Setting Up Home Final Report

Aurora Laghi
aurora.laghi@studio.unibo.it

Anna Vitali
anna.vitali4@studio.unibo.it

Elena Yan
elena.yan@studio.unibo.it

27 marzo 2022

Per il progetto di Sistemi Distribuiti, si è pensato di realizzare un'applicazione che consenta di gestire e pianificare diverse attività domestiche fra diversi utenti, in un contesto di convivenza. L'applicazione potrà essere utilizzata sia tramite un'app Android che mediante una Web App.

Un utente per poter iniziare ad utilizzare l'applicazione deve per prima cosa registrarsi, dopodiché avrà la possibilità di: creare un nuovo spazio rappresentante la casa in cui convive con gli altri utenti, oppure unirsi a uno spazio esistente.

All'interno di uno spazio vi possono essere due tipi di utente: amministratore e membro; gli utenti amministratori si occupano di gestire lo spazio, pianificando i turni e le relative faccende da svolgere e di gestire gli utenti, potendo aggiungere nuovi partecipanti allo spazio, rimuoverli o nominare dei membri amministratori.

Un utente appartenente a uno spazio, avrà la possibilità di visionare per le diverse faccende, i diversi turni che sono stati assegnati e di poter inviare e ricevere notifiche dagli altri membri, per indicare il completamento di un turno. Inoltre, un utente riceverà una notifica quando sarà il suo momento di svolgere una determinata faccenda e in caso di ritardo nello svolgimento di un determinato compito, gli utenti dello spazio potranno inviare notifiche per ricordare all'utente "pigro" che è il suo momento di svolgere il turno.

Indice

Elenco delle figure

Listati

1 Obiettivi/Requisiti

Si vuole realizzare un'applicazione che consenta la gestione di diverse attività domestiche fra diversi utenti che vivono assieme, l'applicazione quindi potrà essere utilizzata sia in un contesto familiare che in un contesto di convivenza.

Gli utenti potranno utilizzare il servizio sia mediante un'applicazione Mobile, per dispositivi Android, che mediante una Web App.

Un utente registratosi al sistema, avrà la possibilità di creare un nuovo spazio rappresentante la casa in cui vive con gli altri membri, oppure unirsi a uno spazio già esistente.

L'applicativo, ha come obiettivo principale di consentire la gestione di diverse faccende domestiche, potendo specificare per ogni lavoro da svolgere:

- l'ordine con cui i diversi membri dovranno turnarsi,
- la cadenza del turno: giornaliera, settimanale o mensile.

Dando la possibilità ai diversi partecipanti di poter visualizzare per ogni faccenda, i diversi turni che sono stati svolti e quelli ancora da effettuare. A tale fine, gli utenti potranno ricevere notifiche le quali lo informeranno che un membro ha svolto il proprio turno, o che è il suo momento di svolgere una faccenda.

Sempre per la gestione dei turni, si vuole dare anche la possibilità agli utenti, di ricordare ai partecipanti "pigri" che è il momento di svolgere il proprio turno, mediante l'invio di un'apposita notifica.

Per consentire l'amministrazione di uno spazio e la pianificazione dei turni, sono previste due categorie di utenti: **membri** e **amministratori**; gli utenti amministratori potranno svolgere delle funzionalità aggiuntive rispetto a quelle previste per i membri:

- potranno aggiungere o rimuovere partecipanti allo spazio,
- eliminare lo spazio in cui partecipano da amministratori,
- nominare altri utenti amministratori,
- stabilire i turni per le diverse faccende avendo anche la possibilità di inserirne di nuove, in base alle proprie esigenze.

Infine, si vuole realizzare un sistema che sia in grado di gestire le richieste provenienti dai diversi utenti sia lato Mobile che Web; l'applicazione Android dovrà funzionare correttamente su diversi dispositivi con differenti caratteristiche hardware, mentre la Web App deve poter essere eseguita correttamente su diversi browser.

1.1 Scenarios

Supponiamo che vi sia un utente che decide di utilizzare l'applicazione, per poter gestire le diverse faccende fra i diversi coinquilini che vivono assieme a lui. Una volta aperta l'applicazione, all'utente verrà richiesto di registrarsi oppure di effettuare il login se già in possesso di un account.

Una volta effettuato l'accesso, l'utilizzatore potrà decidere se creare un nuovo spazio oppure unirsi a uno precedentemente creato. Questo quindi implica che fra tutti i coinquilini, ve ne deve essere uno che crea il nuovo spazio e gli altri che vi accedono.

Dopo aver creato uno spazio avendo specificato il relativo nome, l'utente creatore (amministratore di default), potrà far accedere gli altri coinquilini allo spazio, mediante la condivisione del codice identificativo di questi.

Un utente che accede ad uno spazio è disegnato di default come membro e potrà navigare all'interno dell'applicazione per poter vedere i diversi turni assegnati, per le rispettive faccende e le informazioni sui partecipanti.

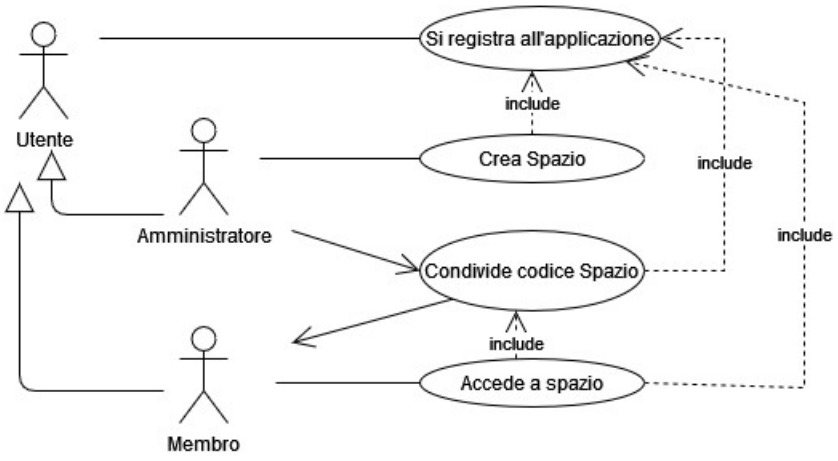


Figura 1: Diagramma dei casi d'uso creazione e accesso ad uno spazio

Una volta che tutti i coinquilini si sono uniti allo spazio, l'amministratore potrà decidere di: cancellare uno o più partecipanti, assegnare ad alcuni dei partecipanti il ruolo di amministratori, oppure assegnare i turni per le diverse faccende.

Per poter nominare altri utenti amministratori o cancellare un partecipante, ci si dovrà spostare nella sezione dell'applicazione dedicata ai partecipanti di uno spazio e compiere tali operazioni, mediante l'apposita interfaccia fornita.

Se si desidera invece assegnare i turni per le diverse faccende, si dovrà andare nella pagina dedicata alle faccende di uno spazio, potendo visualizzare le faccende standard già presenti o aggiungerne di nuove, specificando il loro nome. Una volta selezionata la faccenda, per cui si vuole pianificare i compiti, si dovrà stabilire l'ordine dei partecipanti e la frequenza con cui ci si alternerà, potendo scegliere fra: giornaliera, settimanale o mensile.

Ai partecipanti una volta che sono stati stabiliti i turni, verranno inviate delle notifiche che li manterranno aggiornati sui i turni che sono stati svolti, eventuali modifiche degli assegnamenti o se è il loro turno di svolgere una determinata faccenda; le notifiche potranno essere visualizzate in un'apposita pagina dell'applicativo.

A questo punto, un utente di turno per una certa faccenda, che desidera segnalare il completamento del compito potrà accedere all'applicazione, navigando all'interno di

questa accedendo allo spazio di appartenenza e spostandosi nella sezione dedicata ai turni, indicandone il completamento mediante l'utilizzo dell'apposita interfaccia.

Gli utenti sempre nella pagina dedicata ai turni, potranno visualizzare i compiti che sono stati svolti e quelli pianificati da svolgere, sapere chi è il partecipante di turno in questo momento e nel caso, ricordarli che è il momento di svolgere la faccenda.

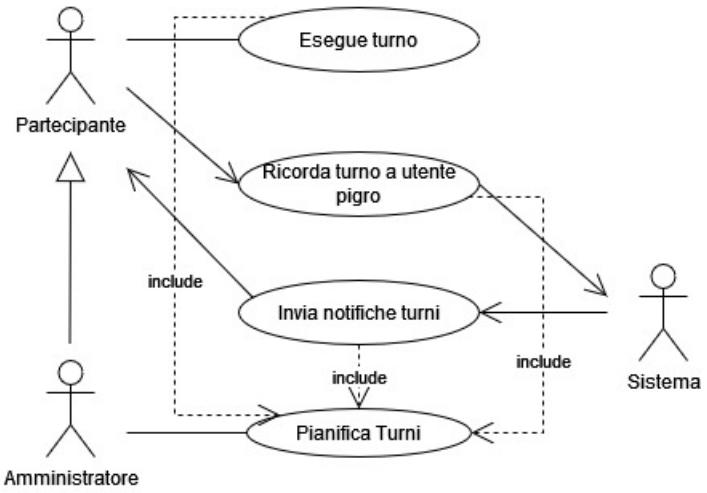


Figura 2: Diagramma dei casi d'uso assegnazione e svolgimento turni

L'applicativo, quindi, gestirà i diversi utenti di uno spazio riconoscendo il **ruolo** che essi svolgono all'interno di questi, mostrando funzionalità aggiuntive agli amministratori che invece non verranno mostrate ai membri.

1.2 Metodi di autovalutazione

Per valutare l'efficacia del software prodotto, si pensa di utilizzare sia test automatici che manuali. In particolare, si pensa di testare la maggior parte delle diverse funzionalità di *back-end* tramite test automatici, mentre la reattività dell'interfaccia grafica, sia della Web App che dell'applicazione Mobile, verrà testata mediante test perlopiù manuali, ma dove possibile soprattutto per l'applicazione Mobile verranno utilizzati anche test automatici. I dettagli riguardanti i test verranno approfonditi in sezione 5.

Per testare invece la qualità del servizio offerto, si guarderà se l'applicativo realizzato è in grado di soddisfare tutti i requisiti emersi nell'analisi e come tali requisiti vengono soddisfatti, cercando di rispettare i diversi obiettivi stabiliti e realizzare un sistema efficiente.

2 Analisi dei requisiti

L'applicazione ha come obiettivo principale, quello di gestire e pianificare diverse attività domestiche fra diversi utenti in un contesto di convivenza, a tale fine sono stati

individuati i seguenti requisiti funzionali e non funzionali.

2.1 Requisiti funzionali

- All'avvio dell'applicazione, verrà mostrata l'interfaccia di login, attraverso cui sarà possibile: registrare un nuovo utente oppure accedere con le proprie credenziali (email, password).
- Dopo aver effettuato l'accesso un utente potrà:
  - Creare un nuovo spazio, rappresentante la casa,
  - Unirsi a un nuovo spazio, precedentemente creato da un altro utente,
  - Accedere a uno spazio a cui già appartiene.

All'interno di uno stesso spazio, possiamo trovare due categorie di utenti: amministratori e membri. Agli utenti **membri** di uno spazio deve essere data la possibilità di:

- Visualizzare le informazioni delle spazio di appartenenza comprese quelle sugli altri partecipanti, potendo vedere anche il ruolo che essi svolgono,
- Visualizzare i diversi turni assegnati per le diverse faccende, in particolare, l'applicazione dovrà consentire di vedere per ogni faccenda, sia i turni che sono ancora da svolgere che lo storico di quelli già eseguiti, nonché la persona che è di turno in questo momento,
- Ricevere notifiche per essere a conoscenza di un turno eseguito da un altro partecipante, o per sapere che è il proprio momento di svolgere un compito,
- Inviare notifiche al partecipante di turno in questo momento, per una relativa faccenda, in modo da ricordarli che è il momento di svolgerlo,
- Dare la possibilità a un partecipante di svolgere un turno in ritardo, rispetto a quanto programmato a causa di eventuali dimenticanze ad esempio.

Gli utenti **amministratori**, invece, oltre a poter svolgere tutte le funzionalità previste per gli utenti membri avranno la capacità di:

- Cancellare lo spazio di appartenenza o modificarne il nome,
- Aggiungere o rimuovere partecipanti allo spazio,
- Cambiare il ruolo di un partecipante da membro ad amministratore,
- Gestire le diverse faccende da svolgere, potendo inserire nuove faccende da aggiungere a quelle standard già presenti,

- Pianificare per ogni faccenda i turni dei diversi partecipanti, scegliendo l'ordine con cui questi verranno alternati e la frequenza del compito da svolgere (giornaliera, settimanale, mensile),
- Eliminare o modificare i turni assegnati, stabilendo un ordine dei partecipanti o una frequenza diversi.

2.2 Requisiti non funzionali

- Il servizio deve essere in grado di gestire richieste provenienti da più utenti contemporaneamente,
- Il design deve permettere una facile estendibilità in caso di modifiche future o eventuali funzionalità aggiuntive,
- L'applicazione Mobile deve essere in grado di funzionare su diversi dispositivi Android, con differenti caratteristiche hardware, che presentano una versione del sistema maggiore o uguale alla 7.0,
- La Web App deve poter essere eseguita e funzionare correttamente sui di diversi tipi di browser,
- L'interfaccia grafica sia lato Mobile che lato Web deve essere reattiva all'interazione con l'utente.

2.3 Analisi e modello del dominio

Dato che l'applicazione consiste in un servizio Web messo a disposizione di diversi utenti, si vuole realizzare il programma aderendo al modello dei servizi *ReSTful*, realizzando un'interfaccia uniforme del servizio offerto che possa essere riutilizzata.

Per questo, le entità base del dominio che si prevede di modellare saranno:

- I **clients** Mobile e Web, che invieranno richieste al Server mediante la specificazione di opportune rotte, le quali consentiranno l'esecuzione di determinate operazioni,
- Il **Server**, il quale si occuperà di esporre il servizio ai clients, di definire le rotte e di implementare le diverse operazioni da effettuare, interagendo con un database relazionale,
- Un **database relazionare**, il quale si occuperà di mantenere le informazioni relative: agli utenti, agli spazi, ai turni, alle faccende ecc. e di fornirle al Server su richiesta, per l'esecuzione delle operazioni.

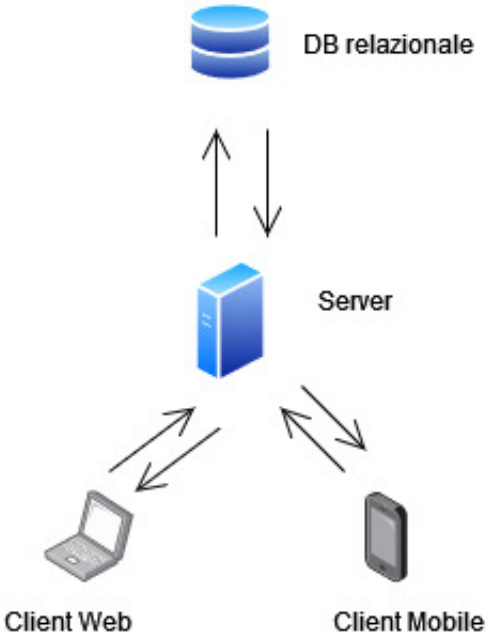


Figura 3: Entità del dominio e le loro interazioni

3 Design

In base alle informazioni che sono state raccolte nelle fasi di analisi, si è passati alla fase di progettazione definendo la struttura delle diverse entità coinvolte e le loro interazioni i cui dettagli sono trattati nelle prossime sezioni.

3.1 Struttura

Come abbiamo detto prima il nostro dominio è costituito da quattro entità base: database relazionale, Server, Client Web e Client Mobile.

Per la realizzazione del progetto, si è pensato di effettuare più repliche del servizio in modo tale da aumentare la *availability* e *reliability* del Server, nel rispondere alle diverse richieste dei clients. Tali repliche sono state realizzate mediante l'utilizzo di Docker e gli elementi di Docker Compose e Docker Swarm, in particolare quest ultimo mette a disposizione un meccanismo di *load-balancing*, che consente di bilanciare il carico di lavoro fra le diverse repliche del servizio, in modo tale che tutte vangano sfruttate e non vi siano repliche sovraccariche di lavoro, per maggiori dettagli relativi alla containerizzazione si veda la sezione 4.2.2.

Di conseguenza, la macro-struttura del nostro sistema può essere rappresentata dalla figura seguente.

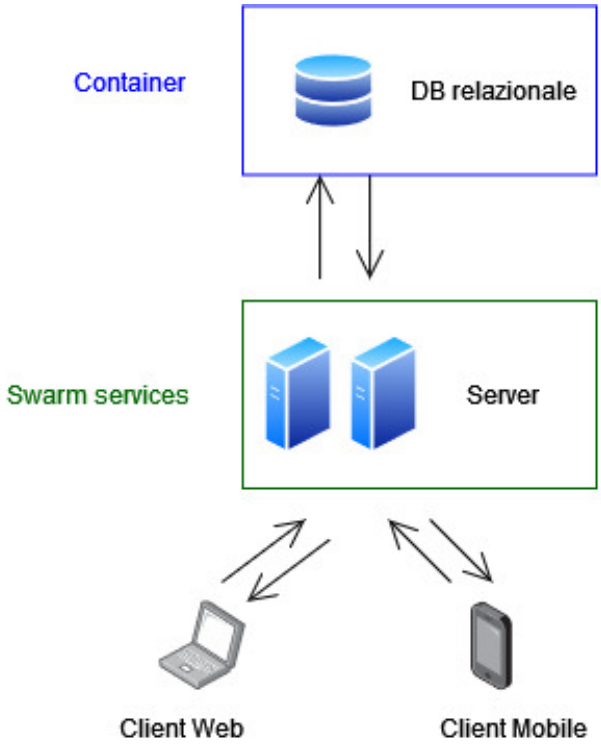


Figura 4: Macro-struttura sistema

3.1.1 Struttura del database

Per prima cosa è stato definito lo schema *Entity/Relationship* del database, mettendo in luce quali sono le diverse entità che dovranno essere modellate e le relazioni che vi sono fra queste.

Come si può vedere dalla figura 5 lo schema è costituito dalle seguenti entità:

- **User**, rappresenta gli utenti iscritti all'applicazione,
- **Space**, rappresenta gli spazi creati a cui un utente si può iscrivere,
- **Shift**, è l'entità centrale dello schema, è collegato a tutte le altre entità, consente di memorizzare un turno assegnato ad un determinato utente facente parte di uno spazio, relativo a una certa faccenda,
- **Housework**, rappresenta le diverse faccende, le quali possono essere di due tipi: standard e custom e vengono espresse con una gerarchia totale ed esclusiva (è possibile avere solo questi due tipi di faccende e una faccenda custom non può essere anche standard),
- **Frequency**, indica la frequenza di un turno, che può essere: giornaliera, settimanale o mensile,

- **Notification**, rappresenta le notifiche che vengono inviate agli utenti dal sistema, relative ai turni.

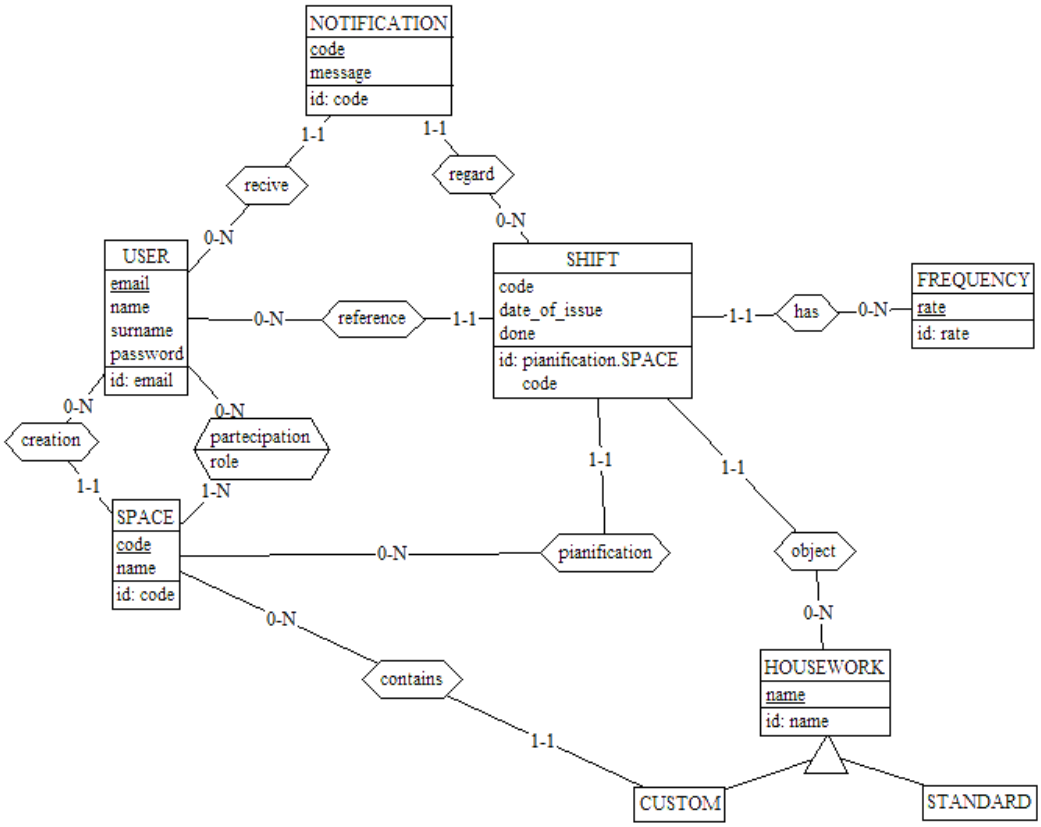


Figura 5: Schema Entity/Relationship

Inizialmente si era pensato di modellare l'entità partecipante come sottotipo di utente, associando a tale entità le relazioni relative ai turni e non direttamente allo `User` e indicando la partecipazione di un utente a uno spazio, mediante apposita associazione da `Participant` a `Space` di tipo `(1:N)` e associando al partecipante un codice identificativo univoco all'interno dello spazio. Questa scelta si è rilevata essere **sbagliata**, perché oltre a inserire maggiore complessità rispetto alla soluzione successivamente adottata, non consentiva la corretta identificazione dei partecipanti. Di conseguenza ci si è resi conto, che la partecipazione di un utente ad uno spazio, doveva essere modellata mediante un'associazione `(N:N)` da `User` a `Space` con attributo ruolo, indicante che un utente, identificato dall'email, può partecipare a più spazi con un certo ruolo che può differire a seconda dello spazio in cui si trova.

3.1.2 Struttura del progetto principale: moduli e dipendenze

Il progetto è stato realizzato effettuando una suddivisione in più moduli, consentendo alle diverse entità di poter utilizzare le stesse realizzazioni per aspetti comuni.

I moduli realizzati sono di seguito elencati:

- **Service**, racchiude il servizio esposto dal Server nonché la definizione dell'API del servizio stesso, offrendo la specifica delle rotte che consentono di manipolare le differenti risorse,
- **Db**, tale modulo viene utilizzato per interagire con il database, esso racchiude le classi che effettuano le query necessarie a performare le operazioni sui dati, in base ai compiti che è necessario svolgere per il funzionamento dell'applicazione,
- **Entity**, in questo modulo troviamo le interfacce e relative implementazioni rappresentanti le entità del database realizzato, ed eventuali loro estensioni, per raccogliere le informazioni ottenute mediante interrogazioni al database,
- **Presentation**, esso fornisce le strutture di serializzazione e deserializzazione necessarie per convertire gli oggetti nel loro formato `JSON` e viceversa. In particolare, gli oggetti di riferimento sono quelli contenuti nel modulo `Entity`.
- **MobileApp**, questo modulo rappresenta il Client Mobile dell'applicazione. Il modulo comprende tutti i componenti necessari per la creazione dell'applicazione Android.
- **WebApp**, è il modulo che rappresenta il Client Web dell'applicazione.

Come si può vedere dalla figura 6, il modulo `Service`, dipende dai moduli `Db` e `Presentation` per il suo funzionamento e il modulo `Db`, per l'esecuzione delle operazioni, sfrutta le classi definite nel modulo `Entity`.

`MobileApp`, invece, dipende sia dal modulo `Entity`, che dal modulo `Presentation` per il suo funzionamento.

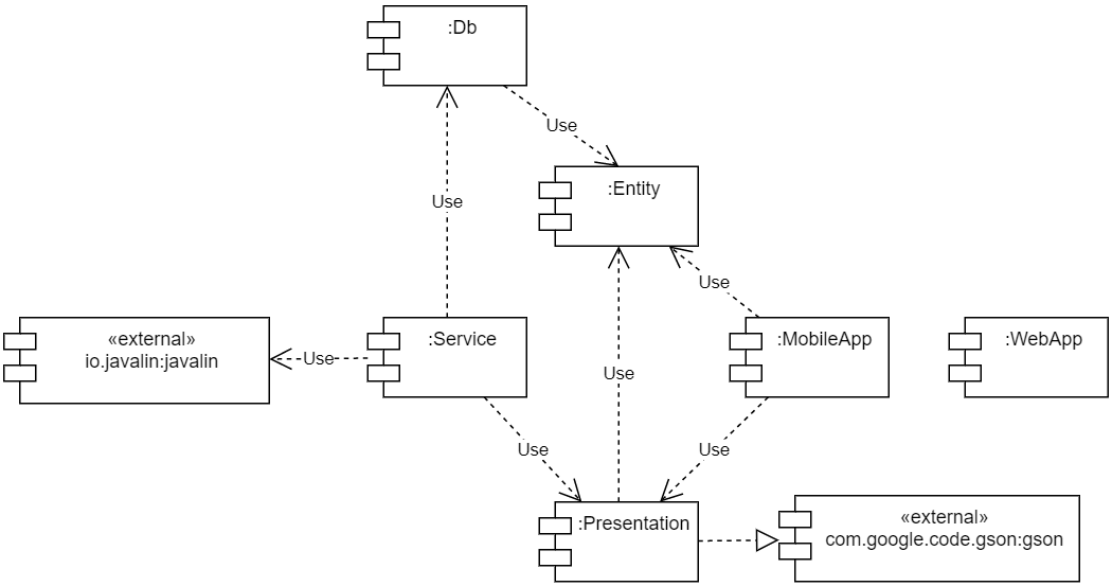


Figura 6: Dipendenze dei moduli del progetto

Alla luce di quanto descritto precedentemente, possiamo notare quindi che Client Mobile e Server utilizzano entrambi i moduli `Presentation` e `Entities`.

3.1.3 Struttura dei moduli comuni

Come abbiamo detto prima, vi sono moduli comuni a più entità che di fatto contribuiscono a definire la struttura di quell'entità, nelle successive sezioni quindi troveremo la descrizione di tali moduli, da tenere a riferimento per comprendere la struttura complessiva delle successive entità che si andranno a descrivere.

3.1.3.1 Struttura modulo Presentation

Il funzionamento dell'applicazione si basa sul meccanismo di *Remote Procedure Call (RPC)*, in cui un Client, effettua una chiamata a una funzione remota sul Server e attende il suo completamento. La chiamata può trasportare degli argomenti, da passare alla funzione, mentre la risposta del Server può trasportare risultati. Gli argomenti e i risultati, vengono passati **per valore**, quindi verrà effettuata una copia dell'oggetto che verrà inviato.

All'interno dei pacchetti che vengono scambiati fra Client e Server, si è deciso di esprimere gli oggetti con il formato `JSON` e per inviare un oggetto e convertirlo nel suo formato `JSON`, è necessario effettuare un'operazione di serializzazione, mentre per poter effettuare l'operazione inversa occorrerà procedere con la deserializzazione. Per poter svolgere questi procedimenti, si è deciso di utilizzare come supporto, la libreria `Gson`.

All'interno del modulo `Presentation` troviamo i `Serializer` e i `Deserializer`, dei diversi oggetti che possono essere inviati nei pacchetti.

Gli oggetti `Deserializer` implementano l'interfaccia `JsonDeserializer` e il metodo `deserialize` ed ereditano da una classe astratta `GeneralDeserializer`, la quale racchiude metodi di utilità, che possono essere utilizzati da tutti i `Deserializer`, per ottenere il valore di un campo di un oggetto in base al suo tipo.

Gli oggetti `Serializer`, invece, implementano l'interfaccia `JsonSerializer` e il metodo `serialize`.

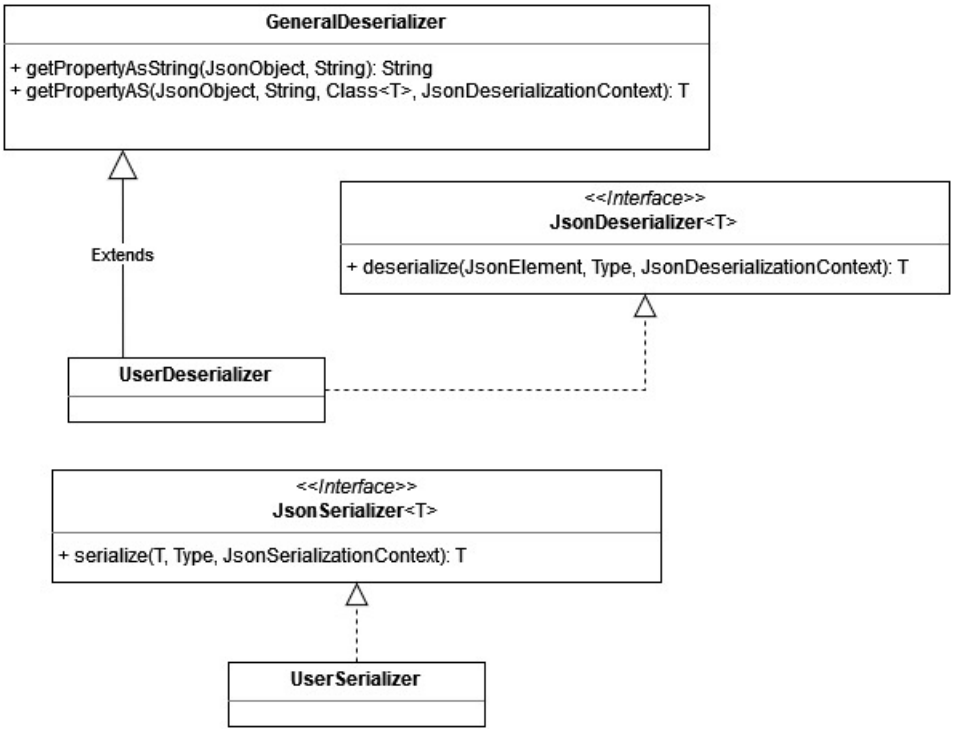


Figura 7: Diagramma delle classi: esempio User Serializer e Deserializer

3.1.3.2 Struttura modulo Entity

Per ogni entità del database è stata realizzata un'interfaccia e relativa implementazione che possiamo trovare all'interno di questo modulo nel package `entity`; di fatto gli oggetti realizzati, all'interno del modulo, sono dei *POJO (Plain Old Java Object)*, quindi si tratta di oggetti che presentano solamente metodi getter e setter per i campi definiti al loro interno, come è possibile vedere dalla figura 8.

La soluzione adottata per la modellizzazione, che prevede un'interfaccia per ogni entità del database e relativa implementazione, consente di poter aggiungere nuove classi, anche in relazione ai dati che possono essere ottenuti tramite le interrogazioni alla base di dati, favorendo **l'estensibilità**. Quindi, ad esempio, per rendere più chiaro questo concetto, per un turno è necessario sapere oltre che i dati del turno stesso, anche il nome e cognome dell'utente che dovrà svolgerlo; queste informazioni all'interno del database si trovano in due tabelle diverse `Shift` e `User` e verranno ottenute tramite un'apposita query effettuata

dal Server. Per poter salvare i dati restituiti dal database, è bastato creare una nuova interfaccia `ShiftWithUserInformation`, che eredita da `Shift` (rappresentante l'entità `Shift` del database) e relativa implementazione `ShiftWithUserInformationImpl` che eredità da `ShiftImpl` a cui poi si aggiungono i campi `name` e `surname` dell'utente.

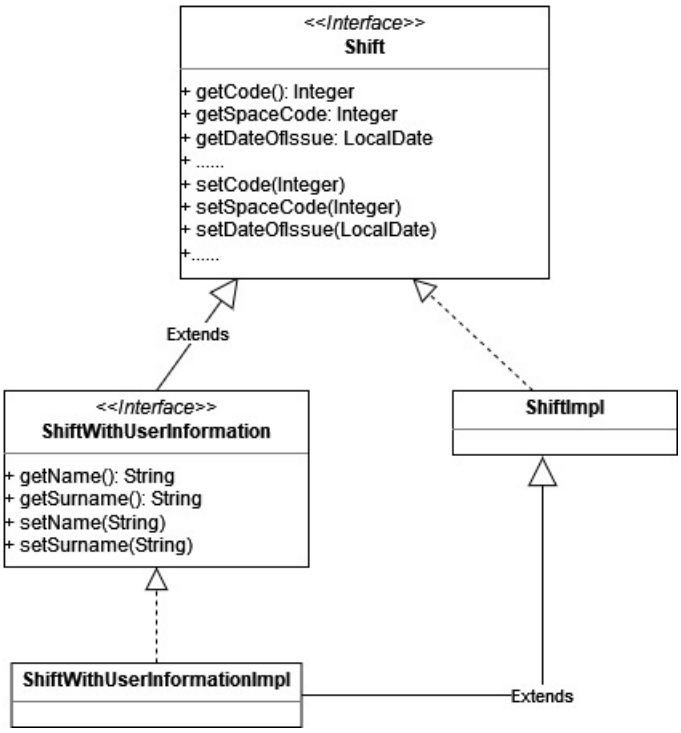


Figura 8: Diagramma delle classi: salvataggio informazioni turni

3.1.4 Struttura del Server

Si vuole realizzare un Server il cui compito principale, oltre che gestire le diverse richieste dei clients, sarà quello di interagire con un database per poter effettuare le operazioni richieste.

Si è deciso pertanto di realizzare uno **statless Server**; di conseguenza, ogni richiesta effettuata dai clients dovrà contenere tutte le informazioni necessarie affinché il Server possa capire correttamente la richiesta e intraprendere l'azione giusta, il Server rispetto a una soluzione **stateful**, quindi, non manterrà salvato alcuno stato dell'applicazione.

Attraverso questo tipo di approccio è possibile favorire: la visibilità, in quanto si potrà comprendere una richiesta senza dover guardare le precedenti, la scalabilità, il Server è in grado di liberare velocemente le risorse e la semplicità, in quanto l'implementazione del Server è più semplice rispetto a una soluzione stateful; il principale svantaggio di questa soluzione però, è che saranno necessarie più interazioni fra Client e Server con possibile riinvio di dati già mandati in precedenza.

Uno degli obiettivi che ci si è dati è quello di realizzare un interfaccia uniforme del servizio offerto, che possa essere riutilizzata e allo stesso tempo adottare un design che permetta una facile estendibilità, in caso di modifiche future o eventuali funzionalità aggiuntive. Per questo motivo il Server è stato realizzato utilizzando più moduli del progetto: un modulo `Db`, per l'interazione con il database, un modulo `Presentation`, per consentire la serializzazione e la deserializzazione dei dati e un modulo `Service`, che rappresenta il servizio offerto dal Server.

Pertanto la struttura del Server stesso può essere descritta analizzando la struttura dei moduli che lo costituiscono, per quanto riguarda il modulo `Presentation` essendo esso comune sia al Client Mobile che al Server la sua descrizione può essere trovata alla sezione 3.1.3.1.

3.1.4.1 Struttura modulo Db

All'interno del package `db`, possiamo trovare la classe `DbConnection`, la quale espone un unico metodo `getMYSQLConnection`, che si occupa di instaurare la connessione al database, questa classe verrà utilizzata per effettuare le diverse interrogazioni, consentendo di poter ottenere la connessione al database MySQL ed eseguire le query necessarie.

Per poter interagire con il database, è stato necessario modellare le diverse entità in esso presenti che troviamo all'interno del modulo `Entity` (da cui dipende il modulo `Db`) descritto nella sezione 3.1.3.2.

All'interno del package `db.operations` del modulo, troviamo le classi che si occupano di interagire con il database, per poter effettuare le operazioni richieste dai clients, le operazioni sono state divise in base all'entità su cui verranno eseguite, il che ha reso più semplice non solo la gestione del codice ma anche la definizione dell'interfaccia del servizio, sviluppata in seguito.

Le diverse classi possono sfruttare i metodi esposte dalle altre, per verificare la consistenza delle operazioni richieste, ad esempio: quando è necessario inserire un nuovo partecipante ad uno spazio, la classe `ParticipantOperationsImpl`, all'interno del metodo `insertParticipant`, sfrutterà le operazioni esposte dalle classi `UserOperations` e `SpaceOperations` (figura 9), per verificare che l'utente sia già registrato e che lo spazio esista, nel caso in cui una delle due condizioni dovesse essere falsa, viene lanciata un'eccezione con opportuno messaggio che spiega l'errore, tale eccezione verrà sfruttata in seguito dal servizio per rispondere al Client con un relativo messaggio di errore.

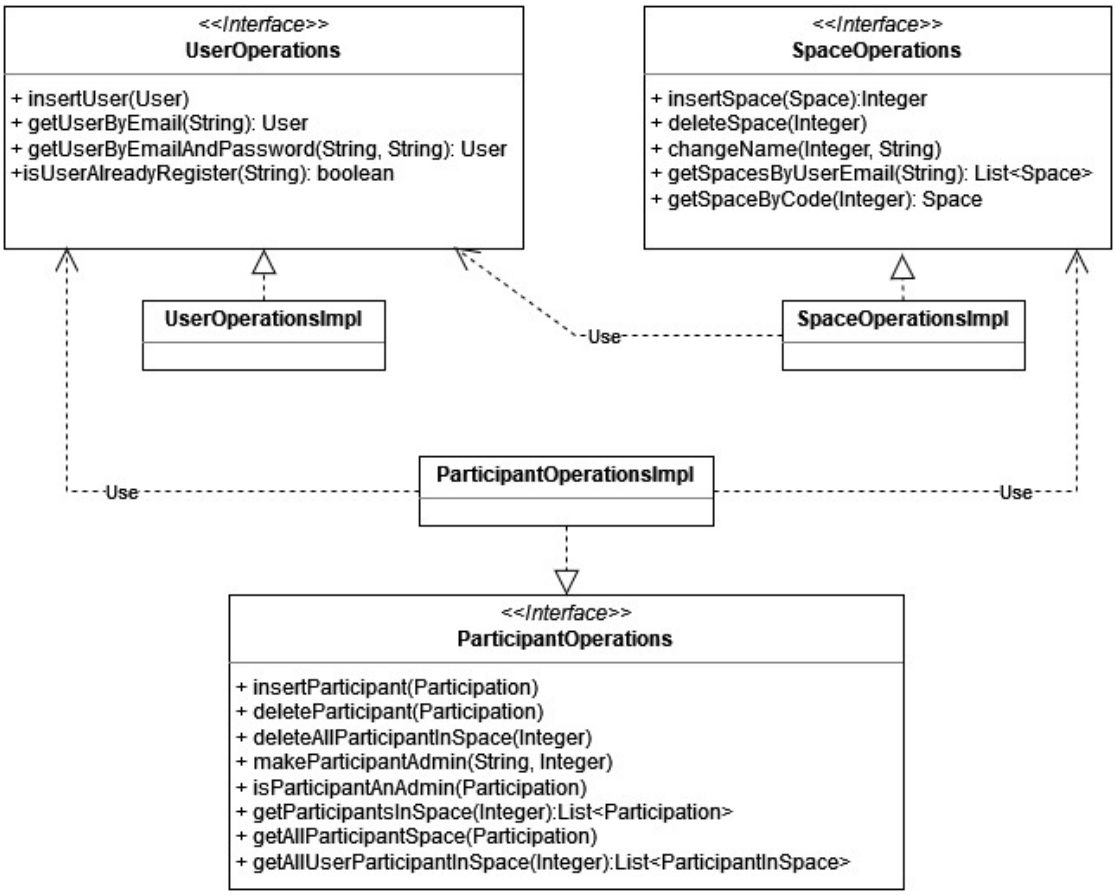


Figura 9: Diagramma delle classi: UserOperations, SpaceOperations e ParticipantOperations

3.1.4.2 Struttura modulo Service

Il modulo `Service` del Server è il modulo più importante, definisce l'API del Server stesso e espone il servizio ai clients.

Abbiamo detto che il funzionamento dell'applicazione si basa sul meccanismo di *Remote Procedure Call (RPC)*, dove i clients inviano una richiesta al Server e attendo la risposta, in ogni caso, le richieste: sono dirette verso un `URL`, specificano un metodo, cioè l'operazione che deve essere eseguita sul Server e la risposta restituisce uno **status code** con un significato standard, a seconda del protocollo che viene utilizzato, nel nostro caso HTTP. In particolare, HTTP, definisce metodi per poter effettuare le operazioni *CRUD (Create, Read, Update, Delete)* e sono:

- **POST**: consente di effettuare l'operazione di Create
- **GET**: richiama l'operazione di Read

- **PUT**: effettua l'operazione di Update/Replace
- **PATCH**: effettua l'operazione di Update/Modify
- **DELETE**: consente di effettuare l'operazione di Delete

Il Server consentirà di effettuare queste operazioni sulle diverse entità dell'applicazione, esponendo delle rotte, con qui questi metodi possono essere richiamati. Una rotta quindi, altro non è che una possibile operazione che il servizio può esporre ai clients per manipolare le risorse; la Web API del servizio consiste proprio nella dichiarazione delle sue rotte. Per descrivere formalmente l'API del Server, si è deciso di utilizzare lo standard OpenAPI e il tool Swagger.

All'interno del modulo, nel package `service`, troviamo un package per ogni entità del database su cui si va ad agire e per ognuna di queste è stata definita una classe `Api` che specifica le operazioni che verranno richiamate sul database e un `Controller`, che invece espone l'API e i metodi che potranno essere richiamati, i quali agiranno su una risorsa di riferimento, definendo anche le rotte che i clients dovranno specificare, per inviare le loro richieste.

All'interno dei `Controller` troviamo la definizione dei metodi: GET, PUT, POST e DELETE, che quando richiamati invocheranno i metodi della classe `Api`, per l'esecuzione delle operazioni, che a loro volta richiameranno i metodi della classe `Operations` del modulo `Db` di riferimento, ad esempio se si sta agendo sui turni si richiameranno i metodi della classe `ShiftOperations` del modulo `Db`.

Una volta effettuata l'operazione richiesta, il `Controller` si occuperà di formulare il pacchetto che dovrà essere inviato in risposta al Client, per fare questo, i `Controller` ereditano da una classe astratta `AbstractController`, che implementa i metodi per serializzare l'oggetto della risposta, sfruttando i `Serializer`, definiti nel modulo presentation.

La classe `Api`, invece, oltre che a richiamare i metodi per effettuare le operazioni sul database, si occupa anche di gestire le eventuali eccezioni che possono essere scaturite durante l'esecuzione dell'operazione e impostare la risposta con relativo codice di errore.

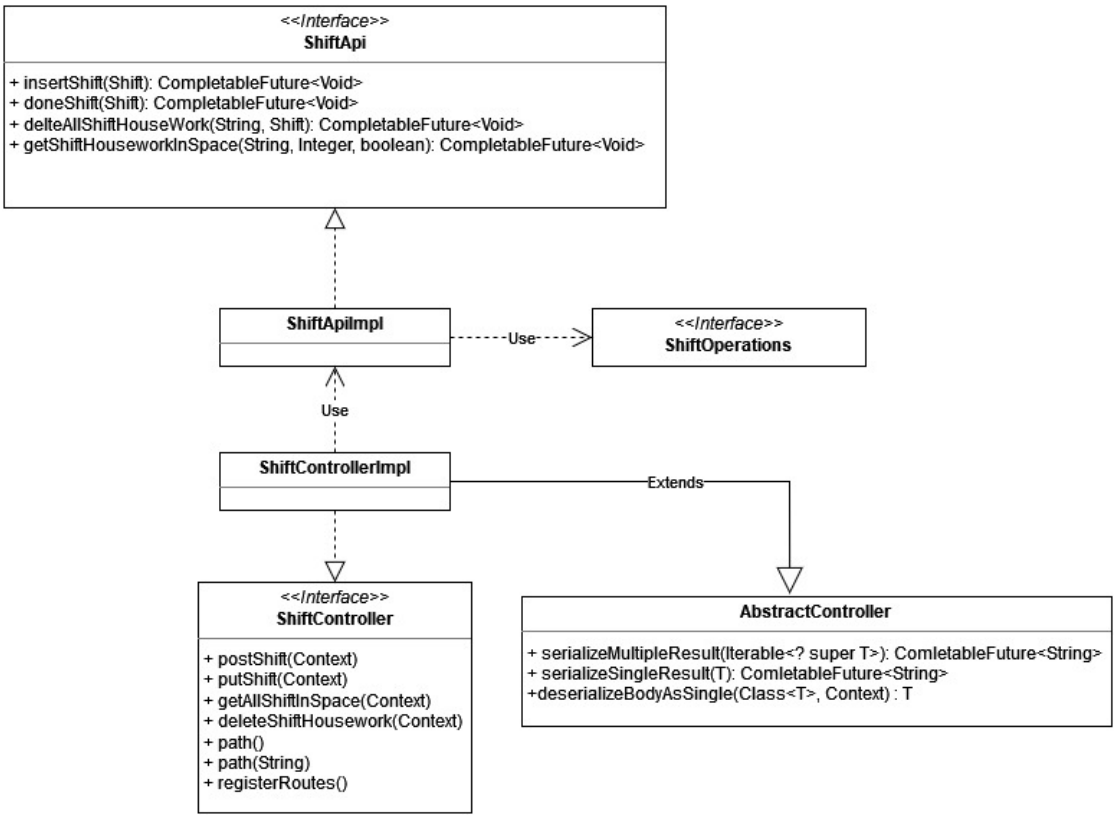


Figura 10: Diagramma delle classi: ShiftApi e ShiftController

Sempre all'interno del package `service`, troviamo la definizione della classe `SuhService`, che detiene il `main` del programma e si occupa di istanziare ed impostare il Server sfruttando la libreria `Javalin`. Questa classe presenta due campi: `port`, specifica la porta su cui il Server si mette in ascolto di nuove richieste e `server`, che invece rappresenta l'istanza del Server, a cui verranno registrate le rotte e in particolare, nel costruttore della classe, vengono utilizzati i `Controller` precedentemente definiti, per ottenere le rotte da registrare.

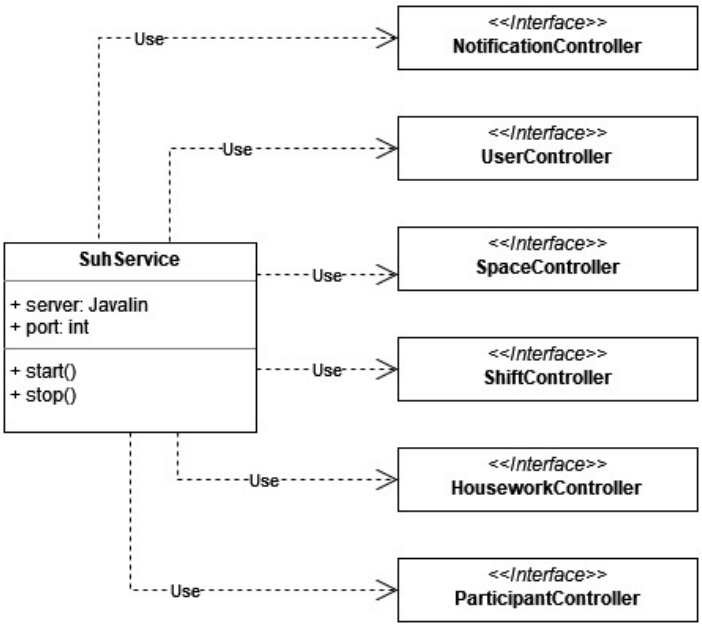


Figura 11: Diagramma delle classi: SuhService e Controllers

3.1.5 Struttura del Client Mobile

Il Client Mobile realizzato consiste in un applicazione per dispositivi Android che presentino una versione del sistema superiore alla 7.0. Per la realizzazione del Client si è cercato di adottare un approccio che consentisse la sua estensibilità per successive aggiunte future, cercando quindi di adottare le scelte di design migliori per il raggiungimento di questo scopo.

L'implementazione del Client, sfrutta tre diversi moduli del progetto, due dei quali comuni al Server, `Presentation` e `Entity`, descritti nella precedente sezione 3.1.3. Di seguito pertanto verrà riportata la descrizione del modulo `MobileApp`, il quale rappresenta il componente principale per il Client mobile, che racchiude l'applicazione Android realizzata.

3.1.5.1 Struttura modulo MobileApp

Un'applicazione Android è composta da più componenti che possono essere: Activity, Fragment, Service, Content Provider e Broadcast Receiver. Gli utenti spesso interagiscono con più app in un breve periodo di tempo e il sistema operativo potrebbe distruggere alcuni processi per liberare le risorse e fare spazio a quelli nuovi. È possibile che i componenti dell'applicazione vengano avviati in ordine diverso o che il sistema operativo possa distruggerli in qualsiasi momento. Quindi non si dovrebbe conservare nei componenti dell'applicazione i dati e lo stato dell'applicazione. Considerando tutto ciò, l'architettura dell'applicazione deve essere progettata in modo da separare al meglio gli aspetti dell'interfaccia con la logica e la gestione dei dati dell'applicazione.

Android Developers consiglia di separare i concetti in due livelli principali[2]: il livello dell'interfaccia utente **UI Layer** e il livello dei dati **Data Layer**, come raffigurato nella figura 12. Il livello dell'interfaccia utente si occupa di visualizzare i dati dell'applicazione sullo schermo e di gestire l'interazione con l'utente. Questo livello comprende:

- **UI elements**, ovvero tutto ciò che viene visualizzato dall'utente e con cui può interagire.
- **State holders**, mantengono lo stato e la logica correlata all'interfaccia utente e si occupano di richiedere il caricamento dei dati necessari per la visualizzazione.

Mentre il livello dei dati contiene tutte le informazioni dell'applicazione ed è costituito da un insieme di *repositories* che possono includere dei *data sources*:

- **Repositories** si occupano di estrarre i dati dai *data sources* per poi esporli al resto dell'applicazione.
- **Data Sourcers** rappresentano l'origine dei dati. Questi possono essere in remoto, ovvero richiesti dalla rete, oppure in locale, per esempio memorizzati in un database locale. Queste classi vengono utilizzate solo dai *repositories*, per rendere i moduli indipendenti.

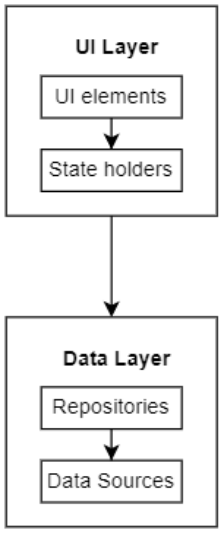


Figura 12: Livelli principali dell'app Mobile

L'applicazione Mobile è stata progettata seguendo gli aspetti appena descritti utilizzando il pattern architetturale **MVVM**, ovvero *Model-View-ViewModel*. Questa architettura è una variante del pattern MVC (Model-View-Controller), infatti consente di separare gli aspetti dell'interfaccia grafica dalla logica del dominio, ma è progettato appositamente per semplificare la programmazione basata su eventi delle interfacce

utente. In questo modo l'interfaccia non deve occuparsi della logica e grazie all'indipendenza delle varie componenti, è possibile rinnovare l'interfaccia senza modificarne il comportamento. Le principali componenti dell'architettura sono:

- **Model**, rappresenta il dominio dell'applicazione e il punto di accesso ai dati, quindi gestisce la logica dell'applicazione e di elaborazione dei dati. Comprende quindi il livello dei dati rappresentato dai *repositories* e dai *data sources*.
- **View**, racchiude gli *UI Elements*, rappresenta quindi la User Interface con cui l'utente interagisce, si occupa di visualizzare i dati e di ricevere le interazioni dell'utente.
- **ViewModel**, è rappresentato dagli *State holders*, fungono da centro di comunicazione tra il Model e la View, infatti interagiscono con il Model richiedendo i dati necessari alla View.

Riassumendo, la struttura utilizzata nell'applicazione è raffigurato nella figura 13.

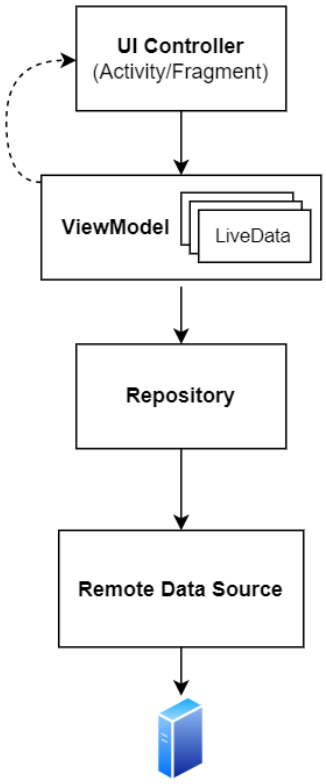


Figura 13: Struttura dell'app Mobile

L'applicazione Mobile è composta da più pagine, e quindi ha più interfacce, ognuno di queste è rappresentata da un **layout** definito con il linguaggio `XML` e una classe `UI Controller` associata. Il layout delinea la struttura dell'interfaccia, definisce tutti gli

elementi View presenti nella pagina, mentre la UI Controller, è rappresentata dalle Activity o dai Fragment. Un'Activity è l'*entry point* per l'interazione tra l'applicazione e l'utente, è il componente associato all'interfaccia che consente di gestire gli elementi della pagina. Un'Activity può contenere più interfacce, detti Fragment. Gli Activity e i Fragment hanno un proprio ciclo di vita, ed è possibile sapere in che fase di questo ciclo di vita ci troviamo: se il sistema ha creato l'attività, se sta arrestando o riprendendo un'attività o se si sta distruggendo il processo in cui risiede l'attività.

I **ViewModel** gestiscono i dati relativi all'interfaccia utente in modo consapevole rispetto al ciclo di vita. I ViewModel contengono più **LiveData**, ovvero classi contenitori di dati che possono essere osservate quando i dati vengono cambiati. In questo modo i componenti dell'interfaccia utente, vengono notificati se i dati sono cambiati e possono aggiornarsi di conseguenza. I LiveData gestiscono tutto automaticamente poiché sono consapevoli delle modifiche dello stato del ciclo di vita durante l'osservazione.

Successivamente abbiamo i **Repository**, sono delle classi che permettono di astrarre l'accesso ai dati. Questo significa che i ViewModel prendono i dati necessari per le interfacce attraverso il Repository. Come detto in precedenza, i Repository si occupano di richiedere i dati dalle varie sorgenti e in particolare essi vengono richiesti da remoto, ovvero dalla rete attraverso le rotte esposte dal Server. Ogni sorgente di dati è rappresentato con una entità **Remote Data Source**.

Per comprendere meglio questi concetti e la struttura, andiamo ad analizzare il diagramma delle classi per la gestione della faccende di uno spazio, raffigurato nella figura 14

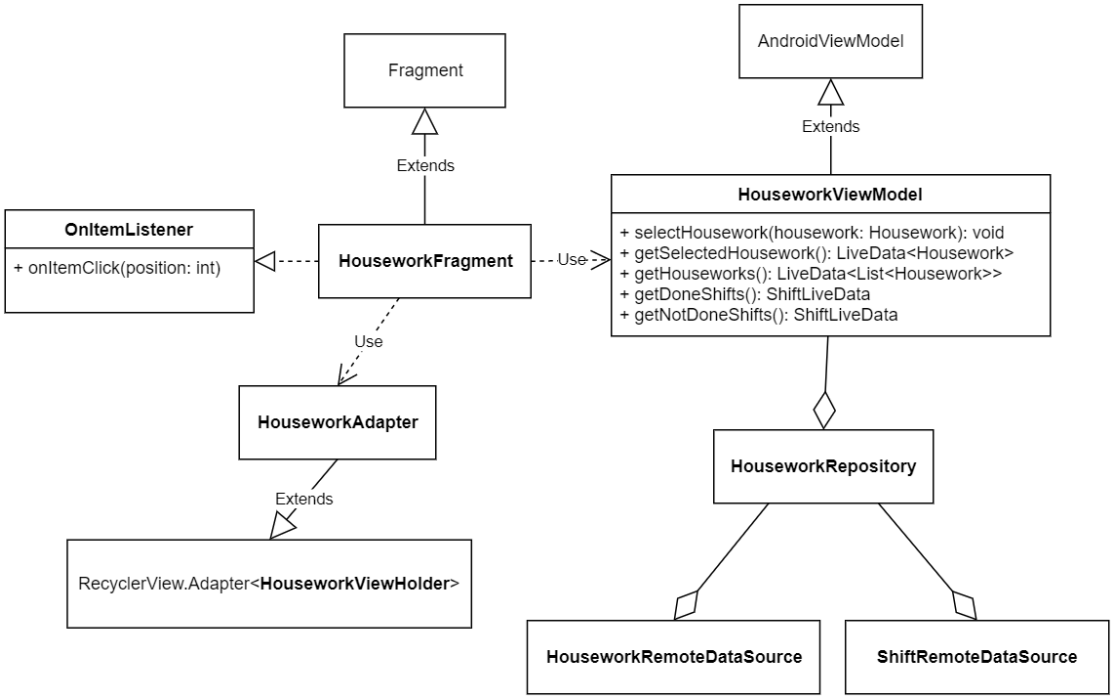


Figura 14: Diagramma delle classi: Gestione delle faccende

La classe `HouseworkFragment` estende dal componente `Fragment` di Android e rappresenta la View della pagina che si occupa di visualizzare la lista delle faccende associate ad uno spazio. Per gestire una lista di elementi in modo semplice ed efficiente si è fatto uso del componente **RecyclerView** di Android. Infatti basta definire l'aspetto di ogni elemento e la libreria RecyclerView crea dinamicamente gli elementi quando necessario; quindi si è creata una classe `HouseworkAdapter` che si occupa di gestire il layout e il comportamento specifico di ogni faccenda della lista. Tale classe estende dall'Adapter base di Android prendendo come input un `HouseworkViewHolder` che contiene i riferimenti ai componenti della View. La classe `HouseworkFragment` implementa anche l'interfaccia `OnItemListener` e presenta un metodo `onItemClick(position: int)` per gestire la selezione di una faccenda e per passare nella successiva pagina.

Per la gestione della logica della View e i dati relativi alle faccende si è fatto uso della classe `HouseworkViewModel`. Tale classe contiene il riferimento alla faccenda selezionata e si occupa di richiedere i dati necessari alla View richiamando la classe `HouseworkRepository`. Tale classe, come spiegato in precedenza, funge da livello di astrazione fra le diverse origini dati e il resto dell'applicazione. Quindi per la gestione delle faccende, la classe `HouseworkRepository` si compone di due origini dati:

- `HouseworkRemoteDataSource`, si occupa di effettuare richieste alla rete per richiedere i dati relativi alle faccende e per svolgere eventuali operazioni di aggiunta o modifica se richieste dall'utente,
- `ShiftRemoteDataSource`, si occupa di effettuare richieste alla rete per richiedere i dati relativi ai turni della faccenda.

3.1.6 Struttura del Client Web

Il Client Web si compone di un unico modulo `WebApp`, che risulta essere indipendente rispetto ai restanti moduli dell'applicazione, di fatto quest'indipendenza è dovuta per lo più all'utilizzo di differenti linguaggi e tecnologie per l'implementazione.

Per la realizzazione del Client, ci si è posti come obiettivi, di realizzare un'applicazione portabile che possa essere eseguita su differenti browser, senza problemi di visualizzazione e di mantenere separati i diversi aspetti presentazionali da quelli di funzionamento, in modo da rendere più semplice e agevole la gestione delle operazioni e eventuali modifiche.

Di seguito, verrà quindi descritto, l'ultimo modulo di cui si compone il progetto che rappresenta l'applicazione Web realizzata.

3.1.6.1 Struttura modulo WebApp

Il Client Web realizzato aderisce al modello a quattro livelli, dove sono previsti: un livello HTML, un livello di presentation, un livello di application logic e infine un livello di storage. Nel nostro caso il Client detiene i livelli HTML e presentation, mentre il Server, i livelli di application logic e storage, dove per application logic intendiamo la gestione e l'esecuzione delle operazioni richieste dall'utente, di fatti il Client non esegue direttamente queste operazioni, ma invia al Server, tutti i dati necessari per poterle eseguire.

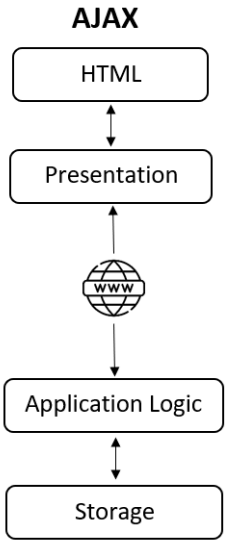


Figura 15: Modello a quattro livelli per le applicazioni Web

La struttura del Client quindi, aderendo al modello in figura 15, è stata realizzata separando i concetti presentazionali dall'applicazione vera e propria. Infatti il Client viene suddiviso in tre directory principali `html`, `css` e `js`, come possibile vedere dalla figura 16.

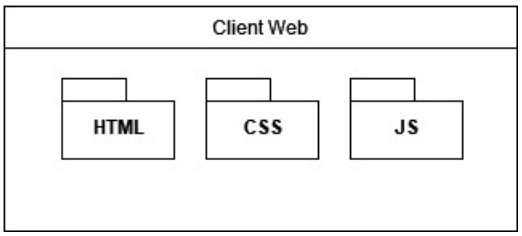


Figura 16: Struttura Client Web

All'interno della directory `html`, possiamo trovare i diversi file `.html`, realizzati attraverso il supporto di `HTML5`, che descrivono la struttura e gli elementi costituenti delle diverse pagine Web; essendo previste due categorie di utenti: admin e member, i quali possono svolgere azioni diverse all'interno dell'applicazione, in alcuni casi è stato necessario la realizzazione di pagine ad hoc, distinte in base al ruolo, per consentire a un utente, di eseguire tutte le diverse operazioni che li sono concesse, in base alla sua funzione all'interno dello spazio.

Nella directory `css`, invece, troviamo il foglio di stile `style.css`, realizzato attraverso gli elementi di `CSS3`, che si occupa di descrivere gli aspetti presentazionali, di alcuni componenti delle diverse pagine.

Infine nella directory `js`, troviamo i diversi sorgenti Javascript, utilizzati per gestire il contenuto delle diverse pagine HTML, ma anche per l'invio delle richieste HTTP al Server; per svolgere quest'ultima funzione è stata utilizzata la libreria `jQuery` e il supporto ad `AJAX` che essa offre, in particolare, l'utilizzo di queste librerie, non solo ha reso semplice e agevole la programmazione lato Client, ma ha anche reso possibile realizzare un sito Web dinamico, reattivo alle azioni compiute dall'utente e in grado di mostrare il contenuto senza che egli ricarichi manualmente la pagina.

Durante la navigazione dell'utente all'interno dell'applicazione, il Client si occupa di mantenere salvate alcune informazioni, per consentire lo svolgimento delle diverse operazioni e queste informazioni, vengono salvate attraverso l'utilizzo dell'API `SessionStorage`, che consente il mantenimento dei dati fintanto che l'applicazione è aperta, al termine della sessione infatti i dati verranno eliminati.

3.2 Comportamento

3.2.1 Comportamento del Server

Il comportamento del Server può essere descritto tramite il diagramma degli stati visibile in figura 17.

Da questo diagramma, possiamo notare che una volta avviato, il Server, si trova in un primo stato di inizializzazione dove vengono impostate le rotte del servizio, dopodiché quando il servizio viene attivato (evento start), il Server passa allo stato **Working**, che è uno stato composto dal quale è possibile uscire solamente quando il servizio viene fermato, cioè quando si verifica l'evento di stop, il quale provoca la terminazione del Server.

Come si può vedere sempre dalla figura 17, quando il Server si trova nello stato Working, si mette in ascolto di nuove richieste provenienti dai clients e quando arriva una nuova richiesta, passa allo stato **Handle Request**, dove esso provvede ad eseguire l'operazione che è stata demandata dal Client. Una volta terminata l'esecuzione si passa allo stato **Response processing**, in cui il Server elabora la risposta, la quale può essere di successo o di fallimento a seconda se l'operazione è andata a buon fine oppure no. Terminata l'elaborazione della risposta, questa viene inviata al Client e il Server ritorna in attesa di nuove richieste.

É opportuno far notare che dopo essere entrati nello stato Working, il Server è in grado di gestire più richieste contemporaneamente, provenienti da clients diversi; per ogni richiesta il comportamento mantenuto dal Server è lo stesso, descritto precedentemente.

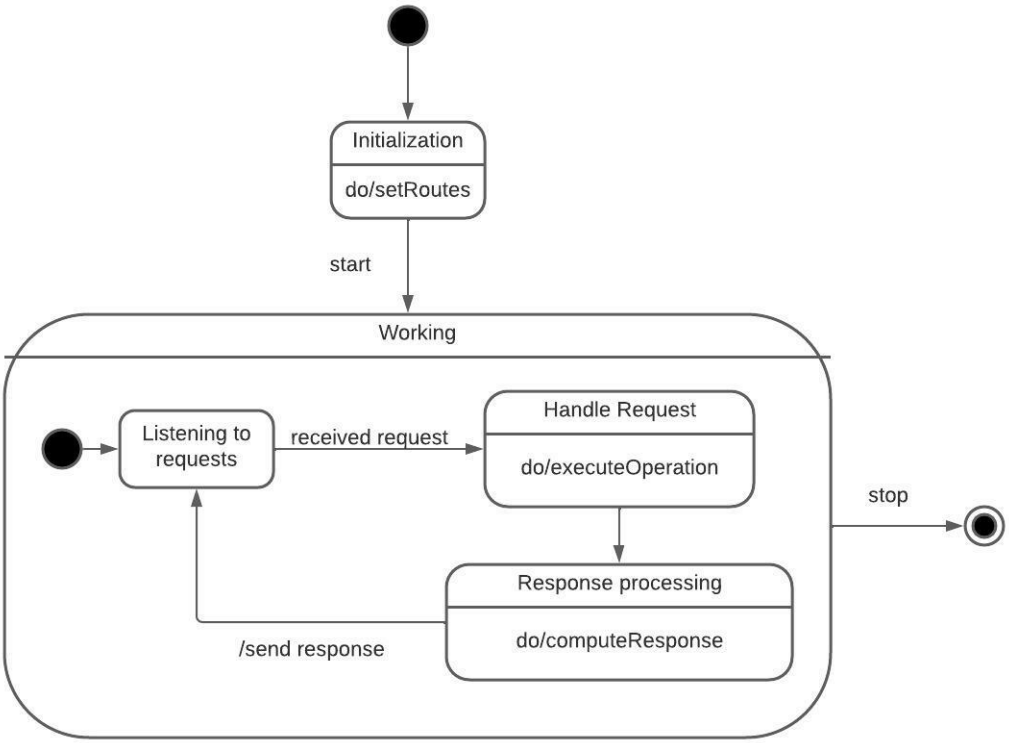


Figura 17: Diagramma degli stati: comportamento del Server

3.2.2 Comportamento del Client Web e Mobile

Le principali funzioni che vengono compiute lato Client sono **innescate** direttamente **dall'utente**. Infatti, è l'utente che mediante lo svolgimento di operazioni sull'applicazione, determinerà i cambiamenti di stato del Client. Inoltre, sempre in base alle operazioni che vengono svolte dall'utente il Client si occupa di inviare le richieste al Server, ad esempio: dopo che l'utente completa correttamente il form di login per l'accesso all'applicazione, il Client si occuperà di inviare una richiesta al Server per l'autenticazione. Quindi, si può affermare che l'utente sia l'elemento cardine della parte Client-side.

Sia Client Web che il Client Mobile, mentre l'applicazione è in esecuzione, mantengono salvate informazioni per la gestione delle operazioni, ad esempio: dopo che il login di un utente è avvenuto con successo, il Client mantiene salvate le informazioni dell'utente, sia per la visualizzazione dei dati che per l'invio delle richieste al Server.

Di seguito verranno illustrati gli stati principali del Client, che sono stati individuati e che l'utente potrà mutare attraverso la semplice interazione con il sistema.

- **Start**: questo è lo stato iniziale dell'applicazione, quando ci troviamo in questo stato l'utente potrà o effettuare il login o procedere alla registrazione. In entrambi i casi si verrà reindirizzati alla pagina principale del sistema.

- **Main**: questo è lo stato principale dell'applicazione da qui l'utente potrà decidere se creare un nuovo spazio, se unirsi ad uno spazio già esistente o accede ad uno degli spazi di cui fa già parte.
- **Space Creation**: se l'utente decide di creare un nuovo spazio, il Client entrerà nello stato di SpaceCreation, per ultimare la creazione effettiva di uno spazio il Client invia una richiesta al Server e attende la risposta, che se è di successo determinerà l'entrata nel successivo stato InsideSpace, se invece la risposta contiene un codice d'errore si rimarrà nello stato corrente e si mostrerà un messaggio di errore all'utente. Quando l'utente crea un nuovo spazio, di default partecipa a tale spazio con il ruolo di amministratore.
- **Space Join**: se invece l'utente decide di unirsi a uno spazio esistente, il Client entrerà nello stato di JoinSpace, per ultimare la partecipazione di un utente ad uno spazio, anche in questo caso, il Client invia una richiesta al Server e attende la risposta, che se è di successo determinerà l'entrata nel successivo stato InsideSpace, se invece è di fallimento si rimarrà nello stato corrente e si mostrerà un messaggio di errore all'utente.
- **Inside Space**: il Client entra in questo stato quando l'utente accede ad uno spazio, si tratta di fatto di uno stato composto i cui dettagli è possibile vedere dalla figura 19.
- **ShowNotification**: nel momento in cui l'utente decide di visualizzare le notifiche relative agli spazi a cui appartiene, il Client entrerà nello stato ShowNotification. L'utente può fare accesso a questo stato sia dallo stato Main (principale), oppure dallo stato InsideSpace.

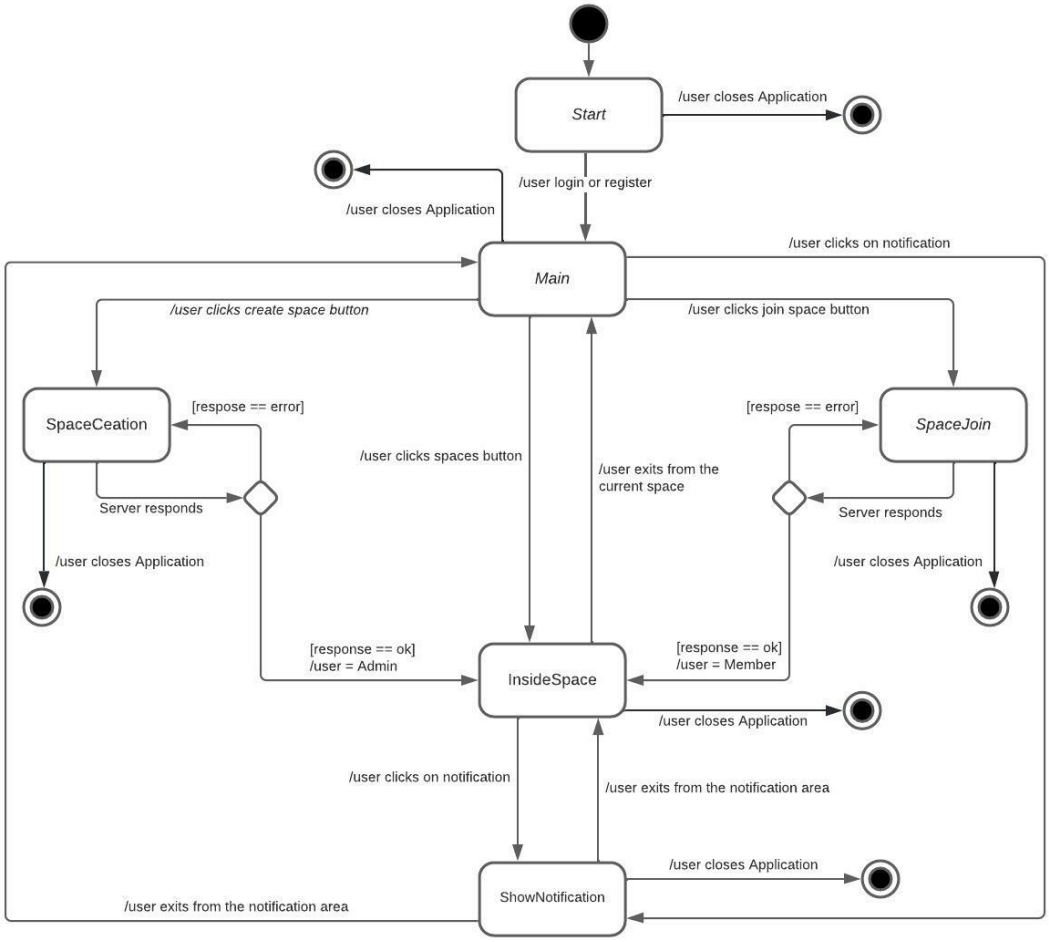


Figura 18: Diagramma degli stati: comportamento del Client

Va inoltre specificato che una volta eseguito l'accesso l'utente può entrare direttamente all'interno di uno spazio, cioè nello stato InsideSpace e poter compiere numerose azioni in base al ruolo ricoperto al suo interno. Allo stesso tempo, dallo stato InsideSpace egli potrà anche tornare allo stato principale Main, ogni qual volta lo desideri.

Infine, come possibile vedere dalla figura 18 l'utente potrà continuare a navigare fra le rispettive pagine del sistema fin quando non decide di chiudere l'applicazione.

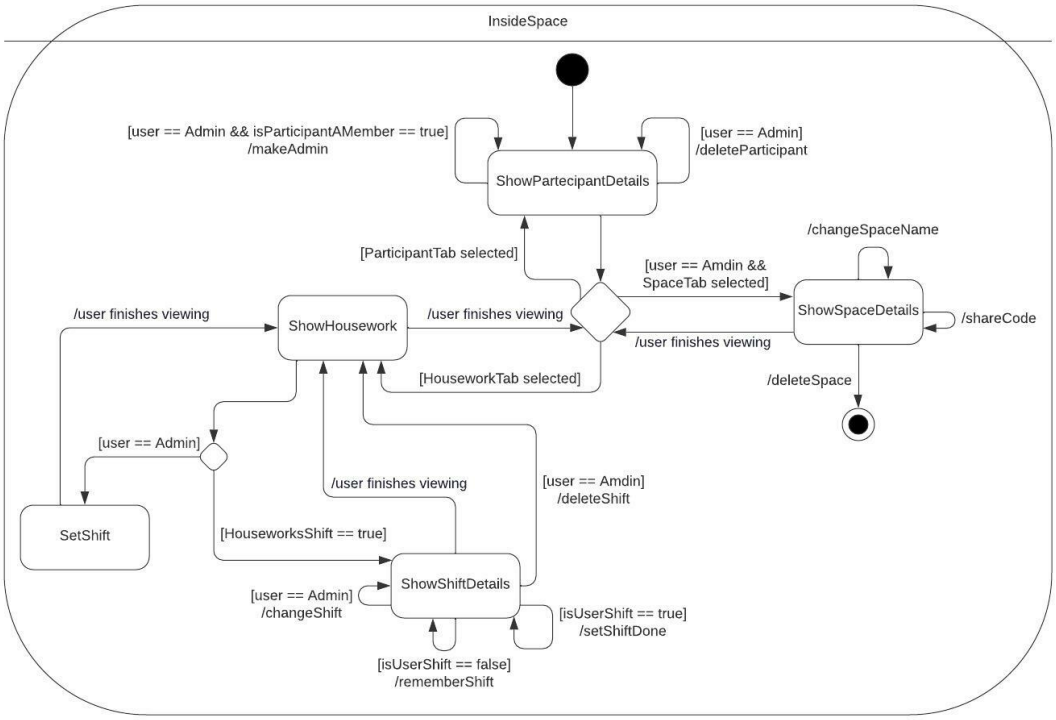


Figura 19: Diagramma degli stati: stato composto Inside Space

La figura 19, rappresenta la composizione dello stato InsideSpace, che mostra quali sono i possibili stati che il Client raggiunge, quando un utente copie delle operazioni dentro a uno spazio, ricordando che se un utente possiede il ruolo di amministratore potrà compiere delle azioni aggiuntive rispetto al membro.

Se l'utente decide di voler visionare le informazioni degli altri partecipanti dello spazio, il Client entra nello stato **ShowParticipantDetails**, all'interno di questo stato se un utente è amministratore può decidere di cambiare il ruolo di un utente da membro ad amministratore oppure rimuovere un partecipante.

Se invece l'utente decide di visualizzare le faccende, il Client entrerà nello stato **ShowHousework**, inoltre se l'utente è un admin, può decidere di settare i turni per una nuova faccenda. Se invece, i turni per una determinata faccenda sono già stati definiti, e l'utente decide di volerne visualizzare i dettagli, il Client entrerà nello stato **ShowShiftDetails**; all'interno di questo stato, nel caso l'utente sia un admin, egli potrà decidere di cancellare i turni assegnati, oppure di modificarli. Infine, sempre all'interno di questo stato, se è il momento dell'utente di svolgere il suo compito, egli potrà settare il turno come svolto, altrimenti può ricordare al partecipante incaricato, di svolgere il proprio turno.

Infine, Se l'utente è un amministratore, quando il Client si trova all'interno dello stato **InsideSpace**, può decidere di visualizzare le informazioni di uno spazio, entrando quindi

nello stato ShowSpaceDetails dove è possibile modificare il nome dello spazio, eliminarlo o condividere il codice identificativo.

3.3 Interazione

Abbiamo detto che le entità del nostro dominio sono essenzialmente quattro: Server, Client Mobile, Client Web e DatabaseMySQL.

Per comprendere come queste diverse entità interagiscono fra loro guardare la figura 20, rappresentante un diagramma di sequenza, che ci mostra le interazioni fra le diverse entità per la registrazione di un nuovo utente; per semplicità, avendo il Client Web e il Client Mobile lo stesso tipo di comportamento, si è rappresentato un'unica entità Client.

Come si può vedere l'interazione parte dal Client, il quale invia una richiesta HTTP `POST` al Server, chiedendoli di inserire un nuovo utente. Il Server prima di effettuare l'operazione di inserimento vera e propria, controlla che i dati passati siano corretti, nel caso in esame, verifica che l'utente non sia già stato registrato. In caso l'utente risulti già essere inserito, al Client viene restituito un messaggio d'errore, altrimenti, si procede all'inserimento dell'utente, in cui sostanzialmente, il Server richiede al database di eseguire una query che effettua l'inserimento, con i dati passati dal Client.

Se l'operazione va a buon fine il Server invia un messaggio di successo al Client, senza restituire alcun dato.

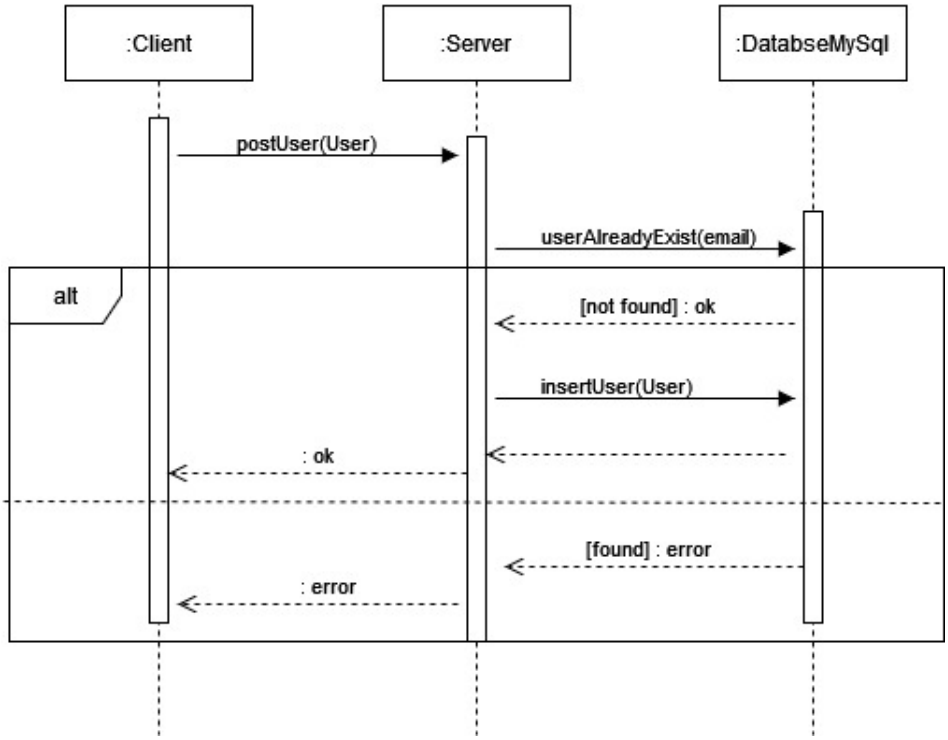


Figura 20: Diagramma di sequenza: interazione fra le diverse entità per la registrazione di un utente

Per vedere un esempio di interazione in cui invece il Server restituisce un risultato al Client, guardare la figura 21. Come possiamo vedere dal diagramma di sequenza, l'interazione è sempre iniziata dal Client, che in questo caso richiede al Server di inserire un nuovo spazio, sempre tramite l'utilizzo del metodo HTTP `POST`. Anche in questo caso, il Server, prima di effettuare l'effettivo inserimento, controlla che i dati passati siano corretti, in particolare verifica che l'utente che vuole creare lo spazio, sia correttamente registrato all'applicazione. Nel caso in cui l'utente non sia registrato, viene restituito un messaggio di errore al Client, altrimenti il Server effettua l'operazione di inserimento, richiedendo al database, di eseguire un'apposita query. Al termine dell'operazione di inserimento, il Server recupera il codice dello Spazio, inserito automaticamente dal database e lo restituisce in risposta all'utente, indicandogli che l'operazione è andato a buon fine.

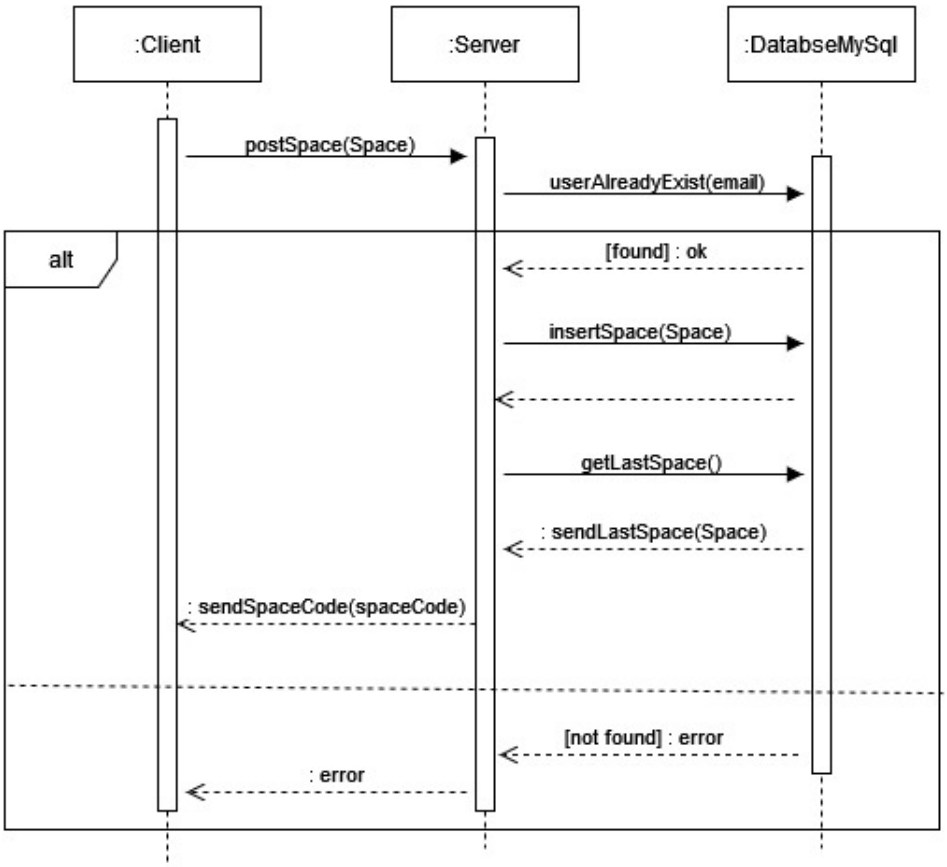


Figura 21: Diagramma di sequenza: interazione fra le diverse entità per inserimento di uno spazio

Dagli esempi precedenti quindi, possiamo concludere che:

- l'interazione parte sempre dal Client che richiede di eseguire una specifica operazione,
- il Server prima di eseguire l'operazione vera e propria, può effettuare altre operazioni sul database, verificando che i dati passati siano corretti e che l'operazione che si sta richiedendo sia valida,
- l'interazione si conclude sempre con il Server che invia una risposta di successo o fallimento al Client per l'operazione richiesta.

Una volta che il Client ha ricevuto la risposta da parte del Server, può decidere di continuare a effettuare richieste interrogando nuovamente il Server, oppure no (per le diverse operazioni che possono essere effettuate si guardi la sezione 4.2).

4 Dettagli implementativi

All'interno di questo capitolo, verranno discusse le tecnologie utilizzate e alcuni dettagli implementativi delle diverse entità che sono state realizzate, cercando di spiegare, dove necessario, come mai sono state effettuate determinate scelte invece che altre.

4.1 Tecnologie utilizzate

MySQL MySQL è un *Database Management System* relazionale, open-source. Esso utilizza il linguaggio SQL, come formalismo per indicare al *DBMS*, le operazioni da svolgere su uno specifico database. Tramite SQL è possibile, infatti, gestire le operazioni di tipo *CRUD* su un database: inserimento, lettura, modifica e cancellazione dei dati. Le caratteristiche principali di questo *DBMS* sono: ampia capacità di integrazione con i principali linguaggi di programmazione e ambienti di sviluppo, ottima efficienza nella gestione dei dati e infine l'offerta di tutte le diverse funzionalità dei *DBMS*.

Per il progetto, MySQL è stato utilizzato per la gestione del database di riferimento dell'applicazione.

Gradle Gradle è un *build automation tool open-source*, JVM-based, che consente di automatizzare la costruzione dei progetti Java; esso rende semplice anche l'impostazione di diverse librerie, che possono essere utilizzate all'interno del progetto, senza la necessità di includere i `.jar`, ma specificando direttamente delle dipendenze da un *repository* remoto. Gradle consente anche la gestione di un progetto costituito da diversi moduli e delle diverse dipendenze che vi possono essere fra questi.

Per il progetto si è fatto utilizzo di Gradle per la gestione dei diversi moduli.

Gson Gson è una libreria che può essere utilizzata per convertire gli oggetti Java nella loro rappresentazione `JSON`, essa può essere anche utilizzata per convertire una stringa `JSON`, nell'equivalente oggetto Java. Gli obbiettivi di Gson, come riporta la documentazione[5] sono: fornire dei meccanismi semplici da utilizzare e dei costruttori per convertire gli oggetti Java in `JSON` e viceversa, consentire la conversione di oggetti Java immutabili preesistenti, nel loro formato `JSON` e viceversa, consentire rappresentazioni custom degli oggetti, supportare oggetti arbitrariamente complessi e infine generare un output `JSON`, che sia compatto e leggibile.

Nel progetto, si è fatto uso di Gson nel modulo `Presentation`, per la realizzazione dei Serializer e Deserializer degli oggetti.

Javalin Javalin è un *lightweight Web framework*, per Kotlin e Java, progettato per essere il più semplice possibile, infatti per il suo utilizzo, è necessario apprendere davvero pochi concetti e mette a disposizione tutti gli elementi necessari per la realizzazione di un servizio Web. Cosa significa *lightweight*? Diciamo che Javalin è *lightweight*, in quanto costituito da poche linee di codice, Javalin infatti, ha una dimensione molto ridotta rispetto ad altri *framework*.

In questo progetto, Javalin è stato utilizzato per la realizzazione del Server.

Docker Docker come riporta la documentazione [3], è una piattaforma open-source per lo sviluppo, il rilascio e l'esecuzione di applicazioni. Esso consente di separare le applicazioni dall'infrastruttura dell'host, in modo da poter fornire il software rapidamente.

Docker consente di racchiudere un'applicazione in un ambiente isolato chiamato container. È possibile eseguire più container isolati all'interno di uno stesso host. I Container sono *lightweight* e pensati per fornire solo il necessario ad eseguire l'applicazione che essi contengono.

All'interno del progetto si è fatto uso di Doker per poter effettuare la containerizzazione del Server, come descritto nella sezione 4.2.2.

jQuery JQuery come riporta la documentazione: è una libreria Javascript ricca di *feature*, veloce e di piccole dimensioni [7]. JQuery, rende semplice la realizzazione di funzionalità come la navigazione del DOM, la gestione di eventi, le animazioni e `AJAX`, attraverso un'API semplice da utilizzare, che può essere impiegata anche per lo sviluppo su diversi browser.

> "With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript"[7].

Per il progetto si è fatto utilizzo della libreria jQuery e del supporto ad `AJAX`, che essa fornisce, per la realizzazione del Client Web.

Bootstrap Bootstrap, attualmente, è considerato il più efficiente e più utilizzato *framework* per la realizzazione di interfacce Web, la sua più grande qualità è quella di rendere semplice la realizzazione di siti Web *responsive*.

Bootstrap fornisce una raccolta di strumenti grafici, stilistici e di impaginazione che possono essere utilizzati per la resa grafica dell'applicazione. Si tratta di un *framework* molto versatile in quanto è compatibile con tutte le ultime versioni dei principali browsers.

All'interno del progetto si è utilizzato Bootstrap, per la gestione degli aspetti presentazionali del Client Web.

Volley Volley è una libreria HTTP fornita da Google che consente di gestire le funzionalità di networking delle applicazioni Android.[6] Volley si integra facilmente con qualsiasi protocollo e fornisce supporto per le stringhe, le immagini e `JSON`. L'utilizzo di Volley è particolarmente consigliato per le operazioni di tipo *RPC* utilizzate ad esempio per popolare un'interfaccia utente nel client. Infatti permette di gestire le richieste in modo asincrono, consente di gestire in automatico le richieste di rete e offre la possibilità di creare connessioni multiple.

La libreria è stata utilizzata all'interno del client Mobile per la gestione delle richieste al Server.

Android Espresso Espresso è un *framework* usato per la realizzazione di test automatizzati dell'interfaccia utente di un'applicazione Android.[1] I test Espresso vengono eseguiti direttamente su un dispositivo fisico o su un emulatore, dove viene simulato il comportamento dell'utente. Infatti esso sincronizza automaticamente le azioni di test con l'interfaccia utente dell'applicazione e garantisce, inoltre, che le attività vengano avviate prima dell'esecuzione dei test lasciando che il test attenda, fino al termine delle attività in background osservate.

Tale *framework* è stato utilizzato per testare alcune interfacce del client Mobile che verranno approfondite nella sezione 5.1.2.

4.2 Dettagli implementativi Server

Per quanto riguarda il Server, si vuole porre l'attenzione su due aspetti riscontrati nella realizzazione, il primo è relativo alla gestione del meccanismo di sicurezza *CORS*, messo in campo dal Client Web e il secondo invece è relativo alla containerizzazione del Server che è stata realizzata mediante l'utilizzo di Docker.

Inoltre, le classi e le interfacce realizzate, sono state arricchite con apposita documentazione `Javadoc`, ed è possibile visualizzare l'API del servizio realizzato al seguente link: `https://app.swaggerhub.com/apis/AnnaVitali/SettingUpHome/0.1.0`

4.2.1 Gestione Cross Origin Resource Sharing

All'interno del modulo `Service` del Server, nelle classi di implementazione dei `Controller`, vengono aggiunte all'header del pacchetto creato, delle informazioni per la gestione dei vincoli di *CORS (Cross Origin Resouce Sharing)*.

In particolare il *Cross Origin Resource Sharing* è un meccanismo basato sull'intestazione HTTP che consente a un Server di indicare qualsiasi origine (dominio, schema o porta) diversa dalla propria, da cui un browser dovrebbe consentire il caricamento delle risorse.

Per i metodi delle classi `Controller` quindi, a seconda dei metodi HTTP richiamati, viene impostato un header per il pacchetto, che consente a un browser di inviare i dati da una qualsiasi origine per il metodo indicato, come è possibile vedere dal listato 1.

```
public class SpaceControllerImpl extends AbstractController implements
    SpaceController {

    @Override
    public void postSpace(Context context) throws HttpResponseException {
        SpaceApi api = new SpaceApiImpl();
        Space space = this.deserializeBodyAsSingle(SpaceImpl.class, context);
        CompletableFuture<String> spaceData = api.insertSpace(space)
                .thenCompose(this::serializeSingleResult);

        context.header( "Access-Control-Allow-Origin", "*" );
        context.header( "Access-Control-Allow-Methods", "POST" );
```

```
        context.contentType("application/json").future(spaceData);
    }
}
```

Listato 1: Aggiunta informazioni all'header del pacchetto per gestione CORS

Va inoltre specificato che tale soluzione è stata utilizzata, in quanto per motivi di sicurezza i browser limitano le richieste *cross-origin*, inviate da script, in particolare l'API `XMLHttpRequest`, utilizzata lato Client Web, segue la stessa politica di origine; ciò significa che un'applicazione Web che utilizza tale API può solo richiedere risorse dalla stessa origine, da cui l'applicazione è stata caricata, a meno che la risposta da altre origini non includa le intestazioni *CORS* giuste.

Inoltre, per i metodi di richiesta HTTP che possono causare effetti collaterali sui dati del Server (in particolare, i metodi HTTP diversi da GET, o POST), la specifica richiede che i browser inviino delle richieste *"preflight"* prima della richiesta vera e propria, sollecitando i metodi supportati dal Server con il metodo di richiesta HTTP `OPTIONS`. Per consentire l'operazione la richiesta `OPTIONS` deve essere approvata dal Server, consentendo così che la richiesta effettiva possa essere inviata.

Di conseguenza per i `Controller`, che presentano metodi `PUT` e `DELETE`, sono state aggiunte delle rotte per la gestione delle chiamate ai metodi `OPTIONS`.

```
@Override
    public void registerRoutes(Javalin app) {
        app.post(path("/"), this::postSpace);
        app.put(path("/{code}"), this::putSpace);
        app.delete(path("/{code}"), this::deleteSpace);
        app.options(path("/{code}"), this::handlePreflightRequest);
    }
```

Listato 2: Aggiunta rotte per la gestione dei metodo Options

4.2.2 Containerizzazione

Per consentire una maggiore *reliability* e *aviability* del Server, si è deciso di utilizzare Docker per la containerizzaizone e in particolare, di utilizzare i servizi messi a disposizione da Docker Swarm e Docker Compose.

L'obiettivo che si vuole raggiungere è quello di riuscire a creare più repliche del servizio del Server tramite l'utilizzo di Docker Swarm, sfruttando poi in fase di esecuzione il meccanismo di *load balancing* messo a disposizione dal servizio stesso, per far fronte alle diverse richieste provenienti dai clients.

Per fare questo, per prima cosa è stato realizzato un `Dockerfile` per la creazione dell'immagine del Server. Dopodiché sono stati creati due file `docker-compose`, uno che istanzia il servizio del database a partire dall'immagine `mysql`(listato 3) e l'altro invece che istanzia diverse repliche del Server tramite l'utilizzo di Docker Swarm (listato 4).

```
version: "3.9"

services:
  mysql:
    image: mysql
    container_name: mysql
    environment:
      MYSQL_ROOT_PASSWORD: password
    ports:
      - "3307:3306"
    networks:
      - service_network
networks:
  service_network:
    name: service_network
    driver: overlay
    attachable: true
```

Listato 3: Docker-compose per il servizio del database

Come si può vedere dal listato 3, all'interno del container non viene caricato nessun volume contenente la struttura del database, questo perché quando inizialmente si è provato a caricare il volume con i dati relativi al database, si è notato che le impostazioni del servizio settate mediante le variabili d'ambiente, non avevano alcun effetto, di fatti la documentazione di dockerHub afferma:

> "When you start the mysql image, you can adjust the configuration of the MySQL instance by passing one or more environment variables on the docker run command line. Do note that none of the variables below will have any effect if you start the container with a data directory that already contains a database: any pre-existing database will always be left untouched on container startup."[4]

Di conseguenza affinché la password possa essere settata correttamente, il database verrà costruito tramite l'esecuzione di un apposito script mysql all'interno di MySQL-Workbench, vedere la sezione 6 per maggiori dettagli.

```
version: "3.9"

services:
  server:
    image: server #image create from Dockerfile
    ports:
      - "10000:10000"
    networks:
      - service_network
    deploy:
      mode: replicated
```

```
    replicas: 2


networks:
  service_network:
    name: service_network
    external: true
```

Listato 4: Docker-compose per il servizio del Server

Un'altra domanda che potrebbe sorgere guardando questa realizzazione è: perché sono stati realizzati due `docker-compose` file anziché uno solo? Tale scelta è stata necessaria perché occorre assegnare un nome specifico al container relativo al database, il quale verrà utilizzato dal Server, per indicare l'indirizzo IP a cui connettersi per potersi collegare a MySQL Server. Questa funzionalità non poteva essere realizzata all'interno del `docker-compose` file del Server, in quanto Docker Swarm ha deprecato la possibilità di assegnare nomi arbitrari ai container, pertanto è stata necessaria la realizzazione di due file `docker-compose` separati.

Infine affinché i container creati possano comunicare fra loro, è stata realizzata una docker network mediante l'uso del driver `overlay`, tale network viene creata nel `docker-compose` del database, come si può vedere dal listato 3.

È stato utilizzato il driver `overlay`, in quanto esso, rispetto agli altri driver, è `swarm scope` di default, il che significa che una network creata mediante questo driver può essere acceduta anche da uno swarm, se invece fosse stata creata una rete mediante, ad esempio, il driver `bridge` essa avrebbe presentato uno scope `local` e lo stack dello swarm creato, non avrebbe potuto accedervi.

4.3 Dettagli implementativi Client Mobile

In questa sezione vengono illustrati gli aspetti rilevanti per la realizzazione dell'applicazione Mobile, descrivendo le scelte e i dettagli implementativi. Anche tutte le classi del Client Mobile sono state documentate con Javadoc.

4.3.1 Gestione delle Activity e Fragment

L'applicazione è composta da due Activity principali:

- `AuthenticationActivity` gestisce il processo di autenticazione dell'utente, comprende due Fragment: `LoginFragment` che gestisce l'accesso all'applicazione di un utente già registrato e `RegisterFragment` che permette la registrazione di un nuovo utente.

- `MainActivity` gestisce tutte le interfacce principali dell'applicazione come la visualizzazione delle notifiche, degli spazi dell'utente, dei partecipanti allo spazio, delle faccende e dei turni.

L'approccio seguito è quella di utilizzare più Fragment invece che Activity perché durante la navigazione dell'applicazione, è necessario gestire più dati i quali devono anche essere scambiati tra le diverse pagine. Per cui, per condividere i dati tra i vari Fragment della stessa Activity sono stati utilizzati i ViewModel che verranno trattati nella sezione successiva.

4.3.2 Gestione e condivisione dei dati tra Fragment

Come detto in precedenza Android gestisce i cicli di vita dei Activity e dei Fragment in base alle azioni dell'utente oppure ad eventi fuori dal controllo dell'applicazione e quindi il sistema operativo può decidere di distruggere o ricreare il controller UI. Ma se il sistema distrugge il controller dell'interfaccia utente, tutti i dati visualizzati nell'interfaccia utente verranno persi ed è necessario che la nuova Activity recuperi nuovamente tutti i dati. Ovviamente per dati semplici l'Activity può effettuare il loro recupero grazie al metodo `onSavedInstanceState()`, ma per dati di grandi dimensioni e dati che devono essere caricati, viene consigliato di gestire quest'aspetto con un nuovo componente ViewModel.

Architecture Components fornisce la classe di supporto **ViewModel** per il controller dell'interfaccia. Gli oggetti ViewModel vengono conservati automaticamente in modo che i dati siano disponibili alla ripresa o al passaggio dell'Activity o Fragment successiva.

Nel listato 5 è mostrato un esempio di creazione del ViewModel. In particolare `SpaceViewModel` deve estendere dalla classe base `AndroidViewModel` e fornire il contesto dell'applicazione.

Il ViewModel contiene degli oggetti particolari, contenitori di dati, chiamati `LiveData`, oppure `MutableLiveData` i quali sono oggetti che contengono dati che possono essere modificati. Il ViewModel contiene anche dei **Repository** da cui richiedere il caricamento dei dati. Gli oggetti `LiveData` e `MutableLiveData` possono essere osservati dai controller, attraverso la loro registrazione come osservatori. In tal modo, ogni volta che la lista si modifica, il controller viene notificato e quindi può aggiornare l'interfaccia.

```
public class SpaceViewModel extends AndroidViewModel {

    private final MutableLiveData<Space> selectedSpace;
    private ParticipantLiveData participantList;
    private SpaceRepository spaceRepository;

    public SpaceViewModel(@NonNull Application application) {
        super(application);
    }

    public LiveData<List<Space>> getSpaceList() {
        return this.spaceRepository.getSpaceLiveData();
    }
    //..
}
```

Listato 5: Esempio di implementazione del ViewModel

Per fare ciò, nelle Activity o nei Fragment interessati, tramite il `ViewModelProvider` è possibile richiedere l'istanza del ViewModel se è già stato creato dai precedenti controller, oppure richiedere una nuova istanza del ViewModel.

4.3.3 Gestione dell'invio delle richieste con Volley

Gli oggetti `Repository` presenti nei ViewModel rappresentano il punto di accesso ai dati. Tali classi contengono una o più `DataSource` con cui andare a richiedere i dati e nel nostro caso i dati vengono richiesti alla rete.

Le richieste vengono effettuate mediante la libreria `Volley` sviluppata da Google per la gestione di richieste HTTP. `Volley` si integra facilmente con qualsiasi protocollo e consente l'invio di stringhe immagini o `JSON`. Si utilizza `Volley` creando un `RequestQueue` e passandogli oggetti `Request`:

- `Request` esegue l'analisi delle risposte grezze e successivamente `Volley` si occupa di inviare la risposta analizzata al thread principale.
- `RequestQueue` gestisce i thread di lavoro per l'esecuzione delle operazioni di rete, la lettura e la scrittura nella cache e l'analisi delle risposte.

```
public void register(User user, Response.Listener<String> responseListener,
    Response.ErrorListener errorListener) {
    final String url = this.baseUrl + "users/register";
    final StringRequest stringRequest = new StringRequest(Request.Method.POST,
        url, responseListener, errorListener) {
        @Override
        public byte[] getBody() {
            return gson.toJson(user, UserImpl.class).getBytes();
        }
    };
    stringRequest.setTag(ActivityUtilities.OSM_REQUEST_TAG);
    requestQueue.add(stringRequest);
}
```

Listato 6: Esempio di richiesta con Volley

Nel listato 6 è rappresentato un esempio di richiesta con `Volley` per la registrazione dell'utente. La richiesta, in questo caso, è una richiesta di stringa ed è rappresentata con l'oggetto `StringRequest` a cui vengono passati: il metodo della richiesta, l'url della richiesta, il `listener` per gestire i dati nel caso di una richiesta effettuata con successo e il `listener` per gestire l'errore nel caso in cui la richiesta non sia andata a buon fine. In questo caso nel body del pacchetto vengono passate le informazioni dell'utente per richiedere la registrazione. L'entità utente viene convertita in una stringa in formato `JSON` attraverso l'utilizzo di `Gson`.

`Volley` purtroppo non fornisce direttamente una rappresentazione di una richiesta tramite `Gson`, ma permette di implementare dei tipi di richieste personalizzate. Questo è possibile estendendo una classe Request base. Quindi, in questo caso abbiamo che

la nuova classe `GsonRequest<T>` estenderà da `JsonRequest<T>` come è possibile vedere nel listato 7; successivamente per completare l'operazione, è necessario implementare il metodo astratto `parseNetworkResponse()` il quale prende come argomento la risposta e tale metodo, deve restituire un oggetto Response secondo il tipo della classe in caso di successo, 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;

    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;
    }

    @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));
        }
    }

    //..
}
```

Listato 7: Creazione di una richiesta personalizzata per gestire le richieste Gson

4.4 Dettagli implementativi Client Web

Di seguito verranno indicati alcuni dettagli implementativi del Client Web, i quali mettono in luce aspetti rilevanti delle tecnologie, che si è deciso di utilizzare per la realizzazione.

4.4.1 Utilizzo di AJAX per l'effettuazione delle richieste

Durante l'implementazione del Client Web si è fatto utilizzo della libreria `AJAX`, in quanto adatta allo sviluppo di applicazioni dinamiche e interattive. Per la programmazione

Client-side si è deciso di adottare Javascript, con il supporto della libreria jQuery. Tale libreria mette a disposizione il metodo `$.ajax()`, per effettuare delle richieste HTTP.

Inoltre, sempre attraverso Javascript è stato possibile condividere i dati fra le diverse pagine memorizzandoli nell'apposita variabile `sessionStorage`, la quale è una delle le principali API che i Browser mettono a disposizione, che però, permette l'archiviazione dei dati fintanto che la sessione resta attiva.

```
$.ajax({
    type: "POST",
    url: "http://" + host + ":" + port + "/users/register",
    data: JSON.stringify(formUser),
    crossDomain: true,
    dataType: "json",
    success: function(){
      window.location = "home.html";
    },
    error: function(jqXHR) {
      var responseText = JSON.parse(jqXHR.responseText);
      var msg = responseText.title;
      $('#msg').html(msg);
    }
});
```

Listato 8: Gestione del metodo `$.ajax()` per la registrazione di un nuovo utente

Come si può vedere il listato 8, mostra com'è possibile effettuare una richiesta di registrazione di un nuovo utente attraverso il metodo `$.ajax()`. Per effettuare la richiesta è necessario definire:

- **type**: il metodo HTTP utile alla richiesta. I valori assunti possono essere POST, GET, PUT o DELETE.
- **URL**: che rappresenta il percorso della risorsa.
- **data**: rappresenta i dati da inviare al Server. Occorre far notare che nel caso in cui la richiesta sia di tipo GET, non è possibile specificare questo elemento; in questo caso, nell'elemento data viene inserito l'oggetto `JSON`, rappresentante il nuovo utente da registrare.
- **crossDomain**: il quale viene impostato, in quanto l'API `XMLHttpRequest`, consente a un Browser, per motivi di sicurezza, di accedere solo a risorse che si trovano nello stesso dominio.
- **dataType**: viene utilizzato per specificare il formato dei dati, che ci si aspetta di ricevere dal Server, in questo caso il formato specificato è `JSON`.
- **success**: specifica la funzione eseguita in caso di successo della chiamata. Tale funzione può accettare tre parametri opzionali che sono: `data`, cioè i dati restituiti dal Server, i quali vengono formattati in base al `dataType`, `textStatus` la

quale è una stringa che descrive lo status della risposta e `jqXHR`, ovvero l'oggetto `XMLHttpRequest`.

- **error**: specifica la funzione che verrà eseguita nel caso in cui la risposta contenga un codice di errore, la quale può prendere i parametri: `jqXHR` che rappresenta l'oggetto `XMLHttpRequest`, `textStatus`, il quale può assumere i valori `timeout`, `error`, `abort`, o `parsererror` e `errorThrown`, cioè la parte testuale della risposta HTTP.

4.4.2 Utilizzo di Bootstrap per la gestione degli aspetti presentazionali

Per quanto riguarda la presentazione si è deciso di adottare il *framework* `Bootstrap`, il quale facilita di molto la gestione degli aspetti presentazionali.

`Bootstrap` mette a disposizione, oltre a diverse classi, che consentono di associare agli elementi uno stile preimpostato, una struttura a griglia, o *grid system*. Questo grid system, si compone di 12 colonne e viene utilizzato per la gestione della *responsiveness* delle pagine.

```
<div class="container">
     <div class="d-grid gap-5 col-6 mx-auto">
       <input class="btn btn-outline-primary btn-lg" type="button"
           onclick="location.href='newSpace.html'" value="Create New Space"/>
       <input class="btn btn-outline-primary btn-lg" type="button"
           onclick="location.href='joinSpace.html'" value="JoinSpace"/>
       <input class="btn btn-outline-primary btn-lg" type="button"
           onclick="location.href='spaces.html'" value="My Spaces"/>
     </div>
   </div>
```

Listato 9: Esempio di utilizzo di Bootstrap per la registrazione di un nuovo utente

Il listato 9, rappresenta un frammento di codice della pagina `home.html`, dove è possibile vedere l'utilizzo di alcune classi di Bootstrap, le quali hanno consentito una migliore organizzazione e visualizzazione del contenuto della pagina.

5 Autovalutazione/Validazione

Per verificare l'efficacia del prodotto realizzato come detto nella sezione 1.2, si è proceduto a testare le diverse componenti del sistema sia tramite test manuali, che mediante test automatici, nelle seguenti sezioni verranno definiti con maggiore dettaglio i test realizzati.

Naturalmente il testing effettuato ha come scopo anche di verificare il corretto comportamento dell'applicazione e 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 `Junit` che mediante test manuali, in particolare, sono stati eseguiti test automatici per il modulo `Db` e il modulo `Presentation`, manuali invece, per il modulo `Service`.

Occorre inoltre far notare che per lo sviluppo non è stato adottato un approccio di tipo *Test Driven Development*, i test sono stati implementati scrivendo prima il codice di produzione e poi quello di testing relativo, quindi ogni volta che si completava la stesura di un metodo si passava poi alla sua verifica mediante il testing.

È importante notare che per quanto riguarda la verifica del modulo `Db`, le operazioni vengano svolte in un certo ordine: prima vengono gestiti gli utenti, poi gli spazi, poi i turni, le faccende ecc. Questo perché, in base alla logica di funzionamento dell'applicazione, vi sono dei vincoli di consistenza che bisogna rispettare, ad esempio: non posso creare uno spazio senza un'utente creatore e non posso assegnare i turni senza uno spazio di riferimento.

TestDBConnection Questo test si occupa di verificare, l'instaurazione della connessione al database, prima di svolgere una qualsiasi operazione è infatti necessario ottenere la connessione al database creato.

TestUserOperations Il test si occupa di verificare le operazioni effettuate sugli utenti, come l'inserimento di un nuovo utente o il login, verificando in caso di errore dei dati inseriti, il corretto rilevamento di un'eccezione.

TestSpaceOperations Questo test si occupa di verificare le operazioni effettuate sugli spazi, è necessario dire che per poter effettuare operazioni sugli spazi occorre prima inserire gli utenti, che si occuperanno quindi di creare un nuovo spazio o partecipare a uno spazio esistente. Anche in questo caso il test verifica il corretto comportamento delle operazioni e in caso i dati passati non siano corretti, analizza il corretto lancio di un'eccezione.

TestParticipantOperations In questo test si analizzano le operazioni effettuate sui partecipanti, come per il precedente test anche qui, per poter testare le operazioni è necessario inserire prima gli utenti e gli spazi, a cui i diversi utenti registrati potranno poi afferire in veste di partecipanti. Il test verifica sia il corretto comportamento delle operazioni effettuate che il rilevamento di eccezioni.

TestHouseworkOperations Questo test verifica le operazioni effettuate sulle faccende, in particolare per l'inserimento di una nuova faccenda custom è necessario l'inserimento prima di uno spazio di riferimento, nonché dell'utente creatore dello spazio. Il test analizza il comportamento delle operazioni effettuate sia su `HouseWork` di tipo standard, che di tipo custom.

TestShiftOperations Il test si occupa di verificare le operazioni effettuate sui turni, analizzando il comportamento dell'applicazione, verificando che sia corretto e che effettui il lancio delle eccezioni quando opportuno. Per poter effettuare le operazioni sui turni è necessario prima inserire: gli utenti, gli spazi a cui afferiscono e le faccende di riferimento.

TestNotificationOperations Questo test verifica che il sistema invii correttamente le notifiche agli utenti quando necessario, in particolare le notifiche inviate sono sempre relative ai turni, quindi come operazioni preliminari viene effettuato l'inserimento degli utenti, degli spazi e dei turni.

TestPresentation Questo test che troviamo all'interno del modulo `Presentation`, si occupa di controllare il coretto funzionamento dei `Serializer` e `Deserializer` realizzati, in particolare, i metodi di testing sono stati divisi in base alla risorsa su cui viene effettuata la serializzazione e la deserializzazione e si verifica la corretta corrispondenza fra l'oggetto deserializzato e quello serializzato.

Test manuali Per testare il modulo `Service`, il quale si occupa della gestione delle diverse richieste provenienti dai Client, sono stati effettuati dei test manuali tramite l'utilizzo del tool Swagger. In particolare, grazie a Swagger, dopo aver definito l'API del servizio è stato possibile usufruire di un'apposita interfaccia grafica, la quale consente di visionare le diverse operazioni che possono essere richieste al Server, dando anche la possibilità di effettuare queste richieste e visualizzare la risposta del Server stesso.

Di fatto si è trovato questo strumento essere molto utile, per verificare il corretto comportamento del Server, ma non solo, anche per consentire ai diversi componenti del team addetti allo sviluppo dei clients, di avere un riscontro su come le richieste dovevano essere effettuate, quali dati dovevano essere passati e in che modo, evitando in caso di errori, molto lavoro di debugging per la formulazione corretta della richiesta.

5.1.2 Testing del Client Mobile

Per il lato Client Mobile sono stati creati dei test automatizzati e dei test manuali per verificare il corretto funzionamento dell'interfaccia utente (UI). In particolare si è utilizzato il *framework* di test Espresso fornito da Jetpack per la creazione dei test automatizzati. Come precedentemente spiegato, i test vengono eseguiti direttamente sul dispositivo Android, fisico o emulato, l'applicazione viene compilata e installata nel dispositivo e viene lanciato il test simulando le interazioni dell'utente e infine viene verificato il corretto comportamento dell'applicazione.

Per far funzionare correttamente i test automatici è necessario che il Server sia in esecuzione, e che il dispositivo Android su cui è presente l'applicazione sia connesso al Server, per sapere come svolgere queste operazioni si guardi la sezione 6.

Successivamente, per consentire la corretta esecuzione dei test, si consiglia di disattivare le animazioni del sistema nel dispositivo andando in Impostazioni Opzioni sviluppatore e disabilitare le seguenti tre impostazioni:

- Scala animazione della finestra
- Scala animazione transizione
- Scala durata animazione

Eseguiti i precedenti passi, è possibile avviare i test sul dispositivo. Se è la prima volta che vengono eseguiti i test, allora è necessario lanciare per primo il test `UserLoginAndRegisterTest` per completare la registrazione dell'utente, questo perché per prima cosa, è necessario l'inserimento di un utente per compiere poi le operazioni successive, verificate negli altri test, come ad esempio la creazione di uno spazio.

UserLoginAndRegisterTest Tale test si occupa di verificare l'accesso e la registrazione dell'utente all'applicazione. Il test tenta di accedere con le informazioni settate dell'utente e nel caso in cui l'operazione sia andata a buon fine, si verifica che le informazioni dell'utente mostrate nella pagina principale siano corrette. Altrimenti, se l'operazione fallisce, significa che l'utente non è ancora registrato al sistema, quindi si procede alla sua registrazione e al controllo delle informazioni dell'utente. Alla fine, si accede nuovamente con le credenziali dell'utente appena registrato, per verificare se l'operazione di registrazione vada a buon fine.

LoginEdgeCasesTest Tale test comprende tutti i casi estremi dell'accesso di un utente all'applicazione. Viene verificato se vengono visualizzati i messaggi d'errore corrispondente, nel caso in cui i campi non siano stati inseriti oppure non siano validi. Viene anche testato l'accesso di un utente inserendo una password sbagliata.

RegisterEdgeCasesTest Tale test comprende tutti i casi estremi della registrazione di un nuovo utente. Come per il `LoginEdgeCasesTest`, vengono controllati i messaggi d'errore dei campi non inseriti e non validi e inoltre è presente anche un test per la registrazione di un utente già registrato al sistema.

CreateSpaceTest Tale test simula la creazione di un nuovo spazio. Si parte dall'accesso di un utente registrato e successivamente, viene creato uno nuovo spazio inserendone il nome. Dopodiché si verifica che lo spazio sia stato creato effettivamente con il nome inserito e se l'utente creatore dello spazio, sia stato inserito come partecipante, con il ruolo di admin.

NotificationTest Tale test verifica la corretta visualizzazione della pagina delle notifiche. Inizialmente, viene eseguito il login dell'utente e successivamente viene cliccata l'icona notifiche, per testare il corretto cambio di pagina all'interfaccia corrispodente.

Test manuali Sono stati eseguiti vari test manuali sia su un device fisico che emulato.

In particolare, per l'emulatore, si è utilizzato l'emulatore Android presente in Android Studio. Esso consente di simulare i dispositivi Android in modo da poter testare l'applicazione con diverse configurazioni, senza dover disporre del dispositivo fisico.

Per quanto riguarda il test sul device fisico, l'applicazione è stato testata sia con un dispositivo con versione Android 7.0, sia con versione Android 10 per verificare il corretto funzionamento su dispositivi con API level Android differenti e per verificare la corretta visualizzazione su schermi con risoluzione diversi.

5.1.3 Testing del Client Web

Per quanto concerne il testing del Client Web sono stati compiuti test manuali dell'applicativo, durante la sua implementazione.

Test manuali Le diverse pagine sono state testate una ad una, definendone prima di tutto il codice HTML e verificandone la loro corretta visualizzazione sul browser. Successivamente sono stati definiti gli elementi grafici, la cui correttezza è stata verificata sempre procedendo all'esecuzione della pagina sul browser e verificando che essa fosse corretta. Infine, si è proceduto a testare la reattività dell'interfaccia grafica e il corretto invio delle richieste al Server nonché la corretta visualizzazione dei dati ricevuti in risposta, tramite la simulazione delle operazioni dell'utente.

In particolare, per il debugging degli elementi grafici e della reattività dell'interfaccia, nonché del corretto invio delle richieste al Server, si è fatto utilizzo degli strumenti di ispezione messi a disposizione dal browser stesso, che non solo offrono la possibilità di stoppare l'esecuzione del codice Javascript mediante l'utilizzo di appositi breakpoint, ma mettono anche a disposizione uno strumento per analizzare i pacchetti inviati e ricevuti dalla rete.

5.2 Modalità di lavoro

Il progetto è stato gestito mediante un apposito repository GitLab, nel quale si è provveduto a creare un branch `master` su cui caricare il lavoro finale e un branch `develop` di sviluppo, sul quale sono stati aperti diversi branch rappresentanti le diverse feature su cui si è lavorato.

Il lavoro è stato suddiviso in tre parti, come il numero di componenti del gruppo, in base alle capacità e competenze di ognuno, in particolare: Aurora Laghi si è occupata dello sviluppo del Client Web, Anna Vitali si è occupata dell'implementazione del database, dello sviluppo del Server, nonché della sua conteinerizzazione e infine Elena Yan si è occupata della realizzazione del Client Mobile.

Per quanto riguarda l'attività svolta, inizialmente sono stati effettuati degli incontri per stabilire l'interfaccia grafica dell'applicazione mediante la realizzazione di opportuni Mock-up, poi sono stati svolti degli incontri per stabilire lo schema del database, che è stato successivamente raffinato in fase di implementazione da Anna Vitali, dopodiché i componenti del team hanno proceduto in parallelo nell'esecuzione dei loro compiti,

confrontandosi in caso di problemi; inizialmente Elena Yan e Aurora Laghi hanno realizzato l'interfaccia dei clients, per poi passare all'implementazione della parte relativa alle richieste una volta che il Server è stato ultimato.

6 Istruzioni per il deployment

In questa sezione verranno indicate le istruzioni per il deployment, che è necessario seguire, per la corretta messa in funzione ed esecuzione del sistema.

6.1 Setup e installazioni di base

Per prima cosa è necessario scaricare il repository da gitLab e aprire il progetto spostandosi nella cartella `SettingUpHome`. Per poter eseguire la build del sistema è necessario aver installato:

- Una versione di Java 11 o superiore
- Un Android SDK
- Gradle
- Docker

Per comodità si può decidere di scaricare direttamente Android Studio che contiene già i diversi elementi specificati precedentemente, a parte Docker e aprire il progetto all'interno dell'IDE.

Una volta verificato di avere tutte le impostazioni necessarie si può effettuare la build del sistema dalla directory `SettingUpHome` lanciando il comando `gradle build` da un qualsiasi terminale.

6.2 Setup della connessione a MySQL Server

Per impostare il database è necessario installare MySQL Server, per farlo basta collegarsi al seguente link `https://dev.mysql.com/downloads/windows/installer/8.0.html` e scaricare **MySQL Community Server**. Una volta avviato l'installer, nella scelta del setup, scegliere l'opzione **Developer Default**.

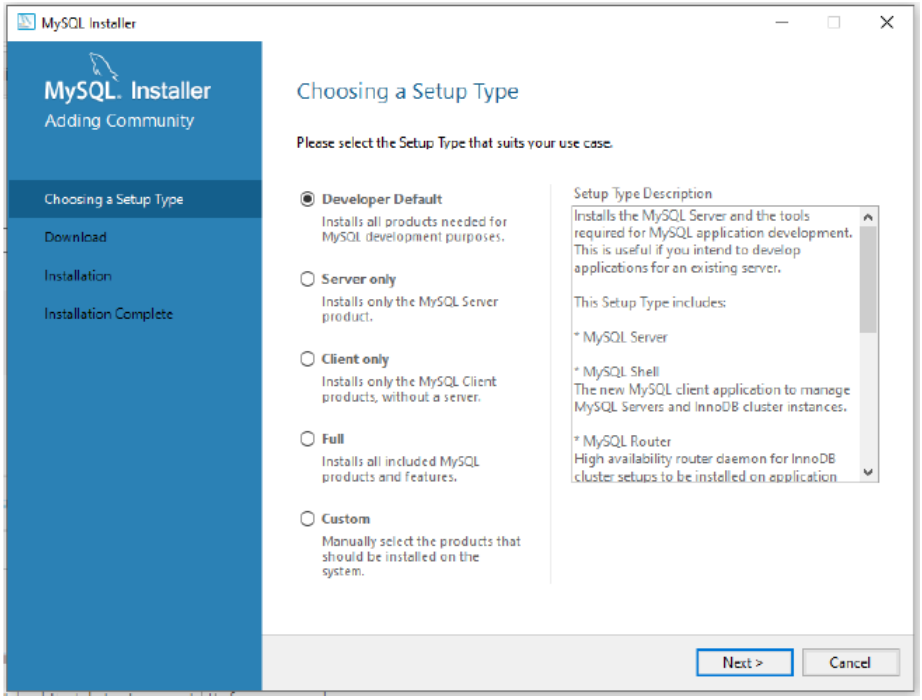


Figura 22: Setup Type Installer MySQL

Procedere nelle diverse schermate dell'installer e avviare l'installazione dei diversi prodotti MySQL. Impostare la connettività secondo le impostazioni di default che vengono presentate (Port: 3306 e X Protocol Port: 33060) e cliccare su next.

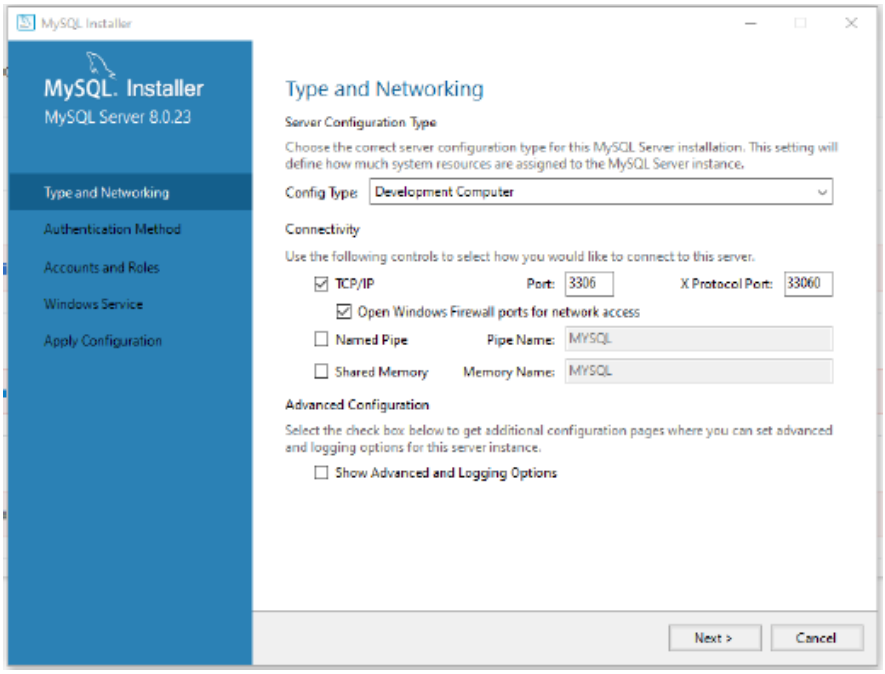


Figura 23: Type and Networking Installer MySQL

Impostare la password dell'utente root e concludere l'installazione, mantenendo le impostazioni di default presentate dall'installer.

Una volta terminata la configurazione, aprite MySQLWorkbench, la quale dovrebbe già presentarvisi con una connessione a MySQL Server, come mostrato nella figura 24.

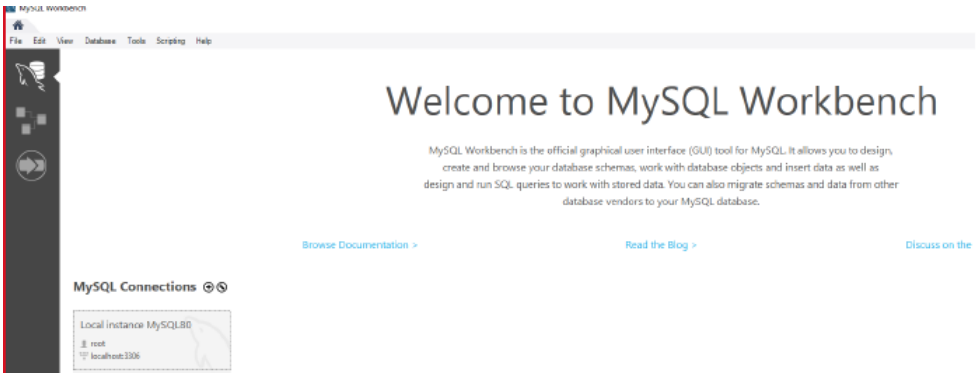


Figura 24: Schermata iniziale MySQLWorkBench

A questo punto cliccare sul pulsante più (+), per aggiungere una nuova connessione. Nella schermata di aggiunta che appare, assegnare un nome arbitrario alla connessione, impostare come hostname **localhost** e come port la **3307**

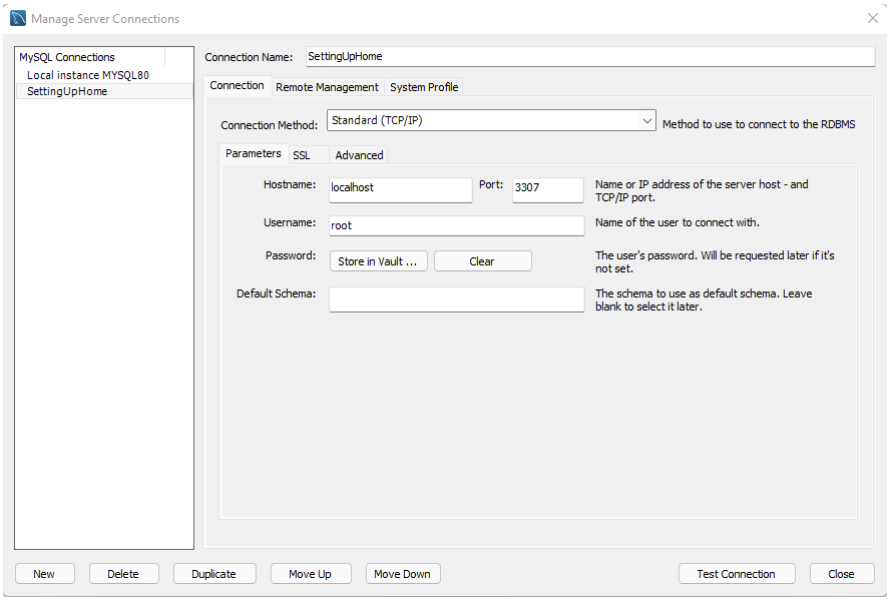


Figura 25: Impostazioni nuova connessione

6.3 Setup del Server e del database

Per poter avviare il Server è necessario prima aver installato Docker, per fare questo seguire le istruzioni al seguente link: `https://docs.docker.com/get-docker/`.

Una volta che si è provveduto all'installazione di Docker, aprire la directory `SettingUpHome` all'interno di un qualsiasi IDE (se non lo si è già fatto in precedenza) e spostarsi all'interno del modulo `Db`, aprire il file `docker-compose.yaml` e impostare la password della nuova connessione a MySQL Server creata in precedenza, nella variabile d'ambiente `MYSQL_ROOT_PASSWORD`, salvare le modifiche.

Sempre all'interno del modulo `Db` aprire la classe `DbConnection`, all'interno del package `Db` e impostate la password della connessione, settando l'apposita variabile `password` dichiarata all'interno del metodo `getMySQLConnection()`, salvare le modifiche.

A questo punto, aprire un prompt dei comandi nella directory `SettingUpHome` e lanciare lo script `startService.sh`, che si occupa di istanziare i diversi container, quest'operazione potrebbe richiedere qualche momento, quindi attendere un attimo prima di procedere con le successive operazioni.

A questo punto, una volta istanziati i container aprire MySQLWorkbench, cliccare sulla nuova connessione che è stata creata in precedenza, inserendo alla richiesta della password quella impostata per l'utente root di tale connessione, se non viene richiesta la password attendere qualche istante che il container del database venga istanziato e ripetere l'operazione.

Una volta effettuato l'accesso alla connessione correttamente, cliccare su File, Open SQL Script e selezionare il file `Setting_up_home_Db.sql`, che si trova all'interno della cartella `Database`, eseguire tale script cliccando sull'apposita icona dell'interfaccia, al

termine dell'esecuzione, effettuare il refresh degli `SCHEMAS`, dovrebbe ora apparire lo schema `setting_up_home`.

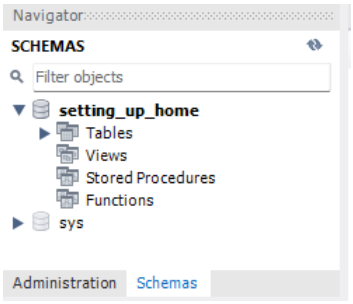


Figura 26: Visualizzazione del database Setting Up Home

Una volta giunti fino a qui, il database è stato settato e il Server è stato fatto partire, per poter iniziare ad utilizzare l'applicazione occorre aspettare che Gradle finisca la build del progetto Server, che potrebbe richiedere un po' di tempo, per controllare l'avanzamento si può utilizzare il comando `docker logs container_id`, trovando il container id di interesse eseguendo il comando `docker ps`.

Se si vuole spegnere il Server e rimuovere i container basta aprire un prompt dei comandi nella cartella `SettingUpHome` e lanciare lo script `stopService.sh`.

6.4 Setup del Client Web

Per riuscire ad avviare il Client Web, è necessario disporre di un browser sul proprio dispositivo, attraverso cui poter visualizzare l'applicazione.

Per prima cosa spostarsi nel modulo `WebApp`; successivamente, è opportuno configurare all'interno del file `config.js`, nella rispettiva cartella `\js` del modulo, il parametro `host` per connettersi correttamente al Server, con l'indirizzo IP della macchina dove è presente il cluster docker del Server.

Per avviare l'applicazione è necessario aprire il file principale `index.html`, contenuto nella cartella `\html` del modulo. Una volta aperto questo file all'interno del browser, l'applicazione è in esecuzione e all'utente viene mostrata la pagina di Login del sistema, in cui se non è già registrato può decidere di effettuare la registrazione e accedere all'applicazione.

6.5 Setup del Client Mobile

Il Client Mobile ha bisogno di un file di configurazione JSON chiamato `config.json` situato nella directory `SettingUpHome\MobileApp\src\main\res\raw`. Un esempio di configurazione è raffigurato nel listato 10, 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 su cui il server è in ascolto delle richieste HTTP.

```
{
  "host": "EXTERNAL_DOCKER_MACHINE_IP",
  "port": 10000
}
```

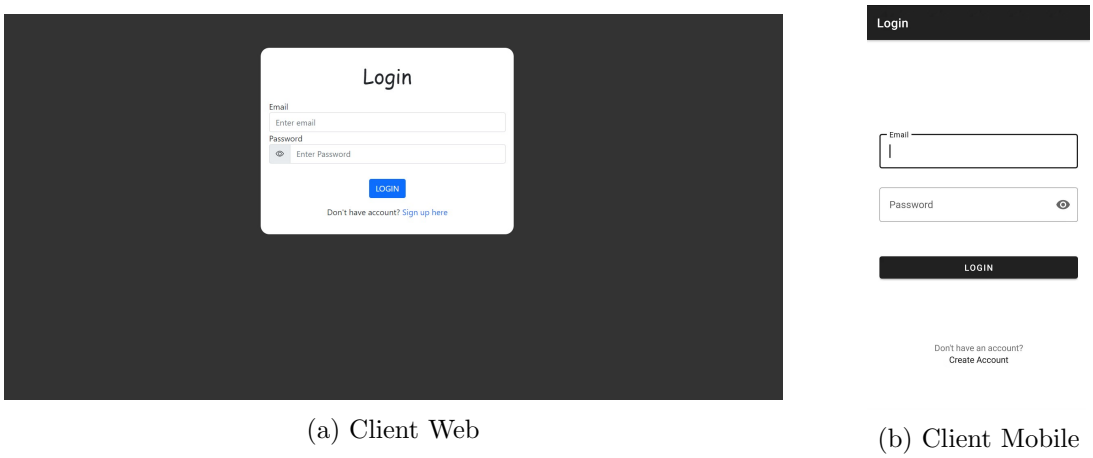
Listato 10: File di configurazione del client Mobile

Una volta caricato il file, è possibile installare l'applicazione con Android Studio collegando il dispositivo fisico oppure un dispositivo virtuale Android emulato.

7 Esempi di utilizzo

Supponiamo di avere un gruppo di coinquilini che vuole utilizzare l'applicazione, per prima cosa ognuno di loro dovrà registrarsi all'applicativo, usando l'applicazione Web oppure l'applicazione Android fornita.

Nel caso in cui un utente sia già registrato, invece, può accedere all'applicazione tramite l'effettuazione del login raffigurato nella figura 27.

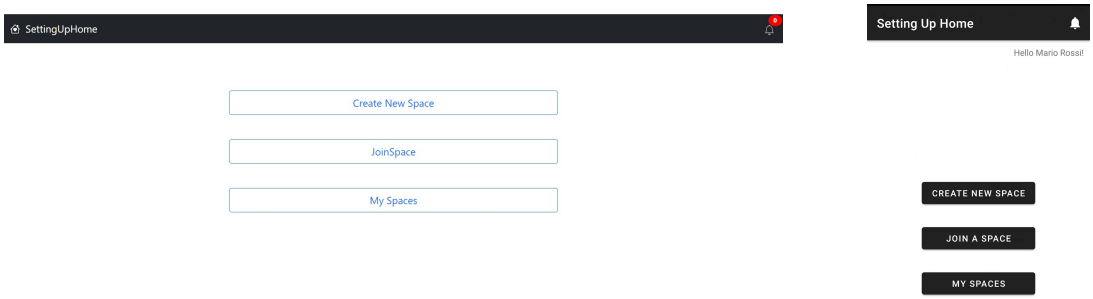


(a) Client Web

(b) Client Mobile

Figura 27: Schermata di login dell'utente

Una volta effettuato l'accesso, l'utente si trova nella pagina principale dell'applicazione in cui può decidere se: creare un nuovo spazio, partecipare ad uno nuovo spazio creato, oppure entrare in uno spazio in cui è già partecipante (figura 28).

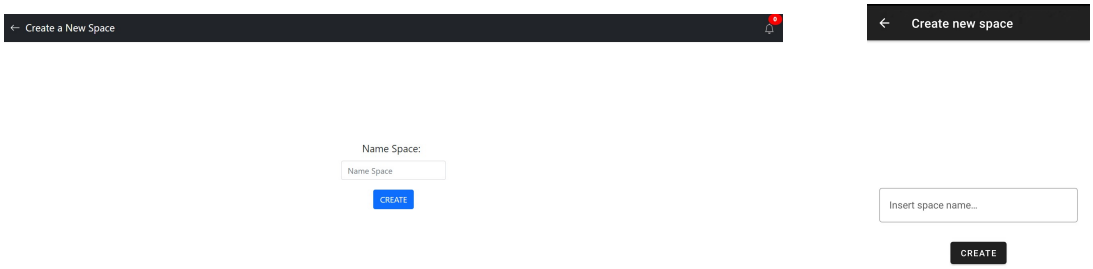


(a) Client Web

(b) Client Mobile

Figura 28: Schermata con il menu principale dell'applicazione

Tornando al nostro gruppo di utenti quindi, vi sarà un utente leader, che effettuerà la creazione di un nuovo spazio, inserendo i dati necessari (figura 29).



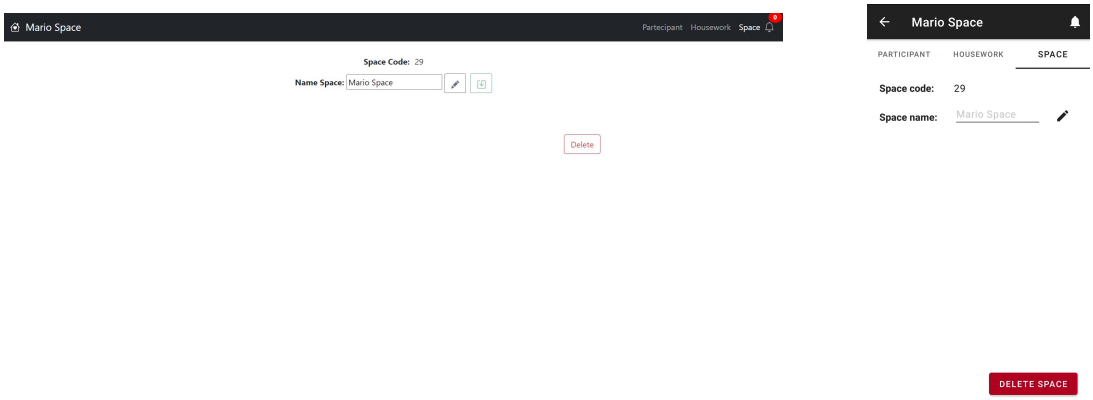
(a) Client Web

(b) Client Mobile

Figura 29: Schermata di creazione di uno nuovo spazio

Una volta che l'utente leader ha creato lo spazio, esso vi afferirà come utente amministratore e in quanto tale, potrà consentire l'accesso agli altri utenti allo spazio creato, tramite la condivisione del codice identificativo, che si trova nella sezione space dell'applicazione ed è visibile solamente agli amministratori.

Nella sezione space dell'applicazione un amministratore può anche decidere di eliminare lo spazio, come possibile vedere dalla figura 30.

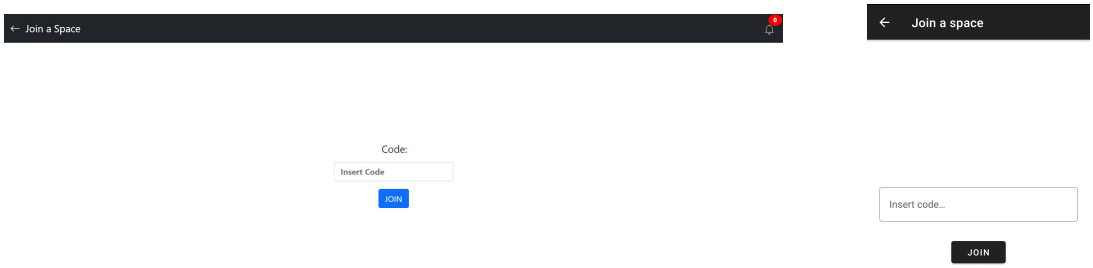


(a) Client Web

(b) Client Mobile

Figura 30: Schermata con la sezione spazio

Gli altri coinquilini, invece, per poter accedere allo spazio creato, dovranno cliccare sul pulsante "Join Space" e inserire il codice fornitoli (figura 31). Si ricorda che, inizialmente, quando un utente entra a far parte di uno spazio, non da lui creato, li verrà assegnato di default il ruolo di member.

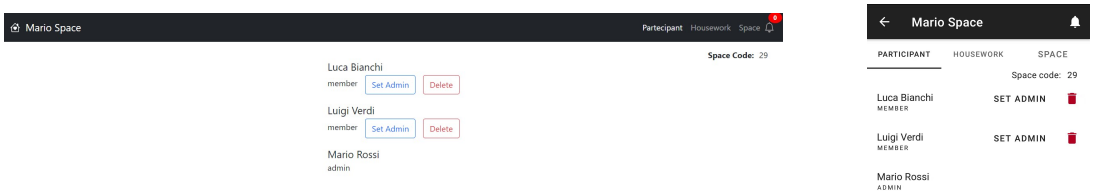


(a) Client Web

(b) Client Mobile

Figura 31: Schermata "Join spacÉ'

A questo punto l'amministratore del gruppo, potrà vedere i diversi utenti che si sono uniti allo spazio, andando nella sezione "Participant", in questa sezione, un amministratore può anche compiere diverse operazioni sui partecipanti come si può vedere dalla figura 32, infatti, è possibile: eliminare un partecipante o cambiare il ruolo di un utente da membro ad amministratore.



(a) Client Web

(b) Client Mobile

Figura 32: Schermata “Participant” di un utente amministratore

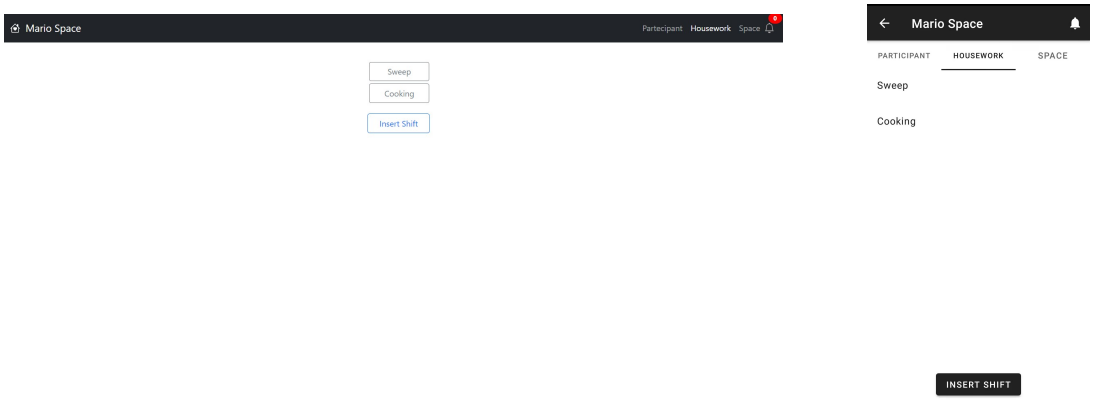
Anche un utente membro che partecipa ad uno spazio, può visualizzare le informazioni relative agli altri partecipanti, accedendo sempre alla sezione “Participant”; egli però non potrà effettuare le operazioni previste per gli amministratori, come è possibile vedere dalla figura 33, la schermata mostrata differisce da quella degli amministratori.

(a) Client Web

(b) Client Mobile

Figura 33: Schermata “Participant” di un utente membro

Supponiamo adesso di voler iniziare a definire i turni, quindi, ci spostiamo nella sezione “Housework” dell’applicazione (figura 34), in questa sezione è possibile visualizzare i turni già impostati per le diverse faccende, oppure se siamo un amministratore possiamo aggiungere dei nuovi turni per una faccenda, cliccando sul pulsante “Insert Shift”.

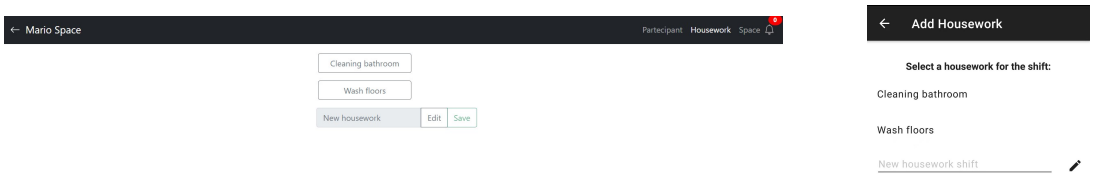


(a) Client Web

(b) Client Mobile

Figura 34: Schermata per l'inserimento di un nuovo turno

Alla pressione sul pulsante "Insert Shift", appare un elenco delle faccende già esistenti, per lo spazio corrente, che non hanno turni assegnati, è anche possibile decidere di aggiungere una faccenda *custom*, oltre a quelle *standard* già presenti, per l'inserimento di nuovi turni (figura 35).

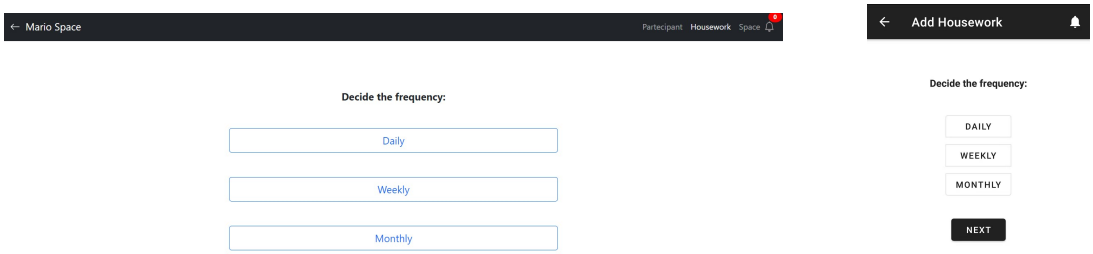


(a) Client Web

(b) Client Mobile

Figura 35: Schermata di selezione della faccenda per il turno

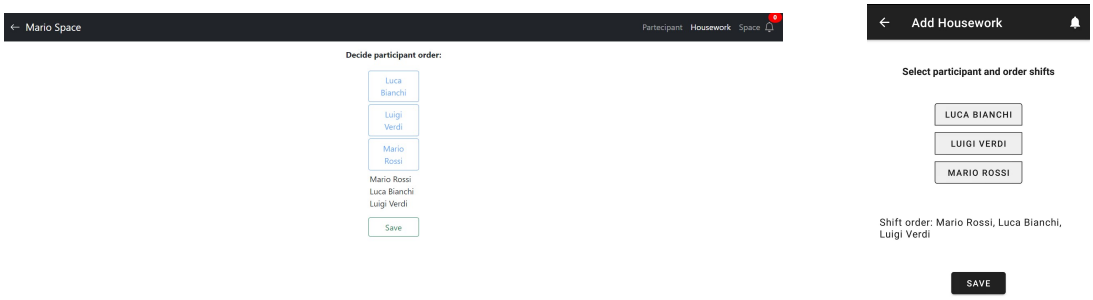
Supponiamo di voler assegnare i turni per la faccenda Wash Floors, per fare questo dobbiamo: selezionare la frequenza dei turni fra i diversi partecipanti (figura 36) e dopodiché selezionare l'ordine con cui essi dovranno alternarsi (figura 37).



(a) Client Web

(b) Client Mobile

Figura 36: Schermata di selezione della frequenza



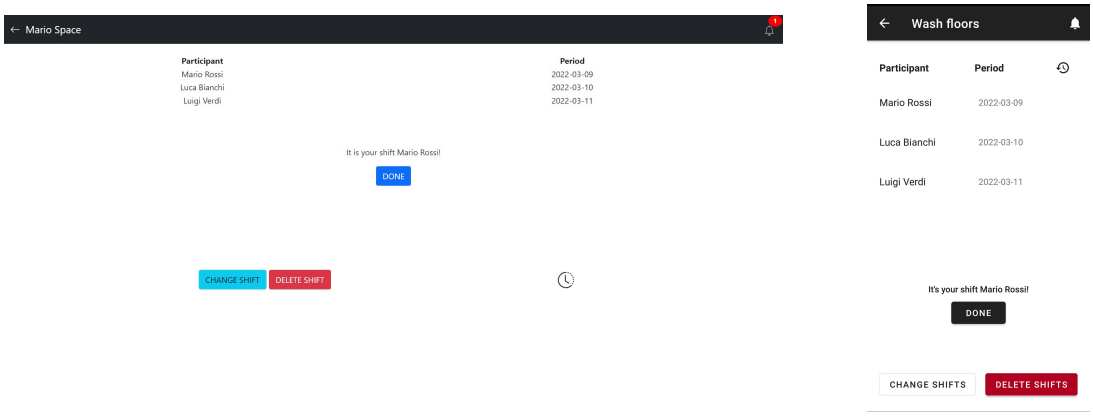
(a) Client Web

(b) Client Mobile

Figura 37: Schermata di selezione dell'ordine di svolgimento del turno

Una volta fatto questo, tornando alla schermata "Housework" apparirà la faccenda Wash Floors con i turni assegnati, se ora clicchiamo su questa faccenda possiamo infatti vedere tutte le informazioni relative ai compiti.

Se siamo un'utente amministratore, cliccando sulla faccenda, potremmo non solo visualizzare i turni, ma anche modificare quelli già esistenti, cambiando la frequenza e l'ordine dei partecipanti oppure possiamo anche decidere di cancellarli, cliccando sull'apposito pulsante, come rappresentato nella figura 38.



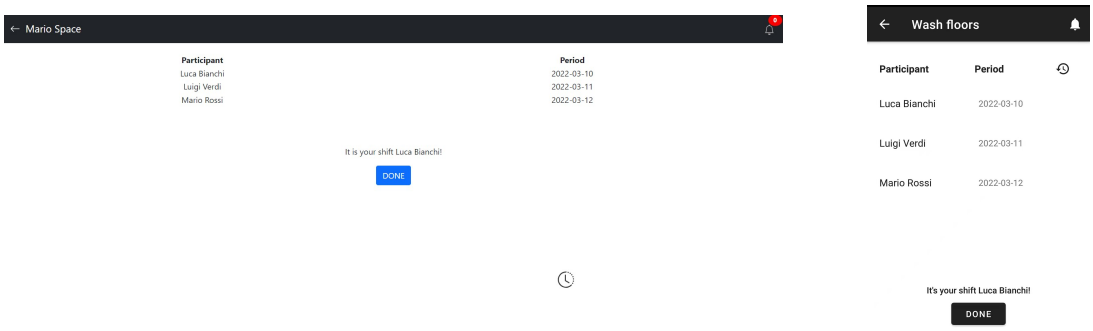
(a) Client Web

(b) Client Mobile

Figura 38: Schermata turni, visualizzazione di un utente amministratore

Se invece siamo un utente membro allora possiamo solo visualizzare tutti i turni assegnati.

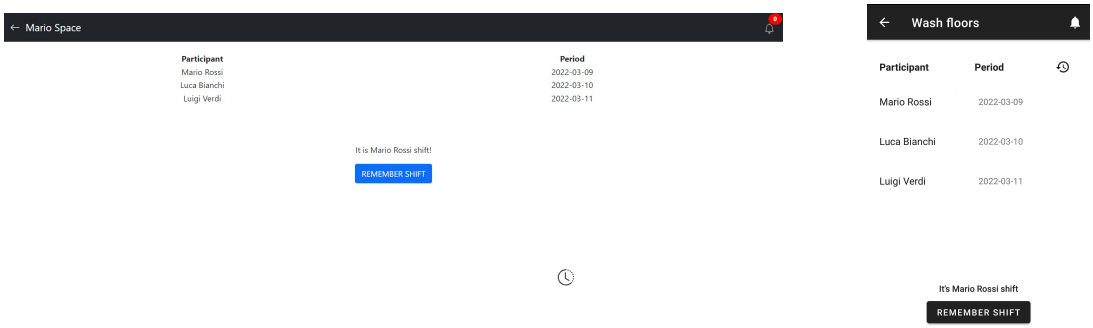
Nel caso in cui siamo l'utente che deve svolgere il turno corrente in questo momento, per la faccenda Wash Floors, allora avremmo la possibilità di segnare il turno eseguito, come rappresentato nella figura 39, se invece non siamo il partecipante di turno in questo momento, posso ricordare di effettuare il turno all'utente designato, come nella figura 40.



(a) Client Web

(b) Client Mobile

Figura 39: Schermata dell'utente del turno corrente

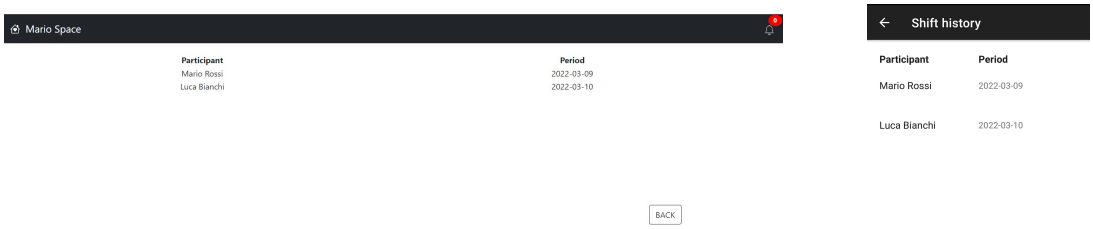


(a) Client Web

(b) Client Mobile

Figura 40: Schermata dell'utente non del turno corrente

Dopo che un turno è stato evidenziato come svolto, sarà possibile visualizzare tale turno nello storico, cliccando sull'apposito pulsante, che conduce alla schermata riportata nella figura 41.

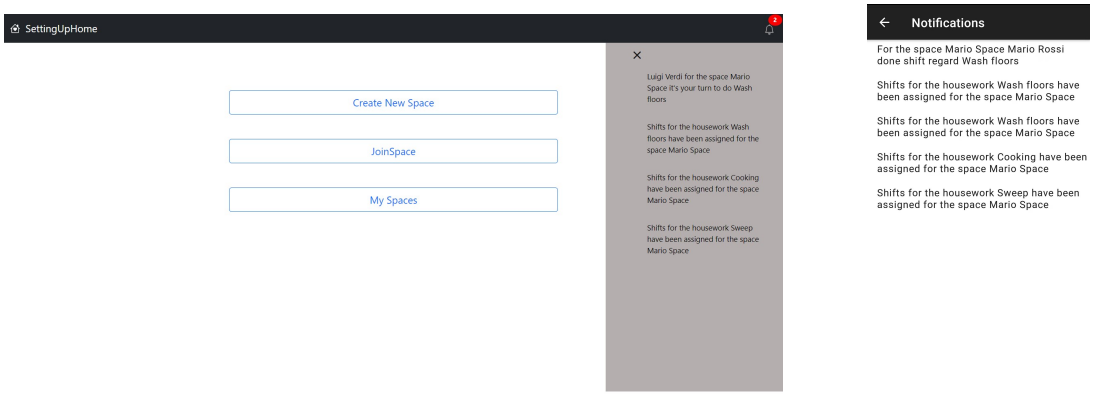


(a) Client Web

(b) Client Mobile

Figura 41: Schermata dello storico dei turni

È possibile inoltre visualizzare le notifiche relative ai turni come si vede nella figura 42, per le diverse faccende dei diversi spazi a cui partecipa l'utente, accedendo all'apposita sezione messa a disposizione.



(a) Client Web

(b) Client Mobile

Figura 42: Schermata notifiche

8 Conclusioni

Si voleva realizzare un'applicazione che consentisse la gestione di diverse attività domestiche fra diversi utenti che convivono assieme, rendendo agevole e semplice l'organizzazione di queste attività. Si è stabilito inoltre, che l'accesso all'applicazione da parte degli utenti, doveva poter avvenire sia mediante una Web App, che mediante un'applicazione Android.

Per realizzare questo obiettivo, per prima cosa sono stati stabiliti i requisiti funzionali e non funzionali che il sistema doveva avere, dopodiché sono state individuate le diverse entità che costituiscono il modello del dominio e poi si è passati alla loro implementazione, definendone struttura, comportamento e interazione.

In particolare, il sistema si è rilevato essere costituito da quattro entità fondamentali: un database relazionale, un Server Web, un Client Mobile e un Client Web.

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

Il funzionamento dell'applicazione si basa sul meccanismo di *Remote Procedure Call*, il quale prevede che il Client effettui una chiamata a procedura remota al Server, attendendo il suo completamento e la ricezione della risposta; di conseguenza, per quanto riguarda l'interazione, si è visto che essa parte sempre dal Client il quale invia una richiesta al Server, che può essere soddisfatta anche mediante diverse interazioni di questi con il database e infine l'interazione si conclude con l'invio della risposta da parte del Server e la ricezione di questa dal Client.

Tutte le diverse funzionalità realizzate sono state testate sia mediante l'utilizzo di test automatici che manuali e al termine dell'attività e analizzando quello che è stato fatto,

possiamo affermare che il sistema realizzato soddisfa gli obiettivi e i requisiti individuati, e che si è complessivamente soddisfatti del risultato raggiunto.

8.1 Lavori futuri

Possibili funzionalità aggiuntive dell'applicazione che possono essere realizzate in futuro, in aggiunta a quelle già presenti, possono essere le seguenti:

- Consentire agli utenti di avere calendario condiviso, dove poter inserire eventi che possono essere visualizzati da tutti i partecipanti alla casa, con la possibilità agli interessati di ricevere una notifica come monito,
- Consentire la gestione delle spese della casa, permettendo agli utenti di inserire nuove voci riguardanti diverse categorie (bollette, cibo, servizi, casa, animali...) e un importo che verrà poi suddiviso con i partecipanti del gruppo coinvolti,
- Mettere a disposizione una chat per la comunicazione fra i diversi componenti del gruppo.

8.2 Cosa è stato appreso

In questa parte della relazione, ogni componente del gruppo, ha provveduto a indicare quali sono stati i concetti appresi grazie alla realizzazione di questo progetto.

Aurora Laghi Attraverso questo progetto, ho potuto approfondire lo studio di alcuni argomenti illustrati durante il corso. In particolare, ho apprezzato la tecnologia Swagger per la gestione delle API REST. Nonostante sia stata implementata dalla collega Anna Vitali, si è rilevato un tool molto intuitivo e utile al fine di testare le risposte del Server, nel momento in cui si sono presentati problemi a livello di richieste lato Client Web.

Inoltre, ho potuto migliorare le nozioni riguardanti i metodi jQuery Ajax, i quali sono risultati l'elemento chiave per lo scambio dei dati col Server. Ancora una volta, ho potuto confermare le potenzialità del tool Gradle, il quale ha garantito lo sviluppo dell'intero progetto in maniera fluida.

In aggiunta, è stato motivante e interessante riuscire a gestire, insieme al team, i problemi di CORS che si sono manifestati durante l'implementazione dell'applicazione lato Client Web.

Infine, è risultato molto stimolante lavorare con le colleghe Anna Vitali ed Elena Yan, poiché si sono mostrate eccellenti nel loro lavoro e dalle quali ho potuto apprendere molti aspetti da loro implementati in maniera veloce attraverso una semplice spiegazione.

Anna Vitali Grazie a questo progetto, ho avuto la possibilità di vedere in pratica alcuni aspetti studiati fin ora solo nella teoria. Ho avuto modo di usare Docker per la prima volta in un progetto, comprendendo meglio le potenzialità di questo strumento e il fatto di aver utilizzato anche Docker Swarm per poter creare delle repliche del servizio, è stato veramente molto utile, per capire come un'applicazione complessa possa essere

realizzata. Ho inoltre avuto modo di utilizzare nuovi strumenti come OpenAPI, per la specifica dell'API e Swagger che grazie a quest'esperienza ho scoperto essere uno strumento veramente molto utile e semplice da utilizzare. Infine, ho anche raffinato le mie conoscenze sul build-tool Gradle, realizzando per la prima volta un progetto costituito da più moduli e gestendo le dipendenze tramite Gradle e Kotlin.

Elena Yan Grazie a questa esperienza ho appreso molte cose, infatti mi ha permesso di rafforzare le conoscenze acquisite durante il corso e di comprendere meglio il funzionamento di un sistema distribuito. Ho imparato ad usare nuovi strumenti come Swagger, molto comodo per visualizzare le rotte definite e per testate le richieste al Server e di conoscere le potenzialità offerte da Docker. Inoltre mi ha permesso di migliorare le conoscenze di Gradle, utilizzato per gestire i moduli del progetto e la gestione delle dipendenze. Sono abbastanza soddisfatta del lavoro che è stato svolto e dei risultati finali ottenuti considerando anche le difficoltà e i problemi incontrati durante lo sviluppo.

Stylistic Notes

- 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] Android. Espresso — android developer. `https://developer.android.com/training/testing/espresso`. Accessed March 2022.

[2] Android. Guide to app architecture — android developer. `https://developer.android.com/jetpack/guide`. Accessed March 2022.

[3] Docker. Docker overview. `https://docs.docker.com/get-started/overview/`. Accessed March 2022.

[4] Docker. Mysql - official image. `https://hub.docker.com/_/mysql/`. Accessed March 2022.

[5] Google. Gson user guide. `https://github.com/google/gson/blob/master/UserGuide.md#TOC-Overview`. Accessed March 2022.

[6] Google. Volley. `https://google.github.io/volley/`. Accessed March 2022.

[7] jQuery. jquery. `https://jquery.com/`. Accessed March 2022.