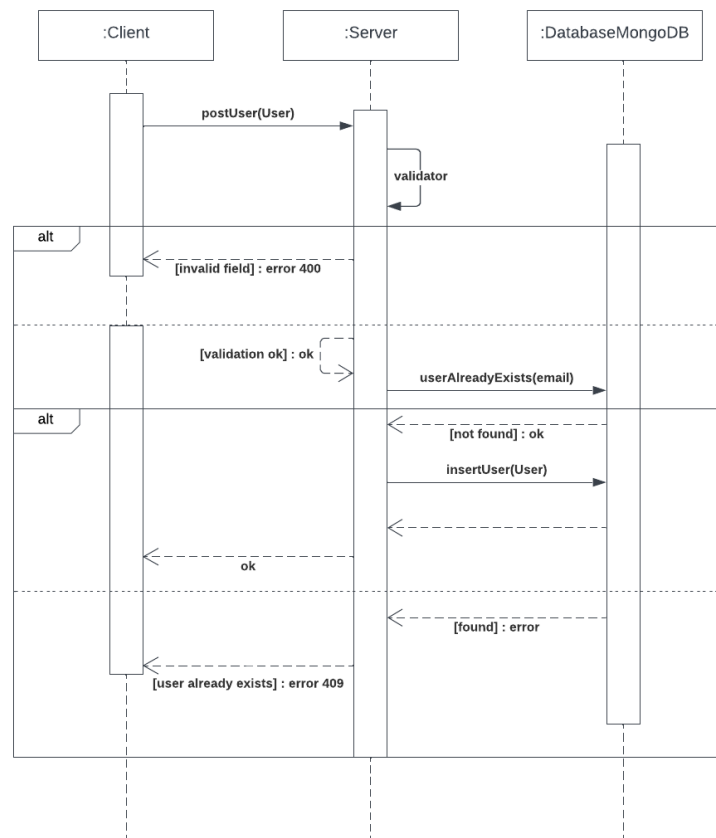


Figura 15: Diagramma di sequenza: interazione tra le entità per la registrazione di un utente

Se invece si vuole osservare un esempio di interazione dove il Server ritorna dei dati al Client, si può visionare la figura 16. Come si può vedere dal diagramma di sequenza, l'interazione viene sempre avviata dal Client, il quale, in questo caso, richiede al Server di inserire una nuova pianta, sempre tramite l'utilizzo del metodo **HTTP POST**.

Come nel caso precedente, il Server, prima di effettuare l'inserimento, verifica che i dati inviati siano compatibili con i tipi dichiarati per l'entità **Plant** del database, inoltre verifica che la richiesta sia stata effettuata da un utente loggato. Se i dati allegati non sono corretti, viene restituito il codice di errore **400 - Bad Request**, se invece l'utente non è registrato viene ritornato il codice di errore **401 - User not Authenticated**; se i controlli vengono superati, il Server effettua l'operazione di inserimento, richiedendo al database di eseguire un'apposita query.

Al termine dell'operazione di inserimento, il Server recupera i dati della pianta appena aggiunta, inserita automaticamente dal database, e li restituisce in risposta all'utente, indicando che l'operazione ha avuto successo.

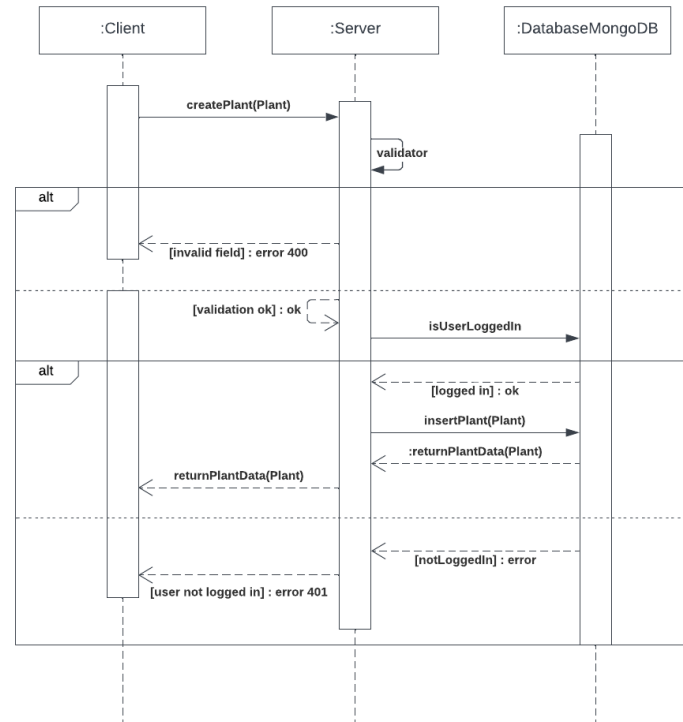


Figura 16: Diagramma di sequenza: interazione tra le diverse entità per l'inserimento di una pianta

In base agli esempi discussi si può concludere che:

- l'interazione nasce sempre dal Client che richiede di eseguire una specifica operazione: in realtà esistono due eccezioni a questa osservazione, che interessano i dispositivi IoT. Difatti, il sensore per la rilevazione dei valori ambientali manda richieste HTTP POST regolarmente al Server, e anche la centralina dell'irrigazione da remoto, quando viene accesa, comunica il suo indirizzo IP al Server, in modo che possa essere associata al relativo utente. Queste operazioni indicate sono automatizzate, in quanto le informazioni inviate sono preimpostate e l'utente non interagisce con esse, di conseguenza, non è necessario effettuare alcun controllo sulla correttezza dei dati inviati,
- il Server, prima di effettuare l'operazione richiesta, effettua altre operazioni preparative in modo che i dati che vengono inseriti siano consistenti e non siano presenti ripetizioni, oltre a verificare la validità dell'operazione richiesta,

- l'interazione si conclude con il Server che invia una risposta di successo o fallimento al Client per l'operazione richiesta

Una volta che il Client ha ricevuto una risposta da parte del Server, ha la facoltà di scegliere se effettuare nuove richieste interrogando il Server, oppure no (per visualizzare le diverse operazioni effettuabili si rimanda alla sezione 4.2).

4 Dettagli implementativi

All'interno di questo capitolo saranno discusse le tecnologie utilizzate ed alcuni dettagli implementativi di maggior rilievo delle diverse entità realizzate, cercando di motivare, ove necessario, la motivazione dietro ad alcune scelte effettuate.

4.1 Tecnologie utilizzate

MongoDB MongoDB un *DataBase Management System* non relazionale, distribuito con licenza *Server Side Public License* (SSPL). Esso è classificato come database di tipo NoSQL, allontanandosi dalla struttura tradizionale basata su tabelle dei database relazionali (come MySQL), favorendo documenti in stile JSON con schema dinamico (formato *BSON*), per rendere l'integrazione dei dati di alcuni tipi di applicazioni più semplice e rapida.

Le caratteristiche principali sono: query ad hoc, supportando ricerche per campi, intervalli ed espressioni regolari; indicizzazione di qualsiasi campo; alta disponibilità e aumento del carico gestito attraverso i replica set.

Nel progetto, MongoDB è stato utilizzato per gestire il database di riferimento dell'applicazione.

Gradle Gradle è un *build automation tool open-source*, JVM-base, che permette di automatizzare la costruzione dei progetti Java; esso semplifica anche la gestione delle librerie esterne, le quali possono essere utilizzate all'interno del progetto, senza dover includere i `.jar`, ma specificando delle dipendenze da un *repository* remoto. Gradle consente anche di gestire progetti composti da più moduli e delle dipendenze che possono crearsi tra i moduli.

All'interno del progetto, Gradle è stato utilizzato per gestire i diversi moduli.

Gson Gson è una libreria utilizzata per convertire gli oggetti Java nella loro rappresentazione JSON e anche per il percorso inverso, cioè convertire una stringa JSON nell'equivalente oggetto Java. Gli obiettivi di Gson, come riportato dalla documentazione[3] sono: fornire meccanismi semplici da utilizzare e costruttori per convertire gli oggetti Java in JSON e viceversa; consentire di convertire oggetti Java immutabili preesistenti nel loro formato JSON e viceversa; permettere rappresentazioni custom degli oggetti; supportare oggetti arbitrariamente complessi ed, infine, generare un output JSON compatto e leggibile.

Nel progetto, la libreria Gson è stata utilizzata nel package `Presentation` del modulo `common`, per realizzare i `Serializer` e `Deserializer` degli oggetti,

Docker Docker, come riportato dalla documentazione[4], è una piattaforma open-source per lo sviluppo, il rilascio e l'esecuzione di applicazioni. Esso permette di separare le applicazioni dall'infrastruttura dell'host, costruendo degli ambienti di esecuzione virtualizzati e autonomi per poter astrarre problemi di configurazione e gestione di ambienti reali.

Docker consente di racchiudere un'applicazione in un ambiente isolato detto *container*; all'interno dello stesso host si possono eseguire più container isolati. I container sono *lightweight* e ideati per fornire il minimo indispensabile per eseguire l'applicazione contenuta.

All'interno del progetto si è utilizzato Docker come strumento per il deployment del sistema, come descritto nella sezione 6.1.

Mongoose Mongoose è una libreria Javascript che consente di definire i modelli di dati per i database MongoDB, semplificando le operazioni effettuate sul database come interrogazioni, aggiunte, aggiornamenti o cancellazione di dati. Una delle caratteristiche principali di Mongoose è la definizione dello schema: tramite questa libreria, è possibile definire la struttura ed i tipi di dati dei documenti MongoDB, rendendo più facile garantire coerenza e validità dei dati. Inoltre, Mongoose fornisce funzioni di convalidazione, middleware, query ed aggregazione avanzata e supporto delle transazioni.

Il ruolo di questa libreria all'interno del progetto è quindi quello di favorire la comunicazione tra il server web e il database.

Volley Volley è una libreria HTTP fornita da Google che permette la gestione delle funzionalità di networking delle applicazioni Android[5]. Questa libreria si integra facilmente con qualsiasi protocollo e fornisce supporto per stringhe, immagini e JSON. Il suo utilizzo è consigliato per le operazioni *RPC*, utilizzate, ad esempio, per popolare un'interfaccia utente nel client: infatti, permette di gestire le richieste in modo asincrono, gestendo in automatico le richieste di rete e offrendo la possibilità di effettuare connessioni multiple.

Questa libreria è stata utilizzata all'interno del Client Mobile per gestire le richieste al Server.

RabbitMQ RabbitMQ è un *message-oriented middleware* open source, scritto in linguaggio Erlang; in altre parole si può definire un *broker di messaggistica* che si basa sul protocollo *Advanced Message Queuing Protocol* (AMQP)[6]. Il protocollo su cui si basa RabbitMQ consente a dei client di inviare e/o ricevere messaggi tramite l'utilizzo di uno o più *broker*; quest'ultimo riceve messaggi dai *Producer*, li inserisce in una *queue* per direzionarli verso i *Consumer*, i destinatari dei messaggi. Essendo AMQP un protocollo di rete, Broker, Producer e Consumer possono trovarsi su macchine diverse.

I servizi di RabbitMQ sono stati utilizzati, all'interno del progetto, per consentire la comunicazione tra i sensori ambientali ed il middleware che inoltra i dati ricevuti al Server.

gRPC gRPC è un *framework* di Google open-source ad alte prestazioni che evolve il protocollo *RPC* (*Remote Procedure Call*). Il vantaggio di RPC è che permette di astrarre la comunicazione di rete punto-a-punto, la serializzazione e

l'esecuzione tra computer. Tramite un *Interface Definition Language (IDL) cross-platform*, gli sviluppatori definiscono un *service contract* per ciascun microservizio; questi contratti sono scritti sotto forma di file testuale `.proto` e descrivono i metodi, gli input e gli output per ogni servizio. Lo stesso file `.proto` può essere utilizzato per i client e server gRPC costruiti su diverse piattaforme di sviluppo.

Questo framework all'interno del progetto ha il ruolo di mettere in comunicazione gli attuatori per l'irrigazione con il Client Mobile.

4.2 Dettagli implementativi Server

Il server web è stato implementato in Typescript, e viene eseguito su Node.js. Si è scelto di utilizzare Typescript per poter beneficiare dei vantaggi di un linguaggio di programmazione tipato, ovvero la possibilità di individuare molti più errori mentre si scrive codice e non solo nel momento in cui viene eseguito.

4.2.1 Gestione validazione richieste HTTP

Uno degli errori più frequenti che si verificano durante le operazioni con il database è quello con codice 400, che corrisponde ad una *Bad Request*: questo avviene quando il corpo JSON della richiesta presenta dati in un formato errato rispetto a come è stata dichiarata l'entità che si intende manipolare.

Questo tipo di errore può essere individuato prima che il Server comunichi con il database e che riceva, quindi, una risposta con errore.

Il procedimento che permette di verificare la correttezza del corpo della richiesta è detta *validazione*; in particolare, nel progetto si è fatto uso di una libreria specifica per la validazione, `ajv`. Durante questa fase si verifica che la richiesta soddisfi le specifiche dell'API. Per fare ciò si definisce il **JSON Type Definition** e il relativo **JSON Schema Type**. In cui nel primo caso si indica la struttura dell'oggetto indicando i campi e il tipo, mentre nel secondo si vanno a definire maggiori vincoli sulla struttura come ad esempio limitazioni sui valori che possono essere assunti o sul formato dei dati.

Per esempio, per validare i dati ricevuti durante una richiesta di login, dove è previsto che l'oggetto trasmesso abbia solo due proprietà, *username* e *password*, entrambe *required*, viene usato lo schema descritto nel listato 1. In esso si può vedere la definizione del JSON Schema Type in cui sono stati impostati anche i vincoli sul formato dei dati.

Quando il Server web riceve una richiesta di login, il suo *body* viene confrontato con lo schema e, se le proprietà presenti sono corrette, si prosegue effettuando l'operazione sul database, altrimenti il Server termina la richiesta con un errore.

```
export const UserLoginSchema: JSONSchemaType<UserLogin> = {
  required: ["username", "password"],
  type: "object",
  additionalProperties: false,
```

```

    properties:{
      username: {
        type: "string",
        format: "email"
      },
      password: {
        type: "string",
        format: "password",
      }
    }
  };

```

Listing 1: JSONSchema per il login utente

La funzione di validazione è `validateRequest` (listato 2). Si tratta di una funzione generica nel tipo di richiesta da validare che prende come input uno schema generico e i dati da validare, in output restituisce un valore *booleano* che indica se i dati rispettano lo schema.

```

export function validateRequest<T>(<
  schema: JSONSchemaType<T>,
  data: unknown
): data is T {
  if (ajv.validate(schema, data)) {
    return true;
  } else {
    console.warn("Validation failed", ajv.errorsText(), data);
    return false;
  }
}

```

Listing 2: metodo validateRequest per la validazione

Infine, per effettuare alcune richieste al Server è necessario aver effettuato l'accesso all'applicazione: ad esempio, tutte le *routes* relative alla gestione delle piante, hanno un pre-controllo che blocca la richiesta nel caso in cui essa non sia parte di una sessione precedentemente creata con il login.

4.3 Dettagli implementativi Client Mobile

Procedendo per questa sezione si potranno osservare gli aspetti rilevanti della realizzazione dell'applicazione Mobile, con la descrizione delle scelte ed i dettagli implementativi. Inoltre, le classi e le interfacce realizzate, sono state arricchite con apposita documentazione Javadoc.

4.3.1 Gestione delle Activity e Fragment

L'applicazione è composta da tre Activity principali:

- **LoadingActivity** *starting application*, si occupa dell'avvio dell'applicazione,
- **LoginRegisterActivity** gestisce il processo di registrazione ed accesso dell'utente, comprendendo due Fragment: **RegisterFragment** che consente ad un nuovo utente di registrarsi e **LoginFragment** che si occupa dell'accesso all'applicazione di un utente già registrato,
- **MainActivity** coordina tutte le interfacce principali dell'applicazione come la visualizzazione della galleria piante, delle statistiche della serra e dell'irrigazione.

Si è deciso di adottare un approccio orientato ad un uso maggiore dei Fragment piuttosto che delle Activity siccome, durante la navigazione all'interno dell'applicazione, bisogna gestire un certo numero di dati che possono anche essere scambiati tra le pagine. Per tale ragione, si è deciso di utilizzare i ViewModel per condividere i dati tra i vari Fragment della stessa attività, di cui si tratterà nella sezione successiva.

4.3.2 Gestione e condivisione dei dati tra Fragment

Come già ribadito, Android gestisce il ciclo di vita di Activity e Fragment in base alle azioni svolte dall'utente o ad eventi non controllati dall'applicazione, per questo motivo il sistema operativo può eventualmente decidere se distruggere o ricreare il *controller UI*. Se però il sistema decide di distruggere il controller dell'interfaccia utente, tutti i dati visualizzati all'interno dell'interfaccia saranno perduti e la nuova Activity dovrà recuperarne tutti i dati. In caso di dati semplici il recupero può essere effettuato con il metodo `onSavedInstanceState()`, ma in caso di dati di grandi dimensioni e che necessitano di caricamento si consiglia di gestirne la ripresa tramite un componente **ViewModel**.

Architecture Components fornisce la classe di supporto **ViewModel** per il controller dell'interfaccia: gli oggetti **ViewModel** sono conservati automaticamente in modo tale che i dati possano essere disponibili alla ripresa o nel passaggio all'Activity o Fragment successivo.

Nel listato 3 viene mostrato un esempio di creazione di un **ViewModel**, in particolare **PlantViewModel** che estende dalla classe predefinita **AndroidViewModel**, la quale richiede il contesto dell'applicazione nel suo costruttore.

```
public class PlantViewModel extends AndroidViewModel {

    private int selectedSector;
    private PlantRepository plantRepository;
    private PlantBySectorLiveData plantBySectorLiveData;
    private MutableLiveData<Plant> currentPlant =
        new MutableLiveData<>();

    public PlantViewModel(@NonNull Application application) {
```

```

        super(application);
    }

    public PlantBySectorLiveData getPlantsListBySector() {
        this.plantBySectorLiveData = new PlantBySectorLiveData
            (getApplication(), this.selectedSector);
        return this.plantBySectorLiveData;
    }

    public LiveData<Plant> getSelected() {
        return this.currentPlant;
    }
    //...
}

```

Listing 3: Esempio di implementazione del ViewModel

Il ViewModel comprende degli oggetti specifici, dei contenitori di dati detti **LiveData**, oppure **MutableLiveData**, dei contenitori di dati che possono subire modifiche. Il ViewModel può contenere anche dei **Repository** da cui richiedere il caricamento dei dati.

Gli oggetti **LiveData** e **MutableLiveData** possono essere osservati dai vari controller, attraverso la loro registrazione come osservatori: in questo modo, ogni qualvolta la lista subisce modifiche, il controller viene notificato e per aggiornare l'interfaccia.

Per fare ciò, nelle Activity o nei Fragment interessati, attraverso il **ViewModel Provider** è possibile richiedere l'istanza del ViewModel se è già stato creato da qualche controller, altrimenti farne richiesta per una nuova.

4.3.3 Gestione invio delle richieste con Volley

Gli oggetti **Repository** presenti nel ViewModel rappresentano il punto di accesso ai dati; queste classi hanno al loro interno una o più **DataSource** tramite cui richiedere i dati e, nel nostro caso, la richiesta dei dati viene effettuata in rete.

Le richieste avvengono tramite la libreria **Volley**, sviluppata da Google per la gestione di richieste HTTP: **Volley** si integra facilmente con qualsiasi protocollo e consente di inviare stringhe, immagini o JSON. Per utilizzare **Volley** si crea una **RequestQueue** alla quale vengono passati gli oggetti **Request**:

- **Request** esegue l'analisi delle risposte grezze e, successivamente, Volley ha il compito di inviare la risposta analizzata al thread principale.
- **RequestQueue** gestisce in thread di lavoro per l'esecuzione delle operazioni in rete, la lettura e scrittura nella cache e l'analisi delle risposte.

```

public void login(Credential credential, Response.Listener<String> listener,
    Response.ErrorListener errorListener) {
    final String url = this.baseUrl + "users/login";
    final StringRequest stringRequest = new StringRequest(Request.Method.
        POST,
        url, listener, errorListener) {
        @Override
        public String getBodyContentType() {
            return "application/json; charset=utf-8";
        }

        @Override
        public byte[] getBody() {
            return gson.toJson(credential, Credential.class)
                .getBytes(StandardCharsets.UTF_8);
        }
    };
    stringRequest.setTag(Utilities.OSM_REQUEST_TAG);
    this.requestQueue.add(stringRequest);
}

```

Listing 4: Esempio di richiesta HTTP con Volley

Nel listato 4 è rappresentato un esempio di richiesta con Volley per il login utente: in questo caso, abbiamo una richiesta di stringa rappresentata dall'oggetto `StringRequest` al quale vengono passati l'url ed il metodo della richiesta, due `listener` per la gestione dei dati in caso di successo e per la gestione di eventuali errori. Nel caso raffigurato, nel *body* del pacchetto sono contenute le informazioni dell'utente necessarie per il login, definite sotto forma di oggetto della classe `Credential`. L'entità `Credential` viene convertita in una stringa JSON tramite l'utilizzo di `Gson`.

Volley, purtroppo, non fornisce una rappresentazione diretta di una richiesta mediante `Gson`, ma consente di implementare dei tipi di richieste personalizzati: ciò è possibile estendendo una classe `Request` base. In questo caso, avremo che la nuova classe `GsonRequest<T>` estenderà dalla classe `JsonRequest<T>` come è possibile vedere dal listato 5; successivamente, per completare l'operazione, si deve implementare il metodo astratto `parseNetworkResponse`, il quale prende come argomento la risposta e, tale metodo, in caso di successo deve restituire un oggetto `Response` secondo il tipo della classe, oppure le informazioni da visualizzare in caso di errore.

```

public class GsonRequest<T> extends JsonRequest<T> {

    private final Gson mGson;
    private final Class<T> mClassType;
    private final Map<String, String> mHeaders;
    private final Response.Listener<T> mListener;
}

```

```

    public GsonRequest(int method, String url, Class<T> classType,
        Map<String, String> headers, JSONObject jsonRequest, Response.Listener<T>
        listener, Response.ErrorListener errorListener) {
        super(method, url, (jsonRequest == null) ? null : jsonRequest
            .toString(), listener, errorListener);
        mGson = GsonUtils.createGson();
        mClassType = classType;
        mHeaders = headers;
        mListener = listener;
    }

    @Override
    protected Response<T> parseNetworkResponse(NetworkResponse networkResponse)
    {
        try {
            String json = new String(networkResponse.data, HttpHeaderParser.
                parseCharset(networkResponse.headers));
            return Response.success(mGson.fromJson(json, mClassType),
                HttpHeaderParser.parseCacheHeaders(networkResponse));
        } catch (UnsupportedEncodingException | JsonSyntaxException e) {
            return Response.error(new ParseError(e));
        }
    }
    //...
}

```

Listing 5: Creazione di un tipo di richiesta personalizzato per le richieste Gson

4.4 Dettagli implementativi parte IoT

Di seguito verranno indicati alcuni dettagli implementativi della parte *Internet of Things*, i quali mettono in luce aspetti rilevanti delle tecnologie utilizzate durante la realizzazione.

4.4.1 Utilizzo di RabbitMQ per la comunicazione delle statistiche

Come già riportato in precedenza, nell'implementazione del misuratore delle statistiche dell'ambiente si è fatto uso del *middleware* RabbitMQ per comunicare le rilevazioni dei sensori al servizio di raccolta dei dati e poi al Server. Per ogni utente si è pensato ad un dispositivo in dotazione che quando è in esecuzione invia nuove rilevazioni continuamente. Nello specifico invia due tipi di messaggio uno con i dati dei sensori e un altro messaggio per comunicare che sta producendo dati quest'ultimo funge da *heartbeat*. L'invio di questi è gestito tramite due cicli infiniti affidati a due differenti thread. Infatti all'avvio dell'applicazione dopo aver configurato tutto il necessario, da un lato si avvia un thread con il compito di inviare i messaggi di *heartbeat*. Il processo principale d'altra parte si inserisce anch'esso

in un ciclo infinito per l'invio dei messaggi con le rilevazioni. In entrambi i casi l'attesa tra due cicli è gestita tramite *sleep*.

L'invio dei dati è gestito dalla classe **Sender** che sfrutta gli elementi della libreria RabbitMQ per comunicare con il *broker*. Nello specifico, la libreria prevede che per comunicare i dati al broker è necessario creare un canale, che si appoggia su una connessione, e su questo pubblicare nuove informazioni.

Nel listato 6 possiamo vedere il codice della classe. Si può vedere come nel costruttore si crea la connessione attraverso la classe **ConnectionFactory** a cui vengono passati gli opportuni parametri. Partendo dalla connessione viene costruito il canale. Nel canale è necessario dichiarare l'*Exchange* a cui inviare i messaggi e il suo tipo. In questo caso si va a inizializzare con un nome che sarà comune per tutti definito dalla costante **EXCHANGE_NAME**, come tipologia si è scelto **topic**, in cui i messaggi vengono instradati nelle varie code sulla base di una *routing key*, ovvero una lista di parole separata da punti. Nella scelta delle chiavi si è prestato attenzione affinché fossero parlanti e quindi si possa capire lo scopo del messaggio. In questo caso tutti i misuratori utilizzano la chiave **register.newSensorDataProducer** quando invia il messaggio in cui ricorda che sta inviando dati, mentre quando invia i dati utilizza una chiave diversa per ogni utente, infatti essa prevede un prefisso comune, **sendData.**, a cui segue lo username dell'utente.

```
Sender(String username) throws IOException, TimeoutException {
    this.routingKey = "sendData." + username;
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost(BROKER_HOST);
    factory.setPort(5672);
    factory.setVirtualHost("/");
    factory.setUsername("admin");
    factory.setPassword("admin");
    Connection connection = factory.newConnection();
    channel = connection.createChannel();
    channel.exchangeDeclare(EXCHANGE_NAME, "topic");
}

public void refreshProducerRegistration() throws IOException {
    JsonObject key = new JsonObject();
    key.addProperty("key", routingKey);
    channel.basicPublish(EXCHANGE_NAME, "register.newSensorDataProducer", null,
        key.toString().getBytes(StandardCharsets.UTF_8));
}

public void sendData(String message) throws IOException {
    channel.basicPublish(EXCHANGE_NAME, routingKey, null, message.getBytes(
        StandardCharsets.UTF_8));
    System.out.println(" [x] Sent '" + message + "'");
}
```

Listing 6: Implementazione della classe **Sender**

La ricezione dei messaggi provenienti dai vari produttori è affidata al servizio per la raccolta dei dati il quale presenta la classe `Receiver` che è la controparte di `Sender`. Anche in questo caso viene stabilita la connessione con il *broker*, viene creato il canale e dichiarato l'*Exchange*. Oltre a ciò, si vanno a definire le code, ovvero vengono dichiarate e si effettua l'associazione con la rispettiva *routing key*. A questo punto viene dichiarato il consumatore dei messaggi che vengono inseriti nella coda. In questo caso sono stati definiti due tipologie di consumatore per i due tipi di messaggi previsti. Questi sono stati implementati attraverso le classi `RegisterNewQueueListener` e `SensorDataListener`, le quali implementano l'interfaccia funzionale `DeliverCallback`.

Nel listato 7, abbiamo l'implementazione di `SensorDataListener`, questa è la funzione di callback che viene eseguita quando si riceve la comunicazione di nuovi dati dai sensori. Il comportamento è quello di inviare i dati ricevuti al server web con una richiesta HTTP POST tramite l'API prevista. Possiamo notare che per inviare i dati è necessario conoscere lo username dell'utente e questo viene recuperato tramite la *routing key* del messaggio ricevuto.

```
public class SensorDataListener implements DeliverCallback {
    @Override
    public void handle(String consumerTag, Delivery delivery) {
        String message = new String(delivery.getBody(), StandardCharsets.UTF_8)
        ;
        String username = delivery.getEnvelope().getRoutingKey().replaceFirst("
        [^.]+" , "");
        try {
            HttpClient client = HttpClient.newHttpClient();
            HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create("http://" + WEBSERVER_HOSTNAME + ":5000/
                stats/" + username))
                .header("Content-Type", "application/json")
                .POST(HttpRequest.BodyPublishers.ofString(message))
                .build();
            client.send(request, (response) -> {
                if (response.statusCode() != 200) {
                    System.err.println("API error " + response.statusCode());
                }
                System.out.println("[x] Registered on server " + message);
                return HttpResponse.BodySubscribers.discarding();
            });
        } catch (Exception e) {
            System.err.println("Web server not available");
        }
    }
}
```

Listing 7: Implementazione della classe `SensorDataListener`

5 Autovalutazione/Validazione

Per verificare l'efficacia e correttezza del prodotto realizzato, come detto nella sezione 1.2, si è proceduto testando le diverse componenti del sistema, mediante test sia automatici che manuali. Nelle sezioni seguenti verranno esposti con maggior dettaglio i test effettuati.

Naturalmente, il testing effettuato ha anche lo scopo di verificare il comportamento corretto dell'applicazione e il soddisfacimento dei requisiti, indicati nella sezione 2.

5.1 Testing

5.1.1 Testing del Server

Il testing del Server è stato effettuato sia mediante test automatici con l'ausilio della libreria **Jest** che mediante test manuali.

Occorre, inoltre, far notare che per lo sviluppo non si è adottato un approccio di tipo *Test Driven Development*: l'implementazione dei test è stata effettuata scrivendo prima il codice di produzione e poi la parte relativa di testing; ogni volta che veniva completata la stesura di un metodo, esso veniva verificato mediante il testing.

testEnvConfig Questa non è una vera e propria classe di test, infatti si occupa di specificare quali azioni devono essere eseguite prima e dopo i test: la connessione al db e la sua inizializzazione con alcune entità predefinite; al termine dei test, invece, effettua la pulizia del db ed effettua la disconnessione.

testUserRoutes Questa classe di test si occupa di verificare le operazioni effettuate sugli utenti, come l'inserimento di un nuovo utente o il login, verificando, in caso di dati errati, il rilevamento corretto di un'eccezione.

testPlantRoutes All'interno di questa classe vengono testate le operazioni effettuabili modellando l'entità Pianta, come l'aggiunta, modifica, cancellazione o ricerca mediante determinate caratteristiche. Anche qui viene verificata la rilevazione di errori, tra le quali la mancata autenticazione dell'utente.

testStatsRoutes Mediante questa classe di test vengono testate le rotte per l'inserimento e l'interrogazione di entità del tipo *EnvironmentalData*, che racchiude tutte le proprietà rilevate dal sensore ambientale, quindi temperatura, pressione atmosferica, umidità e quantità di CO₂. Il test verifica sia il corretto comportamento delle operazioni effettuate che il rilevamento di eccezioni.

testActuatorsRoutes Il comportamento degli attuatori per la centralina idraulica viene verificato in questa classe di test, che comprende solo i test per i metodi

POST e GET, in quanto le rotte a disposizione permettono di effettuare solo queste operazioni. Anche in questo caso il test verifica il corretto comportamento delle operazioni e in caso i dati passati non siano corretti, si aspetta il corretto lancio di un'eccezione.

Test manuali Per testare il modulo `serverBackend`, il quale si occupa di gestire le diverse richieste provenienti dai Client, sono stati effettuati test manuali sfruttando il supporto offerto dal tool **SwaggerHub**. In particolare, grazie a **SwaggerHub**, dopo aver definito l'API del servizio, si è potuto usufruire di un'interfaccia grafica per visionare le operazioni richiedibili al Server e per ciascuna gli input necessari e le risposte possibili sia in caso di successo che di errore.

È stato utilizzato anche il tool **Postman** per effettuare test manuali delle diverse rotte del Server, controllando infine le risposte ottenute per la gestione degli errori. Di fatto, questi strumenti si sono rivelati essere utili per verificare il corretto comportamento del Server, ma non solo, anche per consentire lo sviluppo del Client, per avere un riscontro su come le richieste dovevano essere effettuate, quali dati dovevano essere trasmessi e in quale modo, evitando, in caso di errori, lavoro di debugging per la formulazione corretta della richiesta.

5.1.2 Testing del Client Mobile

Per quanto riguarda il testing del Client Mobile sono stati effettuati test manuali per verificare il corretto funzionamento dell'interfaccia utente (UI) durante la sua implementazione.

Test manuali Le diverse pagine sono state testate una ad una, verificandone la corretta visualizzazione e navigazione nel dispositivo mobile. In seguito, sono stati definiti gli elementi grafici, la cui correttezza è stata verificata eseguendo la pagina all'interno dell'applicazione e verificandone la correttezza.

Per concludere, si è testata la reattività dell'interfaccia grafica e la correttezza dell'invio delle richieste al Server, nonché la corretta visualizzazione dei dati ottenuti in risposta, mediante la simulazione delle operazioni effettuate dall'utente.

5.2 Testing dei componenti IoT

Il testing dei componenti IoT è stato principalmente manuale in quanto si tratta di testare l'integrazione di vari componenti. Inoltre sono stati implementati alcuni test automatici per la centralina dell'irrigazione.

Test automatici Sono stati implementati alcuni test automatici nel componente per la centralina dell'irrigazione. Nello specifico per verificare il corretto funzionamento della classe `RemoteIrrigationController` che implementa il servizio esposto dal server gRPC. I test sono stati effettuati tramite il framework **JUnit**. Attraverso questi si è simulato di inizializzare uno *stub* per il servizio che inviasse

alcune richieste verificando che la risposta fosse quella attesa. In questo modo è possibile verificare localmente il corretto funzionamento del server gRPC.

Test manuali I test manuali hanno riguardato principalmente la verifica della corretta comunicazioni tra componenti, come nel caso dei sensori in cui sono stati eseguiti gli applicativi previsti ed è stato eseguito il broker RabbitMQ in un container Docker. Per riuscire a eseguire questi test è risultato fondamentale l'utilizzo di stampe a video per segnalare le operazioni effettuate.

5.3 Modalità di lavoro

La gestione del progetto è avvenuta utilizzando un apposito repository GitLab, dove è stato creato un branch **master** sul quale è stato caricato il progetto finale, un branch **develop** dove venivano man mano caricate le *feature* definitive: su questo branch sono stati aperti diversi branch, uno per ogni entità principale del progetto.

Parte del lavoro è stato suddiviso tra i membri del gruppo, nello specifico: Alex Baiardi si è occupato dell'implementazione della parte IoT del progetto e della containerizzazione dei moduli, Francesca Frattini ha seguito lo sviluppo del Client Mobile ex-novo; la parte di Server, invece, è stata a quattro mani.

Per ciò che concerne l'attività svolta, sono stati effettuati incontri preparatori in cui è stata definita la struttura architetturale del progetto. In questa fase si è lavorato sulle necessità lato utente stabilendo:

- le modifiche da apportare all'interfaccia grafica dell'applicazione con la realizzazione di opportuni *mock-up*;
- sono state definite formalmente le API per il Server Web in modo che ci fosse un punto di riferimento specifico per lo sviluppo dei componenti che le implementano e per chi le utilizza;
- definizione del modello dei dati con la definizione dello schema del database, che è poi stato raffinato in fase di implementazione.

Dopo questa parte di design si è proceduto congiuntamente nello sviluppo del Server Web in quanto rappresenta l'elemento centrale del progetto che permette l'interazione dei vari componenti. A questo punto i componenti del team hanno lavorato parallelamente nell'esecuzione dei loro incarichi, confrontandosi in caso di problemi: essendo il Server sviluppato ad una versione semi-definitiva, non sono sorti problemi di sincronizzazione con il Client Mobile e la parte IoT.

6 Deployment

6.1 Containerizzazione

Per il *deploy* del sistema si è scelto di utilizzare Docker, il quale fornisce una piattaforma di *containerizzazione* per creare, distribuire ed eseguire applicazioni in modo isolato senza dover gestire problematiche legate alla configurazione e gestione dell'ambiente esterno. Infatti, eseguendo una applicazione all'interno di un *container*, essa viene isolata dalle altre applicazioni in esecuzione grazie alla virtualizzazione a livello del sistema operativo. Si possono vedere i container come macchine virtuali che possono essere inserite in rete, anch'essa virtualizzata: ciò rende questa tecnologia particolarmente utile nell'ambito dei sistemi distribuiti.

All'interno del progetto sono stati utilizzati anche *Docker Compose* e *Docker Swarm*, due strumenti forniti da Docker per facilitare la gestione e l'orchestrazione di più container: Docker Swarm permette di gestire dei cluster di container distribuiti su più nodi in ambiente Docker; il secondo è invece uno strumento per definire e gestire un'applicazione multi-container.

Per il rilascio del nostro sistema sono state disegnate le immagini dei container, pensando di rilasciare ogni modulo ad eccezione di *mobileApp* come un container; ciascuna immagine contiene, quindi, tutto il necessario per costruire ed eseguire un applicativo. Sono state disegnate quattro immagini:

- **webserver**
- **sensorUnit**
- **sensorDataCollector**
- **irrigationUnit**

Tra queste, *webserver* si basa su un'immagine *node* all'interno della quale vengono copiati i file sorgenti necessari e sfruttando *npm* (*Node Package Manager*) vengono installate le dipendenze e si esegue l'applicazione. Le altre immagini sono tutte molto simili tra loro, in quanto trattasi di applicazioni Java gestite tramite Gradle: infatti, esse si basano sull'immagine ufficiale di Gradle che include JDK 17. Partendo da quest'ultima, vengono copiati i sorgenti e la configurazione Gradle, gestendo quindi la compilazione e l'esecuzione dell'applicazione. Per le applicazioni che necessitano di parametri per il funzionamento, questi vengono passati mediante variabili d'ambiente trasmesse durante l'esecuzione dell'immagine.

Il sistema di container è stato strutturato definendo tramite Docker Compose una unità **server** che include: *webserver*, *sensorDataService* ed il broker *RabbitMQ*. Ed una seconda unità **IoT** in cui vengono eseguiti i due applicativi relativi ai sensori ed alla centralina d'irrigazione: questa unità verrà poi replicata per ogni utente del sistema, in quanto corrisponde ai dispositivi dei quali l'utente dovrebbe teoricamente disporre.

6.2 Istruzioni di deployment

All'interno di questa sezione sono indicate le istruzioni per il deployment del progetto, necessarie da seguire per la corretta messa in funzione ed esecuzione del sistema.

6.2.1 Setup ed installazioni di base

Per prima cosa è necessario effettuare il download del repository. Per eseguire la *build* del sistema è necessario avere installato in precedenza:

- Una versione di **Java 17** o superiore
- Un **Android SDK**
- **Gradle**
- **Docker**

Per comodità, si può scaricare direttamente **Android Studio** che contiene già i diversi elementi specificati in precedenza (a parte Docker), ed aprire il progetto all'interno dell'IDE.

Una volta verificato di possedere tutte le impostazioni necessarie, si può proseguire con la parte relativa alla messa in funzione con Docker.

6.2.2 Containerizzazione

Per poter avviare il Server è necessario aver installato Docker in precedenza, per fare ciò seguire le istruzioni al seguente link: <https://docs.docker.com/get-docker/>.

Il progetto è predisposto per essere avviato utilizzando due versioni diverse del database. La versione di default sfrutta una immagine Docker di MongoDB, pensata per il testing in ambiente locale, in cui viene inizializzato un database vergine. In alternativa è possibile utilizzare una istanza di MongoDB in cloud. In questo caso il database contiene già un utente di prova il cui username è `marcobianchi@hotmail.it` e password `password` per poter testare l'applicazione senza effettuare la registrazione. Per poter utilizzare l'istanza esterna di MongoDB è necessario impostare l'URL all'interno del file `.env`, contenuto nella directory `serverBackend/`, alla variabile `PLANTLO_DB_URI` e inserire la stringa commentata.

Una volta installata una versione di Docker, aprire la directory `container/`, al cui interno sono presenti le diverse immagini Docker ed i due file `docker-compose`, aprire un prompt dei comandi dalla directory del progetto ed eseguire lo script bash `start.sh`, che si occupa di istanziare i diversi container: quest'operazione può richiedere diversi minuti al primo avvio, quindi attenderne il completamento. Una volta che i container sono stati istanziati, per poter iniziare ad utilizzare l'applicazione occorre attendere che Gradle completi la build dei moduli del progetto, che

richiedere alcuni minuti: per controllarne l'avanzamento, si può recuperare il container_id di interesse tramite il comando `docker ps` e successivamente effettuare il comando `docker logs *container_id*`.

Per avviare i componenti IoT associati ad un utente è necessario aprire il file chiamato `docker-compose-IoT.yaml` e modificare i valori compresi tra `*` inserendo l'IP della macchina da cui viene lanciato Docker e la porta su cui la centralina di irrigazione deve mettersi in ascolto (solitamente la 10000) e lo username dell'utente alla voce `USER` dell'immagine `irrigation_unit` e `sensor_unit`. Per il corretto funzionamento di questi componenti è necessario che l'utente sia stato registrato, perciò nel caso in cui si utilizzi il database in cloud è possibile utilizzare l'utente di prova `marcobianchi@hotmail.it`, altrimenti è necessario effettuare la registrazione tramite apposite API o con applicazione mobile.

Fatto ciò, si può eseguire il secondo script bash `startIoTmodule.sh` per avviare i servizi IoT. Una volta che tutti i container sono in esecuzione, si può avviare l'applicazione Android seguendo le istruzioni nel paragrafo sottostante.

Quando si vogliono terminare tutti i container Docker e terminare il Server basta aprire un prompt di comandi nella cartella di progetto e lanciare lo script `stop.sh`

6.2.3 Setup del Client Mobile

Il Client Mobile necessita di un file di configurazione JSON chiamato `config.json` situato all'interno della directory `sd-project-baiardi-frattini-ay2122/mobileApp/src/main/res/raw`. Un esempio di configurazione è raffigurato nel listato 8, in particolare:

- `host` identifica l'host del Server, quindi bisogna modificare il valore `EXTERNAL_DOCKER_MACHINE_IP` con l'indirizzo IP della macchina dove è presente il cluster docker del Server
- `port` è il numero che identifica la porta sulla quale il Server è in ascolto delle richieste HTTP

```
{
  "host": "EXTERNAL_DOCKER_MACHINE_IP",
  "port": 5000
}
```

Listing 8: File di configurazione del Client Mobile

Una volta modificato il file, si può installare l'applicazione con Android Studio collegando il dispositivo fisico oppure tramite l'emulatore integrato di Android Studio.

7 Esempi di utilizzo

Supponiamo che un utente vuole gestire le piante presenti all'interno della sua serra: per prima cosa dovrà registrarsi al sistema utilizzando l'applicazione Android fornita (figura 17).

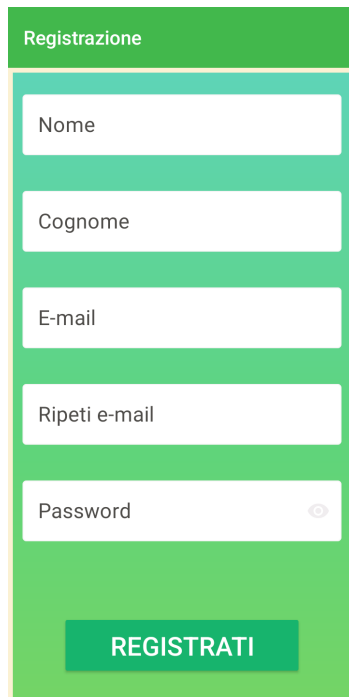


Figura 17: Schermata di registrazione

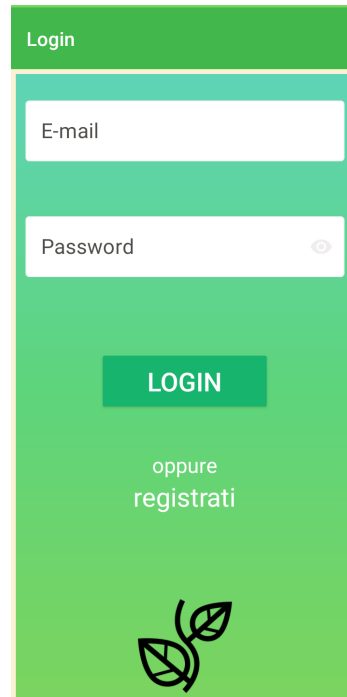


Figura 18: Pagina di login

Nel caso in cui l'utente si sia già registrato in precedenza, può accedere all'applicazione effettuando il login rappresentato in figura 18.

Una volta effettuato l'accesso, l'utente si trova nella pagina principale dell'applicazione da cui può decidere se: creare una nuova pianta, visualizzare quelle precedentemente create ed eventualmente modificarle/cancellarle, visualizzare i dati ambientali della serra oppure gestire l'irrigazione da remoto (figura 19).

Scegliendo la creazione della pianta, si aprirà l'interfaccia apposita, tramite cui, inserendo i dati necessari, si potrà aggiungere la pianta alla propria galleria (figura 20).



Figura 19: Homepage dell'applicazione



Figura 20: Schermata di aggiunta pianta

Nella sezione galleria dell'applicazione le piante presenti vengono visualizzate sotto forma di elenco, raggruppate in base al settore di appartenenza, indicato dall'utente in fase di creazione (figura 21).

In questa sezione è possibile modificare ogni pianta, cambiandone la specie e la data di semina, oppure eventualmente eliminarla dalla galleria (figure 22 23).

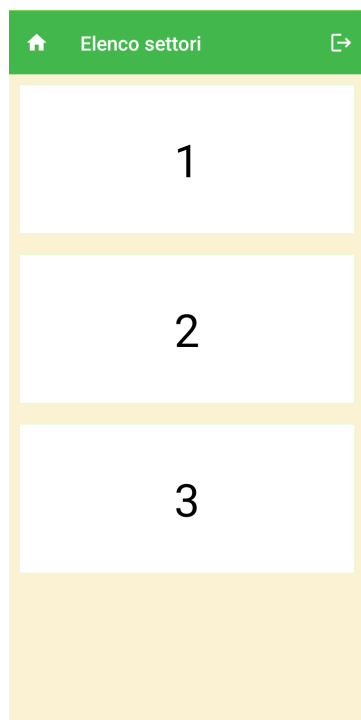


Figura 21: Elenco settori



Figura 22: Galleria piante settore 3



Figura 23: Impostazioni pianta

Supponiamo invece che l'utente voglia visualizzare le statistiche ambientali della serra, tramite l'apposita interfaccia vengono visualizzati i dati relativi all'ultima misurazione, di cui sono indicate data e ora, ed è possibile effettuare il *refresh* dei dati, in quanto il sensore invia nuovi dati ogni minuto (figura 24).

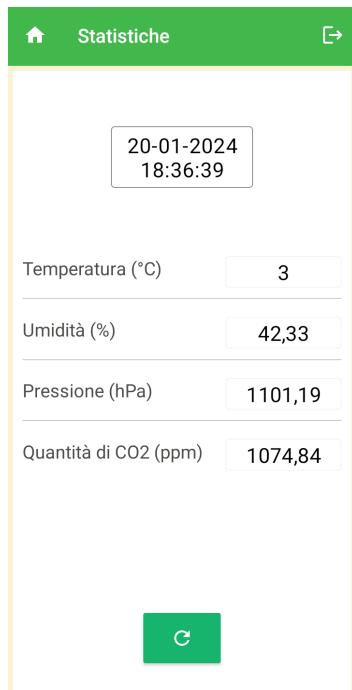


Figura 24: Schermata statistiche serra



Figura 25: Pagina gestione irrigazione

Nel caso in cui si voglia gestire la centralina dell'irrigazione, cliccando sull'apposito pulsante nell'homepage, verranno visualizzati i vari settori della serra e lo stato dell'irrigazione per ognuno di essi (accesa/spenta); l'utente può attivare o disattivare l'irrigazione in ogni settore (figura 25).

8 Conclusioni

L'obiettivo di questo progetto era quello di realizzare un'applicazione che consentisse la gestione di una serra *intelligente*, rendendone l'organizzazione semplice e agevole. Si è inoltre stabilito che, l'accesso all'applicazione da parte degli utenti avvenga mediante un'applicazione Android.

Per realizzare questo obiettivo, dapprima sono stati stabiliti i requisiti funzionali e non funzionali che il sistema doveva avere, dopodichè sono state individuate le diverse entità che costituiscono il modello del dominio, le quali sono state successivamente implementate definendone struttura, comportamento ed interazione.

In particolare, il sistema si è rivelato essere costituito da quattro entità fondamentali: un database non relazionale, un Server Web, un Client Mobile e gli elementi di IoT.

Il Server realizzato è di tipo *stateless*, quindi non mantiene salvato alcuno stato dell'applicazione e si occupa di soddisfare le diverse richieste provenienti dal Client tramite l'interazione col database. Il comportamento del Client, invece, è determinato dalle azioni che compie l'utente, le quali possono far emanare l'invio di richieste verso il Server.

Il funzionamento dell'applicazione è basato sul meccanismo di *Remote Procedure Call*, il quale consiste in una chiamata a procedura remota del Client verso il Server, mettendosi poi in attesa di ricevere una risposta al completamento della richiesta. Per quanto riguarda l'interazione, essa nasce sempre dal Client, tramite l'invio di una richiesta al Server: quest'ultima viene soddisfatta attraverso una o più interazioni del middleware con il database, infine l'interazione viene conclusa con l'invio di una risposta da parte del Server e la sua ricezione dal Client.

Le diverse funzionalità realizzate sono state testate tramite l'utilizzo di test sia automatici che manuali e, analizzando quanto fatto al termine dell'attività, si può affermare che il sistema realizzato soddisfa gli obiettivi ed i requisiti individuati e che si è complessivamente soddisfatti del risultato raggiunto.

8.1 Lavori futuri

Alcune possibili funzionalità aggiuntive dell'applicazione che possono essere implementate in futuro, in aggiunta a quelle già presenti, potrebbero essere:

- Integrare una parte di IA per il riconoscimento delle piante tramite la fotocamera del dispositivo Android,
- Perfezionare la sezione IoT per dare la possibilità all'utente di impostare una determinata temperatura all'interno della serra,
- Includere la possibilità di pianificare l'irrigazione in modo automatizzato con la creazione di programmi basati sul giorno della settimana e sull'orario.

8.2 Cosa è stato appreso

In questa sezione, ogni componente del gruppo, ha indicato quali sono stati i concetti appresi durante la realizzazione di questo progetto.

Alex Baiardi Con questo progetto ho avuto la possibilità di fare esperienza con vari aspetti che entrano in gioco nello sviluppo di applicazioni distribuite. In particolare, è risultato utile per capire meglio nella pratica come gestire il coordinamento tra vari applicativi andando a definire i vari meccanismi di comunicazione con la possibilità di comprenderne meglio le potenzialità. Ho trovato, in questo, molto utile e interessante l'utilizzo di gRPC con la possibilità di definire un servizio attraverso un IDL e grazie a questo poter generare automaticamente parte del codice. Allo stesso tempo ho avuto la possibilità di inserire tra le mie conoscenze anche la gestione delle comunicazioni tramite RabbitMQ. In questo progetto ho avuto modo di utilizzare per la prima volta Docker comprendendone il funzionamento e le potenzialità. Inoltre, questo progetto si è rivelato una ulteriore occasione per consolidare e approfondire l'utilizzo Gradle per la gestione delle build e Git per la coordinazione del lavoro in team.

Francesca Frattini Grazie a questa esperienza ho appreso diverse cose, difatti mi ha permesso di rafforzare le conoscenze acquisite durante il corso e comprendere meglio il funzionamento di un sistema distribuito. Ho avuto occasione di utilizzare nuovi strumenti come OpenAPI, per la specifica dell'API e Swagger, che si è rivelato essere uno strumento molto utile e di facile utilizzo. Inoltre, lo sviluppo del progetto mi ha permesso di migliorare le conoscenze di Gradle, utilizzato per la gestione dei diversi moduli del progetto e delle dipendenze, realizzando per la prima volta un progetto costituito da più moduli. Infine, grazie all'uso della libreria Jest, ho imparato ad effettuare il testing di un progetto Typescript. Sono abbastanza soddisfatta del lavoro che è stato svolto e dei risultati ottenuti, considerando le difficoltà ed i problemi riscontrati durante lo sviluppo.

Note stilistiche

- Il **grassetto** è stato utilizzato per indicare aspetti importanti
- il *corsivo* è stato utilizzato per indicare aspetti tecnici, o parole inglesi non di uso comune nella lingua italiana
- il `monospace` è stato utilizzato per indicare aspetti relativi al codice o all'implementazione

Riferimenti bibliografici

- [1] MongoDB. Database mongodb atlas. 2023. URL <https://www.mongodb.com/it-it/atlas/database>.
- [2] Android. Guide to app architecture - android developers. dec 2023. URL <https://developer.android.com/topic/architecture?hl=it>.
- [3] Google. Gson user guide. nov 2023. URL <https://github.com/google/gson/blob/main/README.md>.
- [4] Docker. Docker guide. jan 2024. URL <https://docs.docker.com/get-started/overview/>.
- [5] Google. Volley overview. dec 2023. URL <https://google.github.io/volley/>.
- [6] Broadcom. Rabbitmq - features. jan 2024.