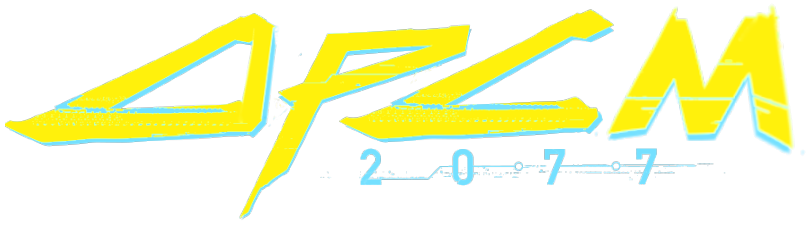

MANUALE MANUTENTORE

BLOCKCOVID

dpcm2077@gmail.com

INFORMAZIONI SUL DOCUMENTO

Versione	1.0.0
Uso	esterno
Stato	approvato
Destinatari	Prof. Tullio Vardanega
	Prof. Riccardo Cardin
	Imola Informatica
	DPCM 2077
Redazione	Matteo Budai
	Ivan Piacere
	Damiano Bertoldo
Verifica	Daniele Spigolon
	Antonio Badan
Approvazione	Sara Privitera

Registro delle modifiche

Versione	Descrizione	Data	Autore	Ruolo
1.0.0	Approvazione del documento	2021-06-04	Sara Privitera	Responsabile
0.1.11	Aggiornamento e verifica §3.5, §3.6, §3.7	2021-06-04	Ivan Piacere Antonio Badan	Amministratore Verificatore
0.1.10	Aggiornamento e verifica §6.2 - §7.3	2021-06-04	Damiano Bertoldo Antonio Badan	Amministratore Verificatore
0.1.9	Aggiornamento e verifica §2.5 - §2.6 - §2.7	2021-06-04	Matteo Budai Daniele Spigolon	Amministratore Verificatore
0.1.8	Aggiornamento e verifica §2.1 e §2.3	2021-06-04	Matteo Budai Daniele Spigolon	Amministratore Verificatore
0.1.7	Stesura e verifica §5.3	2021-06-03	Damiano Bertoldo Antonio Badan	Amministratore Verificatore
0.1.6	Approvazione del documento	2021-05-09	Damiano Bertoldo	Responsabile
0.0.6	Stesura e verifica §7 e §8	2021-05-09	Samuele De Grandi Daniele Spigolon	Amministratore Verificatore
0.0.5	Stesura e verifica §1, §3 e §6	2021-05-08	Samuele De Grandi Daniele Spigolon	Amministratore Verificatore
0.0.4	Stesura e verifica §5	2021-05-07	Samuele De Grandi Daniele Spigolon	Amministratore Verificatore
0.0.3	Stesura e verifica §2.5 - §2.6 - §2.7	2021-05-06	Matteo Budai Daniele Spigolon	Amministratore Verificatore
0.0.2	Stesura e verifica §2.1 - §2.2 - §2.3 - §2.4	2021-05-04	Matteo Budai Daniele Spigolon	Amministratore Verificatore
0.0.1	Stesura e verifica §4	2021-05-04	Matteo Budai Daniele Spigolon	Amministratore Verificatore

Indice

1 Introduzione

1.1 Scopo documento

Lo scopo di questo documento consiste nel fornire una guida all'installazione e alla manutenzione del sistema BlockCovid. Il contenuto è suddiviso in sezioni dedicate ai singoli componenti del prodotto:

- App utenti: l'APPLICAZIONE$_G$ mobile dedicata ai dipendenti e agli addetti alle pulizie;
- Web app: l'INTERFACCIA$_G$ web per gli amministratori;
- BACKEND$_G$: il SERVER$_G$ che espone la API$_G$;
- Database;
- BLOCKCHAIN$_G$;

Per ogni sezione sono indicati i requisiti e i passi per l'installazione. Per le componenti più complesse viene fornita una descrizione dell'ARCHITETTURA$_G$ e una guida all'estensione.

1.2 Glossario

All'interno del documento sono presenti termini che assumono significati diversi a seconda del contesto. Per evitare ambiguità, è stata posta alla fine del documento una sezione di nome Glossario che conterrà tali termini con il loro significato specifico. Per segnalare che un termine del testo è presente all'interno del Glossario, verrà aggiunta una G pedice posta a fianco del termine ambiguo.

2 App utenti

2.1 Introduzione

Questa parte del documento è orientata agli utenti che utilizzano l'applicazione ANDROID$_G$.

2.1.1 Scopo del prodotto

L'applicazione Android è sviluppata per due diverse tipologie di utente ovvero il dipendente e l'addetto alle pulizie.

In generale offre all'utente le seguenti funzionalità:

- **Login:** L'utente ha la possibilità di autenticarsi inserendo il proprio username e password;
- **Logout:** L'utente ha la possibilità di deautenticarsi premendo sull'elemento della lista "Logout" del menù principale in alto a destra.

Per quanto riguarda il dipendente, l'applicazione offre i seguenti servizi:

- **Scansione:** Il dipendente ha la possibilità di scansionare una POSTAZIONE$_G$ per poter visualizzare lo stato di essa e altre informazioni come le prenotazioni associate ad essa;
- **Occupazione:** Il dipendente, dopo aver scansionato una postazione, può occuparla se questa è prenotata da lui o è libera e igienizzata;
- **Igienizzare:** Il dipendente, dopo aver scansionato una postazione, la può igienizzare se questa risulta non igienizzata;
- **Lista prenotazioni:** Il dipendente può visualizzare le prenotazioni effettuate premendo sull'elemento della lista "Visualizza prenotazioni" del menù principale in alto a destra;
- **Disdire prenotazione:** Il dipendente può disdire una prenotazione dopo che è entrato nella pagina in cui visualizza tutte le prenotazioni effettuate;
- **Guida:** Il dipendente può visualizzare la guida premendo sull'elemento della lista "Guida" del menù principale in alto a destra;
- **Prenota postazione:** Il dipendente può prenotare una postazione premendo sull'elemento della lista "Prenota postazione" del menù principale in alto a destra. Dopo aver premuto dovrà inserire

la data, l'ora di inizio, l'ora di fine e la stanza obbligatoriamente. Una volta premuto sul bottone "Cerca" visualizzerà tutte le postazioni di quella stanza che sono disponibili e potrà decidere quale prenotare.

Per quanto riguarda l'addetto alle pulizie, l'applicazione offre i seguenti servizi:

- **Lista postazioni e stanze:** Il dipendente, dopo aver premuto sull'apposito bottone può ottenere la lista delle stanze o delle postazioni da igienizzare;

- **Igienizzare:** Il dipendente, dopo aver ottenuto l'elenco delle stanze o delle postazioni può igienizzarle premendo sull'apposito bottone.

2.2 Requisiti e installazione

2.2.1 Requisiti

Per poter sviluppare sul proprio PC l'applicazione sono necessari i software e gli strumenti indicati in questa sezione. I software da installare saranno divisi in base al loro scopo.

Per scaricare il codice sorgente dell'applicazione bisogna andare nella pagina di GitHub$_G$ che lo ospita, che si trova qui, cliccare su Clone or download e successivamente premere su Download ZIP.

Un'alternativa più efficace a questo procedimento è scaricare il progetto tramite Git. Se non si dispone di Git è possibile scaricarlo seguendo quanto indicato nella sezione Source Code Management. Per scaricare il progetto in questo modo, digitare il seguente comando tramite un terminale o prompt dei comandi nel sistema in uso:

git clone https://github.com/DPCMGroup/bc19-mobile.git

2.2.2 Prerequisiti hardware e software

Le tecnologie utilizzate per sviluppare l'applicazione Android richiedono parecchie risorse nel loro uso contemporaneo. Si consiglia quindi di avere un computer con processore almeno quad-core e una memoria RAM di almeno 8 GB.

2.2.3 Ambiente di sviluppo

2.2.3.1 Android Studio

L'applicazione è stata sviluppata utilizzando l'ambiente di sviluppo Android Studio, attualmente alla versione 4.1.3.

Installazione di Android Studio su Windows
È possibile scaricare Android Studio su WINDOWS$_G$ visitando il sito ufficiale riportato qui, andando alla sezione "Android Studio downloads". Per eseguire l'installazione, bisognerà seguire la guida riportata nella sezione Windows cliccando nel seguente link qui.

Installazione di Android Studio su MacOS
La guida per scaricare Android Studio per MACOS$_G$ è identica a quella per Windows. Per eseguire l'installazione, invece, bisognerà seguire la guida riportata nella sezione Mac cliccando nel seguente link qui.

Installazione di Android Studio su Ubuntu (e derivate, e altri derivati di Debian)
La guida per scaricare Android Studio per LINUX$_G$ è identica a quella per Windows e MacOS. Per eseguire l'installazione, invece, bisognerà seguire la guida riportata nella sezione Linux cliccando nel seguente link qui.

2.2.4 Linguaggi utilizzati

2.2.4.1 Kotlin

Installazione di Kotlin su Windows
È possibile installare KOTLIN$_G$ su Windows visitando questa pagina. Il link porta alla pagina di jetBrains e ti permette di installare l'ultima versione di Kotlin disponibile anche con il plugin su Android Studio.

Installazione di Kotlin su MacOS
L'installazione per MacOS è identica a quella per Windows.

Installazione di Kotlin su Ubuntu (e derivate, e altri derivati di Debian)
È possibile installare Kotlin su sistemi Linux in modo identico a MacOs e Windows.

2.2.4.2 XML

La configurazione dell'applicazione e alcune sue dipendenze sono gestite tramite un file denominato AndroidManifest.xml. Una dipendenza importante da inserire nel file denominato AndroidManifest.xml è: android:networkSecurityConfig="@xml/network_security_config". Bisogna inserire anche la dipendenza *<uses-permission android:name="android.permission.NFC" />* per permettere l'attivazione della tecnologia NFC sullo smartphone. Inoltre, lo sviluppo di applicazioni Android richiede una cartella di progetto denominata "res" che contiene tutti i file XML per gestire risorse come layout, immagini, menu, stringhe e altro. Per permettere la comunicazione con il server bisogna creare il file denominato network_security_config.xml nella cartella res/xml con il seguente contenuto:
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
<base-config cleartextTrafficPermitted="true" />
</network-security-config>.
È richiesta una buona conoscenza del linguaggio XML.

2.2.5 Librerie utilizzate

2.2.5.1 Kotlin MVP auto

Viene utilizzato il plugin Kotlin MVP auto per generare in automatico una porzione di codice che rispetti l'architettura MVP. Il plugin è scaricabile a questa pagina. Per poterlo usare devono essere aggiunte le seguenti dipendenze al file build.gradle(:app):

- implementation 'com.github.cn-ljb:kotlin-mvp-lib:1.2.0';
- implementation 'com.github.cn-ljb:netlib:1.0.1';
- implementation 'com.github.cn-ljb:daolib:1.0.1'.

2.2.5.2 OkHttp

OkHttp è una libreria che permette di effettuare richieste al server. Per usarla è necessario aggiungere nel file build.gradle(:app) la dipendenza:
implementation 'com.squareup.okhttp3:okhttp:3.8.1'

2.2.6 Source code management

Per poter effettuare il VERSIONAMENTO$_G$ del codice sorgente è richiesto di utilizzare Git. Per poterlo installare è necessario recarsi a questa pagina. Non è strettamente necessario, ma è consigliato per integrare le proprie modifiche nel REPOSITORY$_G$.

2.2.7 Build automation

La build automation (ovvero la gestione del processo di build) è affidata a Gradle, integrato e utilizzato in Android Studio. I file di build sono due: uno per tutto il progetto ed uno per il solo modulo app. Tramite Gradle il progetto dell'applicazione viene compilato, TESTATO$_G$ ed eseguito attraverso l'IDE Android Studio.

2.3 Estendibilità

2.3.1 Creazione di un metodo

Tramite l'utilizzo dell'architettura Model View Presenter è facile implementare nuovi metodi nella business logic o nella vista e in seguito collegarli tra loro tramite il Presenter. Per creare una nuova Activity che rispetti l'architettura MVP bisognerà premere con il tasto destro su *com.example.bc19mobile*

e successivamente selezionare *New MVP Kotlin*. Assegnando un nome a questa Activity verrà creata una struttura che rispetta l'architettura.

2.4 Architettura

Il modello architetturale scelto è il Model View Presenter che è fortemente consigliato per chi sviluppa delle applicazioni per dispositivi Android. Il MVP fornisce un modo semplice per mostrare la struttura del prodotto garantendo modularità, testabilità e in generale una base di codice più pulita e gestibile. Ne deriva quindi l'applicazione del paradigma separation of concerns, che separa la responsabilità tra le differenti parti del pattern. Come detto in precedenza verrà usato il plugin Kotlin MVP auto per generare in automatico codice che rispetti l'architettura.

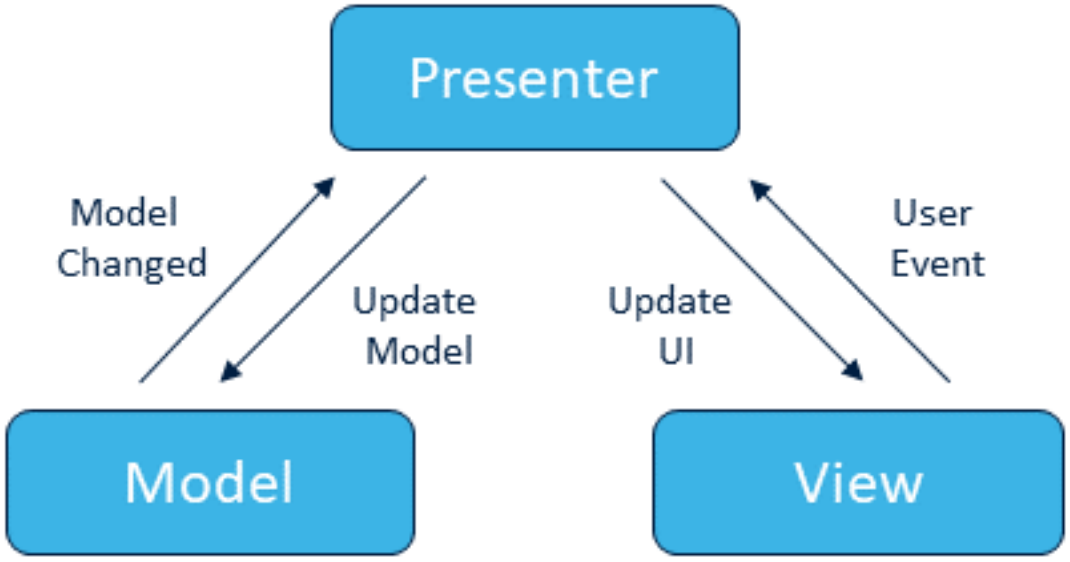

Figura 1: Model-View-Presenter

2.4.1 Model

Il Model è la parte dell'architettura che ha il compito di gestire i dati e rappresenta il layer di persistenza dell'applicazione. La maggior parte delle operazioni e dei controlli vengono svolti al suo interno. Contiene anche i metodi che avviano le connessioni alle API ed interagiscono con esse eseguendo numerose funzionalità.

2.4.2 View

La View ha la responsabilità di passare i dati al Presenter. Essa è implementata da:

- Attività (Activity);

- Qualsiasi forma grafica con cui l'utente finale dell'applicazione andrà ad interagire.

2.4.3 Presenter

Il Presenter funge da livello intermedio tra la View e il Model. Tutta la logica di presentazione appartiene ad esso ed è responsabile dell'interrogazione del modello e l'aggiornamento della vista, reagendo alle interazioni che compie l'utente nella UI. Un valore aggiunto è che il Presenter dipende dall'astrazione della View e non dalla sua concretizzazione, quindi non conosce la sua implementazione. Tutto ciò favorisce una più facile attività di test.

2.4.4 Contract

Il Contract può essere visto come un contratto nel quale vengono definiti tutti i metodi che verranno utilizzati dalla View, dal Presenter e dal Model. Quando si ha intenzione di scrivere una nuova funzionalità, è buona norma scrivere un Contract. Esso descrive la comunicazione tra View-Presenter e Model-Presenter, consentendo una progettazione più pulita e diminuendo le dipendenze tra le componenti. Il Contract è un'interfaccia e contiene le altre interfacce della View, Presenter e Model per garantire le varie comunicazioni.

2.5 Diagramma dei package

Vengono presentati di seguito i diagrammi UML dei package relativi all'applicazione.

2.5.1 Visione Generale

Il package principale dell'applicazione è nominato *com.example.bc19mobile*. Nel seguente diagramma UML vengono mostrate tutte le dipendenze che esistono tra i vari package.

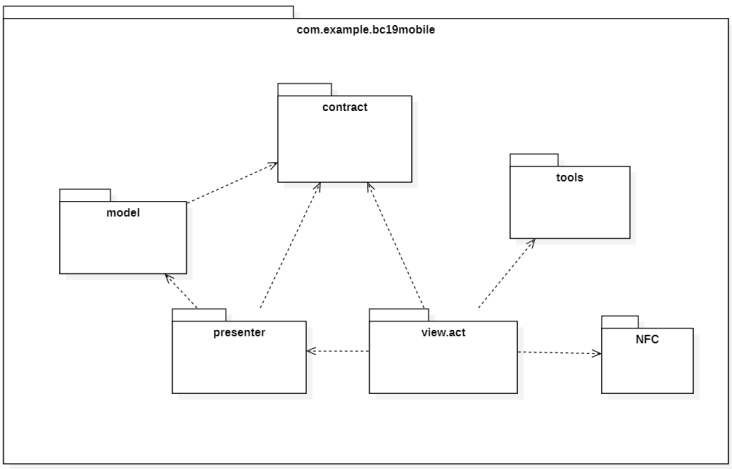

Figura 2: Visione generale dei package

2.5.2 Model

Nel seguente package vengono raggruppati altri package contenenti tutte le classi utilizzate per la business logic.

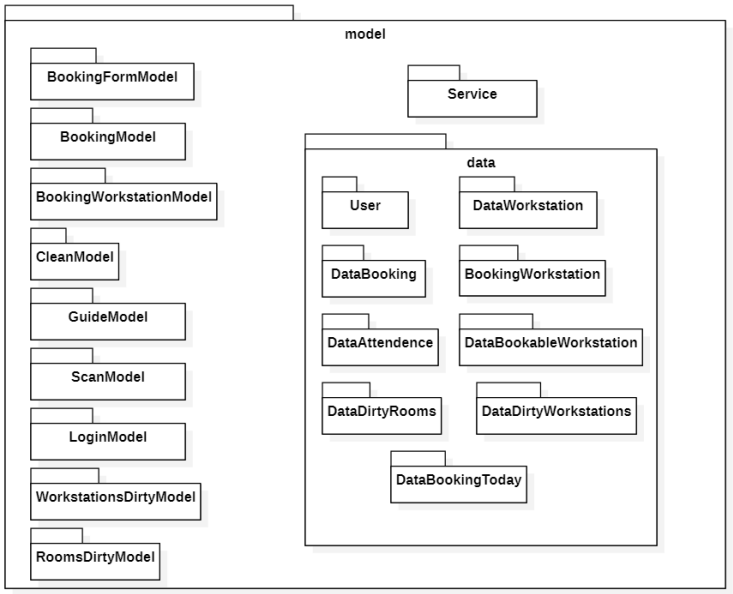

Figura 3: package-Model

2.5.3 View

Nel seguente package vengono raggruppate tutte le classi dedicate alle funzionalità di user interface collegate con i loro rispettivi file XML che modificano la loro interpretazione grafica.

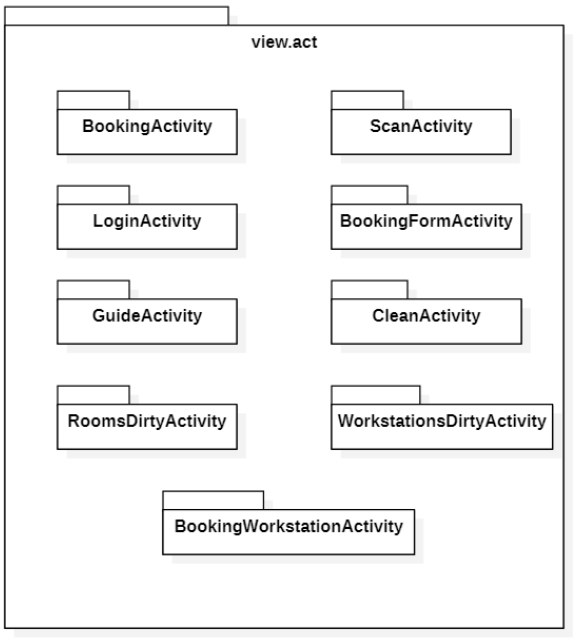

Figura 4: package-View

2.5.4 Presenter

Nel seguente package vengono raggruppate tutte le classi dedicate alla comunicazione tra le altre due componenti dell'architettura.

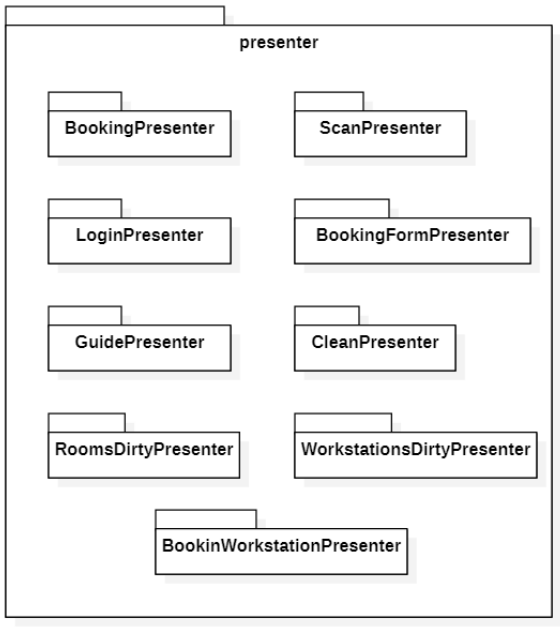

Figura 5: package-Presenter

2.5.5 Contract

Nel seguente package vengono raggruppate tutte le interfacce Contract che servono per facilitare la comunicazione tra le varie componenti di MVP. Visto che a loro volta contengono altre interfacce possono essere rappresentate come dei "package".

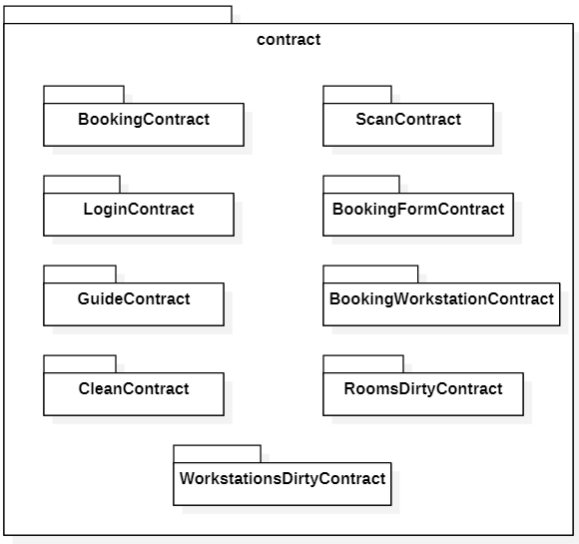

Figura 6: package-Contract

2.5.6 Tools

Nel seguente package vengono raggruppate tutte le classi ed interfacce dedicate alle funzionalità generiche che possono essere usate in base ai contesti per dare supporto alla vista.

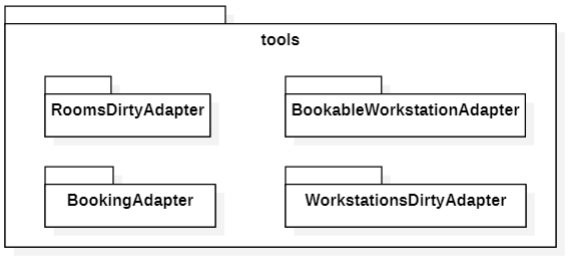

Figura 7: package-Tools

2.5.7 NFC

Nel seguente package vengono raggruppate tutte le classi, interfacce e oggetti dedicate alle funzionalità generiche dei tag NFC che possono essere usate in base ai contesti per dare supporto alla vista.

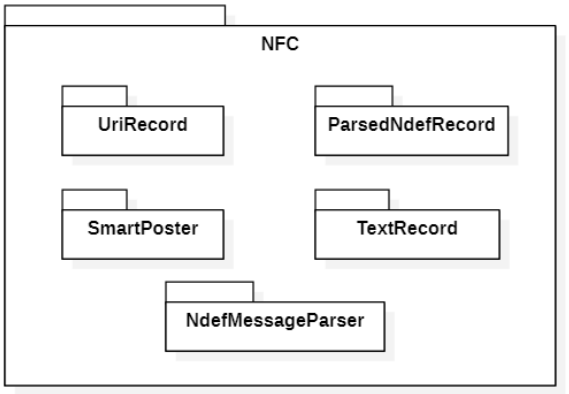

Figura 8: package-NFC

2.6 Diagramma delle classi

Nelle descrizioni che seguono si possono notare alcune mancanze, di cui illustriamo ora le motivazioni:

- non sono presenti i costruttori delle classi perché essi non hanno alcun effetto oltre a quello di creare gli oggetti;
- in ogni sottosezione vengono descritti solamente i metodi significativi per quel contesto (es. non vengono descritti ogni volta i metodi del Service qualora non venissero utilizzati).

2.6.1 Login

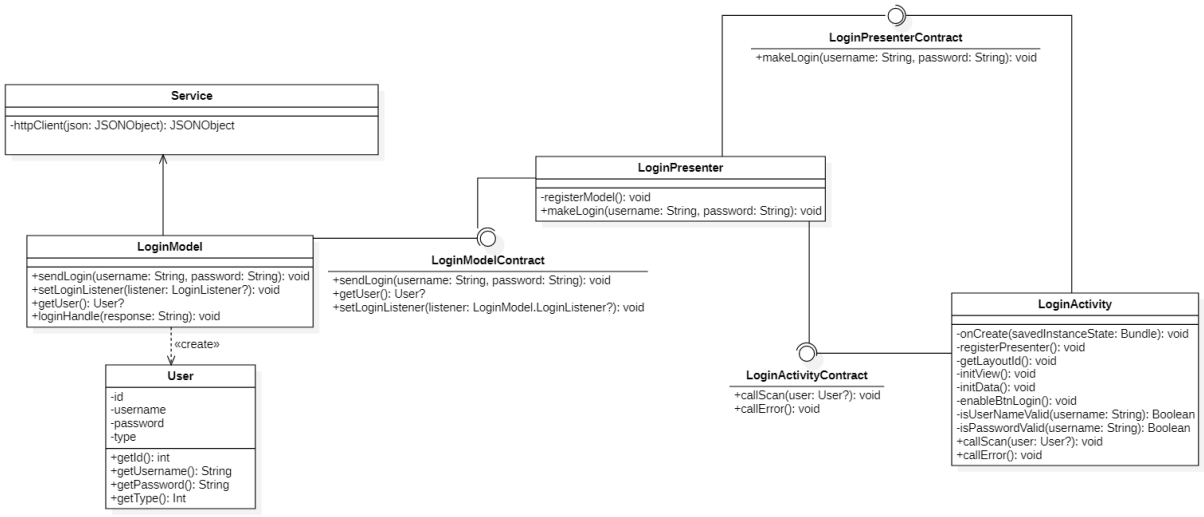

Figura 9: Login-Diagramma delle classi

La classe LoginActivity rappresenta la pagina dedicata all'autenticazione. L'utente inserisce username e password. Se registrato nel sistema l'utente potrà accedere all'applicazione altrimenti visualizzerà un messaggio d'errore.

La classe LoginActivity offre i seguenti metodi:

- **onCreate(savedInstanceState:Bundle): void**: si occupa della creazione della parte grafica visualizzata dall'utente;
- **getLayoutId(): void**: associa il rispettivo file in formato xml alla View;
- **registerPresenter(): void**: associa il Presenter alla View;
- **initView(): void**: mappa l'Activity associandole le variabili: username, password, loginBtn, errore;
- **initData(): void**: definisce l'aggiornamento della vista secondo gli input dell'utente;
- **enableBtnLogin(): void**: abilita il bottone di Login;
- **isUserNameValid(username:String): Boolean**: verifica della conformità dell'username;
- **isPasswordValid(username:String): Boolean**: verifica della conformità della password;
- **callScan(user:User?): void**: aggiorna la vista dopo la verifica delle credenziali se sono corrette.
- **callError(): void**: aggiorna la vista dopo la verifica delle credenziali se risultano errate.

2.6.2 Scan

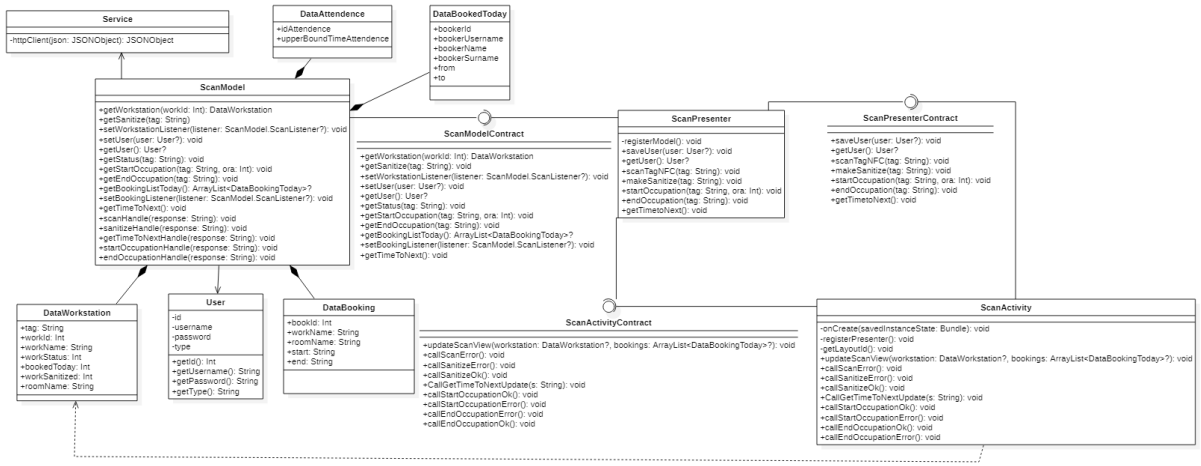

Figura 10: Main-Diagramma delle classi

La classe ScanActivity rappresenta la pagina principale dell'applicazione. Al suo interno è possibile eseguire varie operazioni premendo sul menù a tendina in alto a sinistra. La funzione principale della classe ScanActivity è la scansione dei TAG NFC$_G$ che permettono la visualizzazione dei seguenti campi:

- workName;
- workStatus.

In caso di prenotazioni, queste verranno mostrate. Un'altra funzionalità di questa attività consiste nel premere sul bottone igienizzata una volta igienizzata la postazione così da cambiare stato e poter occupare la postazione se disponibile.

La classe ScanActivity offre i seguenti metodi:

- **onCreate(savedInstanceState:Bundle): void**: si occupa della creazione della parte grafica visualizzata dall'utente;
- **registerPresenter(): void**: associa il Presenter alla View;
- **getLayoutId(): void**: associa il rispettivo file in formato xml alla View;
- **updateScanView(workstation:DataWorkstation?. bookings:ArrayList<DataBookingToday>?): void**: Si occupa di far visualizzare le caratteristiche di una postazione dopo la scansione;
- **callScanError(): void**: si occupa di mostrare un messaggio di errore nel caso la scansione non vada a buon fine;
- **callSanitizeError(): void**: si occupa di mostrare un messaggio di errore nel caso l'igienizzazione non vada a buon fine;
- **callSanitizeOk(): void**: si occupa di aggiornare la vista in seguito all'igienizzazione di una postazione;

- **CallGetTimeToNextUpdate(s:String): void**: si occupa di aggiornare la vista in base a quante ore l'utente potrà stare al massimo sulla postazione desiderata;
- **callStartOccupationOk(): void**: si occupa di aggiornare la vista in seguito all'inizio dell'occupazione;
- **callStartOccupationError(): void**: si occupa di mostrare un messaggio di errore nel caso l'inizio dell'occupazione non vada a buon fine;
- **callEndOccupationOk(): void**: si occupa di aggiornare la vista in seguito alla fine dell'occupazione;
- **callEndOccupationError(): void**: si occupa di mostrare un messaggio di errore nel caso la fine dell'occupazione non vada a buon fine.

2.6.3 Guide

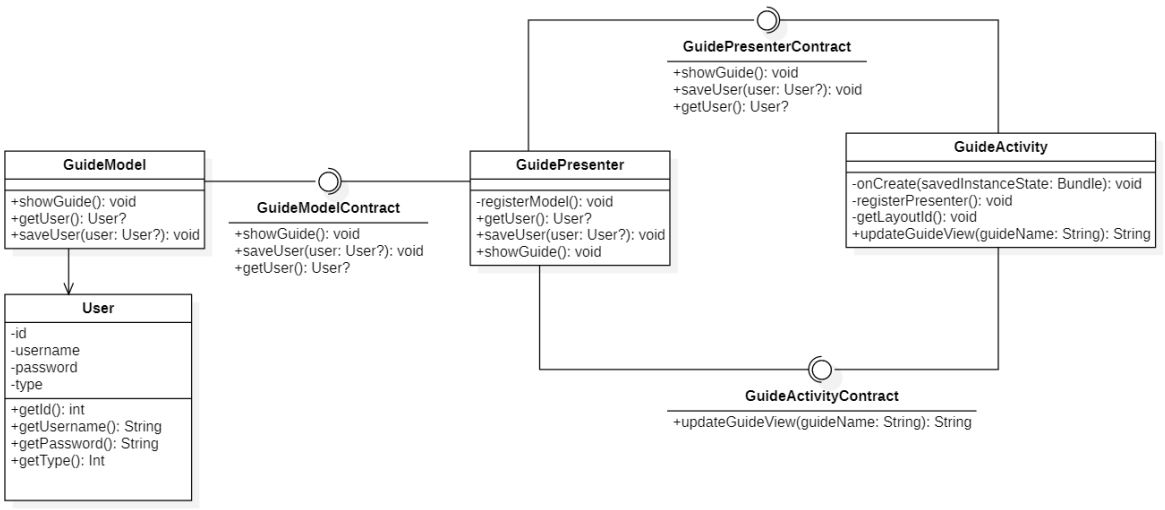

Figura 11: GuideActivity-Diagramma delle classi

La classe GuideActivity rappresenta la pagina dedicata alla visualizzazione della guida. L'utente per visualizzare la guida deve premere sul menu a tendina del menù principale(ScanActivity) e successivamente deve premere sulla voce "Guida" e verrà reindirizzato alla pagina dedicata.

In questa pagina verrà mostrata la guida dedicata all'utente. In base al suo identificativo con cui ha effettuato il login sarà possibile distinguerlo tra le due tipologie, dipendente e addetto alle pulizie, così da mostrare a ciascuno la guida corrispondente.

La classe GuideActivity offre i seguenti metodi:

- **onCreate(savedInstanceState:Bundle): void**: si occupa della creazione della parte grafica visualizzata dall'utente;
- **registerPresenter(): void**: associa il Presenter alla View;
- **getLayouId(): void**: associa il rispettivo file in formato xml alla View;

- **updateGuideView(guideName:String): String**: si occupa di far visualizzare la guida a seconda dell'utente autenticato.

2.6.4 BookingForm

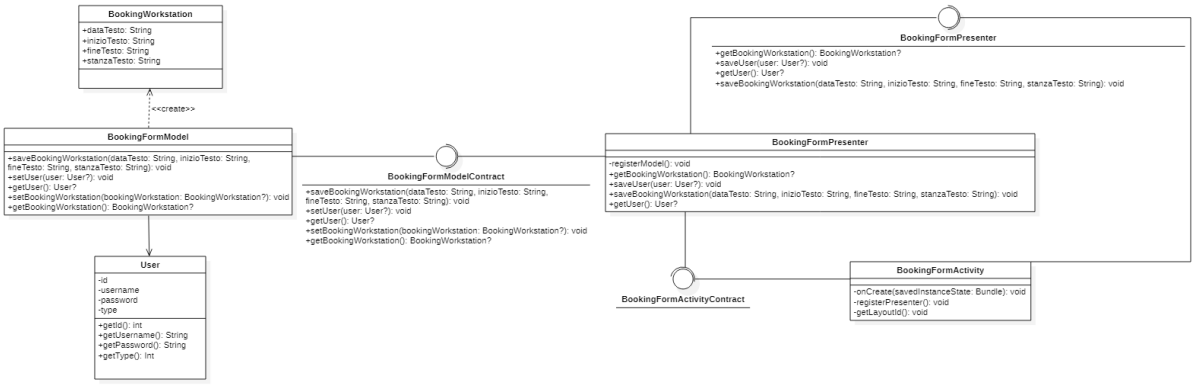

Figura 12: BookingForm-Diagramma delle classi

La classe BookingFormActivity rappresenta la pagina dedicata alla visualizzazione della ricerca di una postazione. L'utente per visualizzare questa pagina deve premere sul menu a tendina del menù principale(ScanActivity) e successivamente deve premere sulla voce "Prenota Postazione" e verrà reindirizzato alla pagina dedicata.

In questa pagina i campi da compilare sono:

- date;
- startTime;
- endTime;
- room.

La classe BookingFormActivity offre i seguenti metodi:

- **onCreate(savedInstanceState:Bundle): void**: si occupa della creazione della parte grafica visualizzata dall'utente;
- **getLayoutId(): void**: associa il rispettivo file in formato xml alla View;
- **registerPresenter(): void**: associa il Presenter alla View.

2.6.5 Bookings

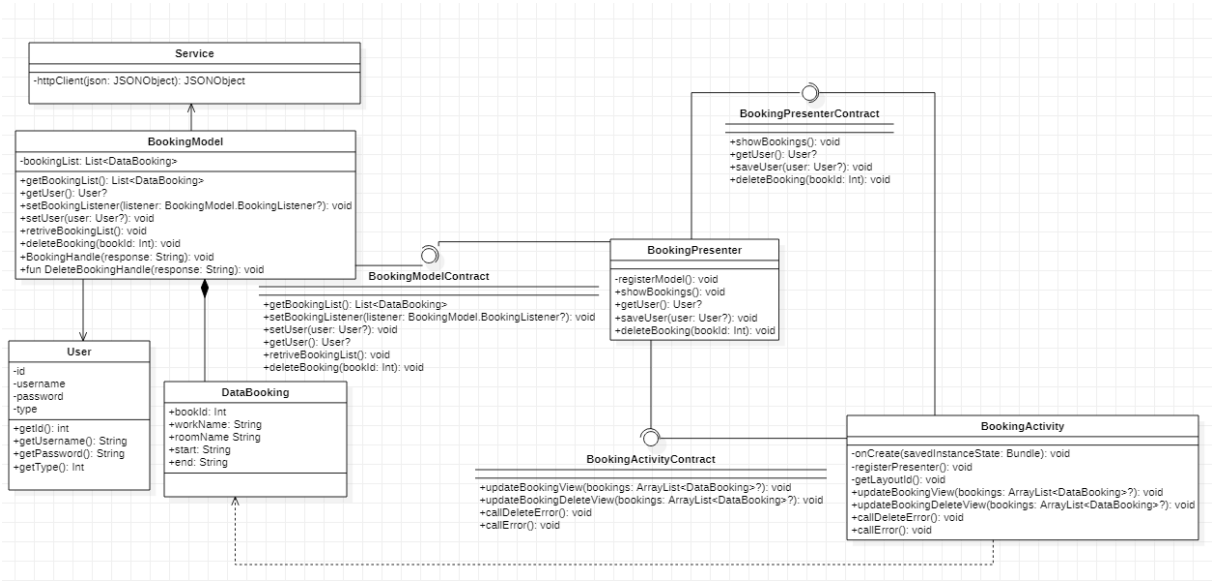

Figura 13: Bookings-Diagramma delle classi

La classe BookingActivity rappresenta la pagina dedicata alle prenotazioni effettuate da un utente. L'utente per visualizzare le prenotazioni deve premere sul menù a tendina del menù principale(ScanActivity) e successivamente deve premere sulla voce "Visualizza Prenotazioni" e verrà reindirizzato alla pagina dedicata.

In questa pagina verranno mostrate in una lista tutte le prenotazioni effettuate dall'utente. Per ogni prenotazione i campi visualizzati sono:

- bookId;
- workName;
- roomName;
- start;
- end.

Nel caso non ci siano prenotazioni effettuate verrà mostrata una semplice stringa di testo con scritto: "Non ci sono prenotazioni effettuate".

La classe BookingActivity offre i seguenti metodi:

- **onCreate(savedInstanceState:Bundle): void**: si occupa della creazione della parte grafica visualizzata dall'utente;
- **getLayoutId(): void**: associa il rispettivo file in formato xml alla View;
- **registerPresenter(): void**: associa il Presenter alla View;

- **updateBookingView(bookings:List<DataBooking>): void**: si occupa di far visualizzare le prenotazioni di un utente in una lista se ci sono;
- **callError(): void**: si occupa di far visualizzare la stringa "Non ci sono prenotazioni";
- **updateBookingDeleteView(bookings:ArrayList<DataBooking>?): void**: si occupa di aggiornare la vista nel caso una prenotazione sia eliminata premendo sull'apposito bottone;
- **callDeleteError(): void**: si occupa di far apparire un messaggio di errore nel caso l'eliminazione della prenotazione non sia riuscita.

2.6.6 Clean

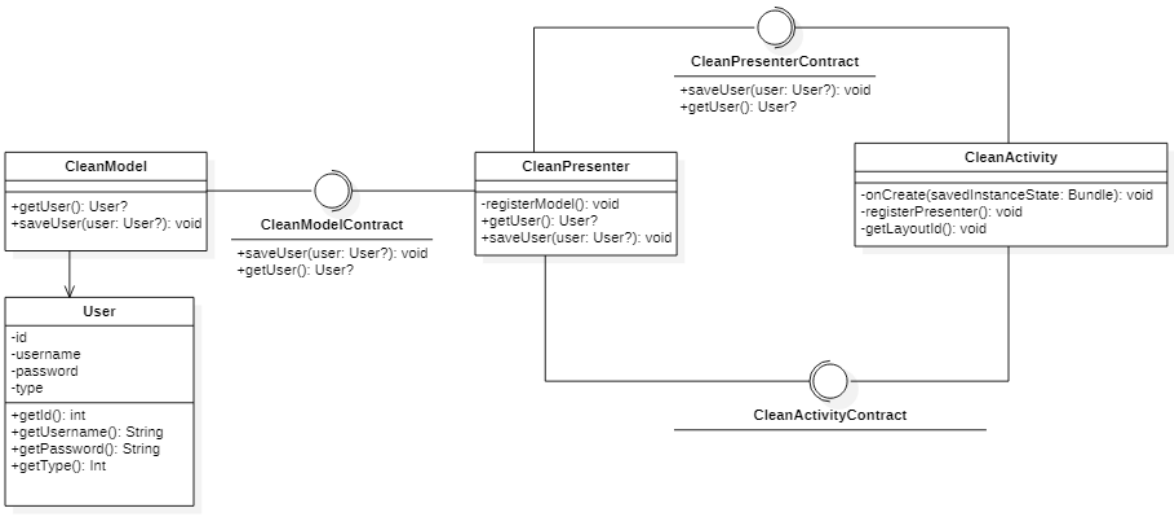

Figura 14: Clean-Diagramma delle classi

La classe CleanActivity rappresenta la pagina nella quale l'addetto alle pulizie può ricercare le postazioni e le stanze da igienizzare.

La classe CleanActivity offre i seguenti metodi:

- **onCreate(savedInstanceState:Bundle): void**: si occupa della creazione della parte grafica visualizzata dall'utente;
- **getLayoutId(): void**: associa il rispettivo file in formato xml alla View;
- **registerPresenter(): void**: associa il Presenter alla View.

2.6.7 BookingWorkstation

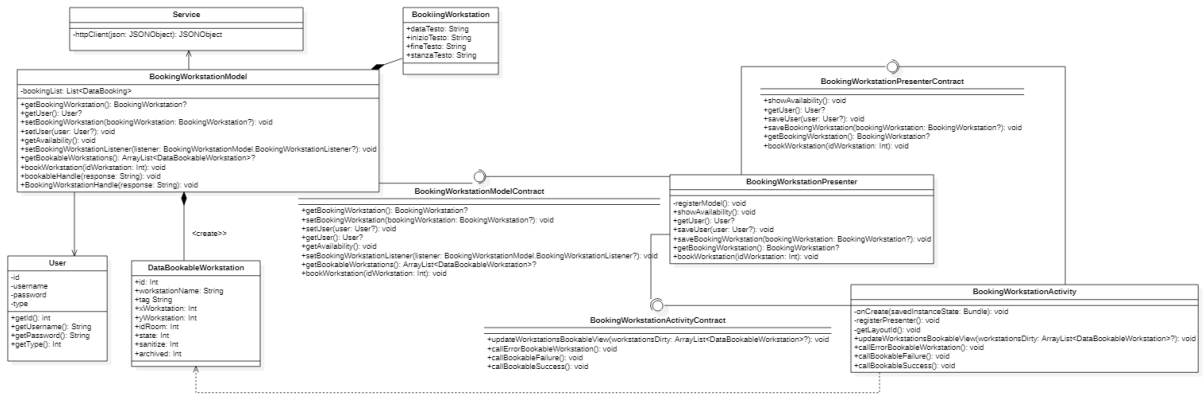

Figura 15: BookingWorkstation-Diagramma delle classi

La classe BookingWorkstationActivity rappresenta la pagina nella quale il dipendente può prenotare una postazione tra quelle disponibili.

La classe BookingWorkstationActivity offre i seguenti metodi:

- **onCreate(savedInstanceState:Bundle): void**: si occupa della creazione della parte grafica visualizzata dall'utente;
- **getLayoutId(): void**: associa il rispettivo file in formato xml alla View;
- **registerPresenter(): void**: associa il Presenter alla View;
- **updateWorkstationsBookableView(workstationsDirty:ArrayList<DataBookableWorkstation>?): void**: aggiorna la vista con una lista di postazioni prenotabili;
- **callErrorBookableWorkstation(): void**: da un messaggio di errore nel caso non ci siano postazioni prenotabili;
- **callBookableFailure(): void**: si occupa di far vedere un messaggio di errore nel caso la prenotazione non sia stata effettuata;
- **callBookableSuccess(): void**: si occupa di aggiornare la vista in seguito alla prenotazione avvenuta con successo.

2.6.8 RoomsDirty

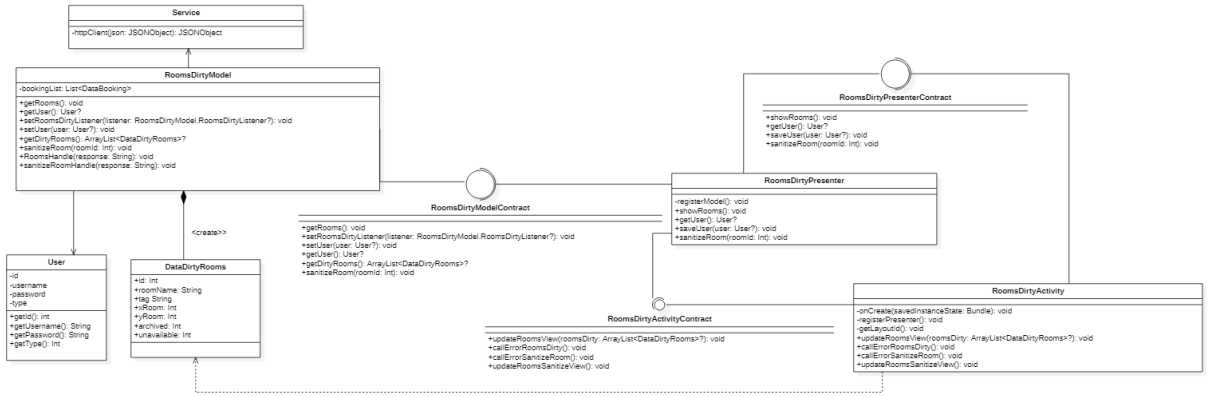

Figura 16: RoomsDirty-Diagramma delle classi

La classe RoomsDirtyActivity rappresenta la pagina nella quale l'addetto alle pulizie può igienizzare le stanze non igienizzate.

La classe RoomsDirtyActivity offre i seguenti metodi:

- **onCreate(savedInstanceState:Bundle): void**: si occupa della creazione della parte grafica visualizzata dall'utente;
- **getLayoutId(): void**: associa il rispettivo file in formato xml alla View;
- **registerPresenter(): void**: associa il Presenter alla View;
- **updateRoomsView(roomsDirty:ArrayList<DataDirtyRooms>?): void**: aggiorna la vista con una lista di stanze igienizzabili;
- **callErrorRoomsDirty(): void**: da un messaggio di errore nel caso non ci siano stanze igienizzabili;
- **callErrorSanitizeRoom(): void**: ritorna un messaggio se l'igienizzazione di una stanza non è avvenuta con successo;
- **updateRoomsSanitizeView(): void**: si occupa di aggiornare la vista in seguito all'igienizzazione di una stanza avvenuta con successo.

2.6.9 WorkstationsDirty

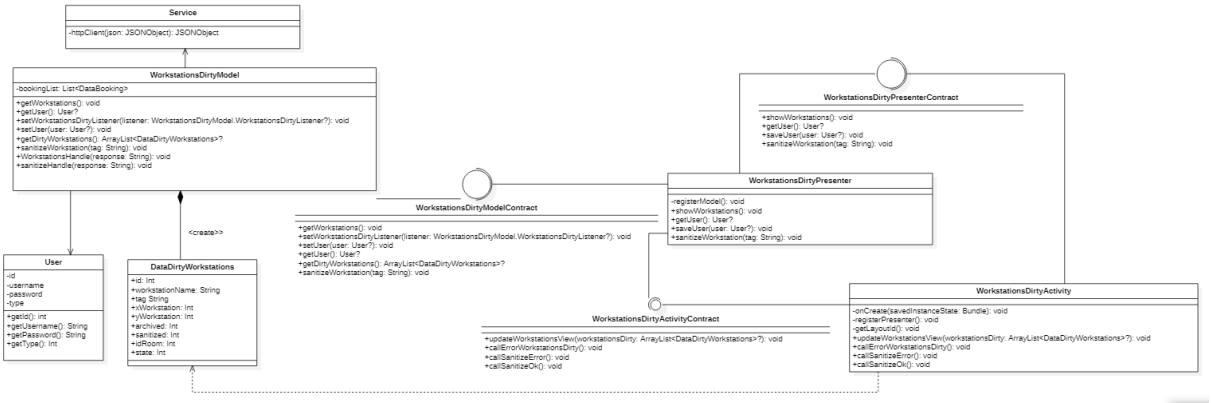

Figura 17: WorkstationsDirty-Diagramma delle classi

La classe WorkstationsDirtyActivity rappresenta la pagina nella quale l'addetto alle pulizie può igienizzare le postazioni non igienizzate.

La classe WorkstationsDirtyActivity offre i seguenti metodi:

- **onCreate(savedInstanceState:Bundle): void**: si occupa della creazione della parte grafica visualizzata dall'utente;
- **getLayoutId(): void**: associa il rispettivo file in formato xml alla View;
- **registerPresenter(): void**: associa il Presenter alla View;
- **updateWorkstationsView(workstationsDirty:ArrayList<DataDirtyWorkstations>?): void**: aggiorna la vista con una lista di postazioni igienizzabili;
- **callErrorWorkstationsDirty(): void**: da un messaggio di errore nel caso non ci siano postazioni igienizzabili;
- **callSanitizeError(): void**: ritorna un messaggio se l'igienizzazione di una postazione non è avvenuta con successo;
- **callSanitizeOk(): void**: si occupa di aggiornare la vista in seguito all'igienizzazione di una postazione avvenuta con successo.

2.7 Diagramma di sequenza

Vengono presentati di seguito i diagrammi UML di sequenza relativi all'applicazione.

2.7.1 Diagramma per il login

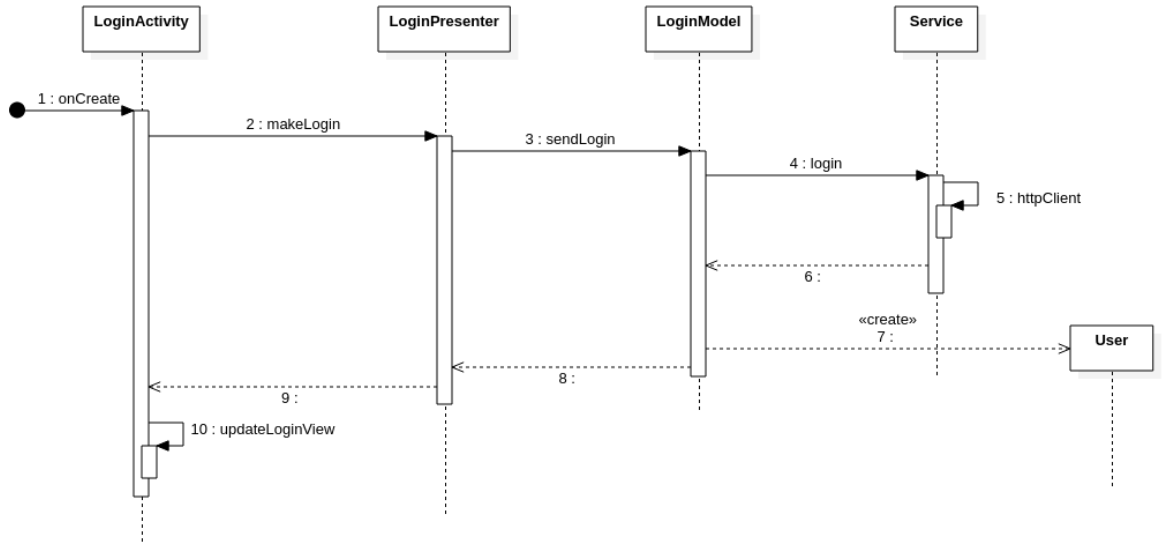

Figura 18: Login-Diagramma di sequenza

Questo diagramma di sequenza rappresenta la serie di azioni svolte dall'applicazione per autenticare un utente. La sequenza inizia con la creazione della vista tramite il metodo *onCreate*. Successivamente viene richiesto al Presenter di effettuare l'autenticazione tramite il metodo *makeLogin* che a sua volta richiederà al Model di effettuare l'autenticazione attraverso il metodo *sendLogin*. Il Model chiama *login* del Service tramite il quale genera il json con username e password. Ora il Service chiede di verificare al backend se esiste un utente associato a quelle credenziali tramite il metodo *httpClient*. Qualora l'utente esistesse viene creato un oggetto di classe User con l'id restituito da *httpClient* e avvengono tutte le chiamate di ritorno. Infine viene aggiornata la vista tramite il metodo *updateLoginView*.

2.7.2 Diagramma per la visualizzazione delle prenotazioni di un utente

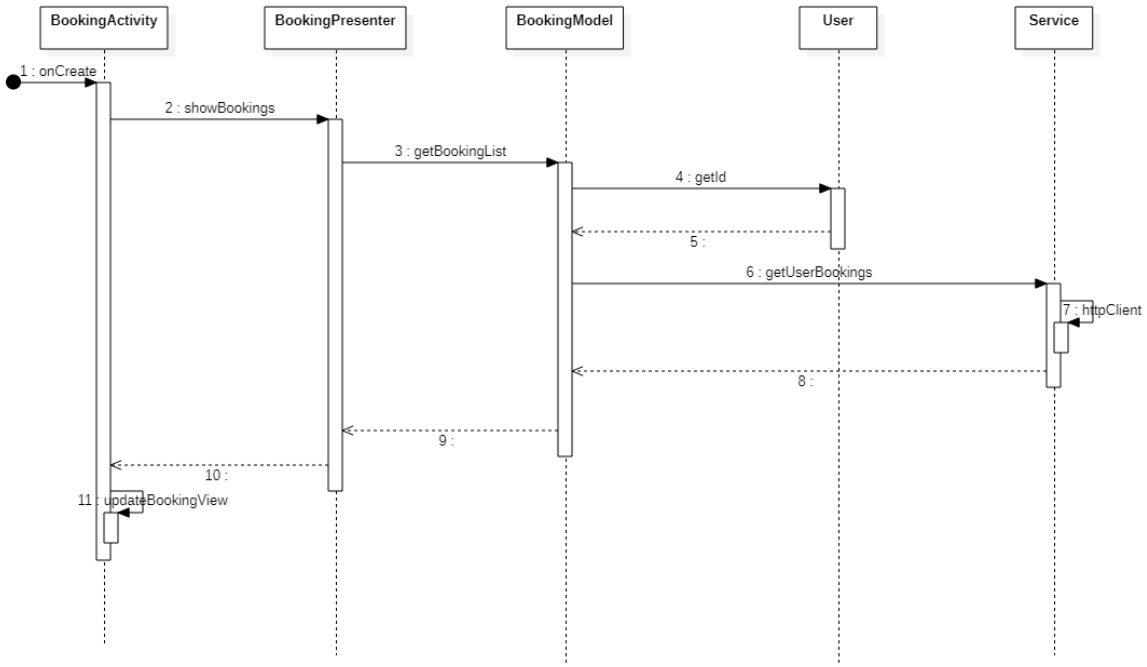

Figura 19: ShowBookings-Diagramma di sequenza

Questo diagramma di sequenza rappresenta la serie di azioni svolte dall'applicazione quando viene effettuata la richiesta di visualizzare le prenotazioni da un utente. La sequenza inizia con la creazione della vista tramite il metodo *onCreate*. Successivamente viene richiesto al Presenter di visualizzare le prenotazioni tramite il metodo *showBookings* che a sua volta richiederà al Model la lista delle prenotazioni attraverso il metodo *getBookingList*. Il Model per ottenere la lista delle prenotazioni farà prima una chiamata alla classe User con il metodo *getId* per ottenere l'id dello user. Il metodo successivo che deve usare è *getUserBookings* con il quale genera il json con l'id dello user passandolo così al Service. Il compito del Service è di ottenere la lista delle prenotazioni grazie al json trasmesso dal Model e per farlo effettuerà una richiesta al server tramite il metodo *httpClient*. In seguito verranno fatte tutte le chiamate di ritorno, ovvero verrà ritornata la lista delle prenotazioni al Model che la ritornerà al Presenter che la ritornerà a sua volta alla vista. In conclusione per visualizzare la lista delle prenotazioni la vista chiamerà il metodo *updateBookingView*.

2.7.3 Diagramma per ottenere le caratteristiche di una postazione

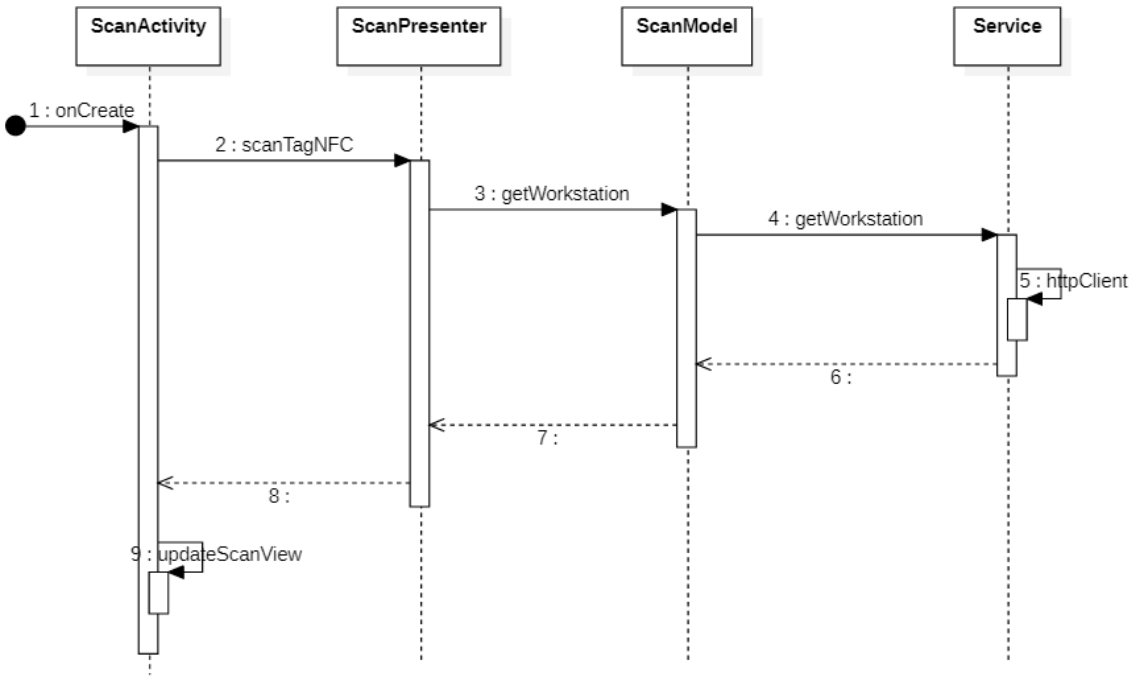

Figura 20: Scan-Diagramma di sequenza

Questo diagramma di sequenza rappresenta la serie di azioni svolte dall'applicazione quando viene scansionato un tag NFC da un utente. La sequenza inizia con la creazione della vista tramite il metodo *onCreate*. Successivamente viene richiesto al Presenter di visualizzare le caratteristiche delle postazioni in seguito alla scansione del tag NFC tramite il metodo *scanTagNFC* che a sua volta le richiederà al Model attraverso il metodo *getWorkstation*. Il Model, per ottenere le informazioni di una postazione, otterrà l'id della Workstation grazie al tag associato ad esso. Successivamente grazie al metodo *getWorkstation*, che genera un json con l'id della postazione, andrà a richiedere al Service tutte le caratteristiche. Per ottenere queste informazioni il Service userà il metodo *httpClient*. In seguito verranno fatte tutte le chiamate di ritorno, ovvero le caratteristiche della postazione verranno ritornate al Model che le ritornerà al Presenter che le ritornerà a sua volta alla vista. In conclusione per visualizzare le informazioni di una postazione la vista chiamerà il metodo *updateScanView*.

2.7.4 Diagramma per registrare l'igienizzazione di una postazione

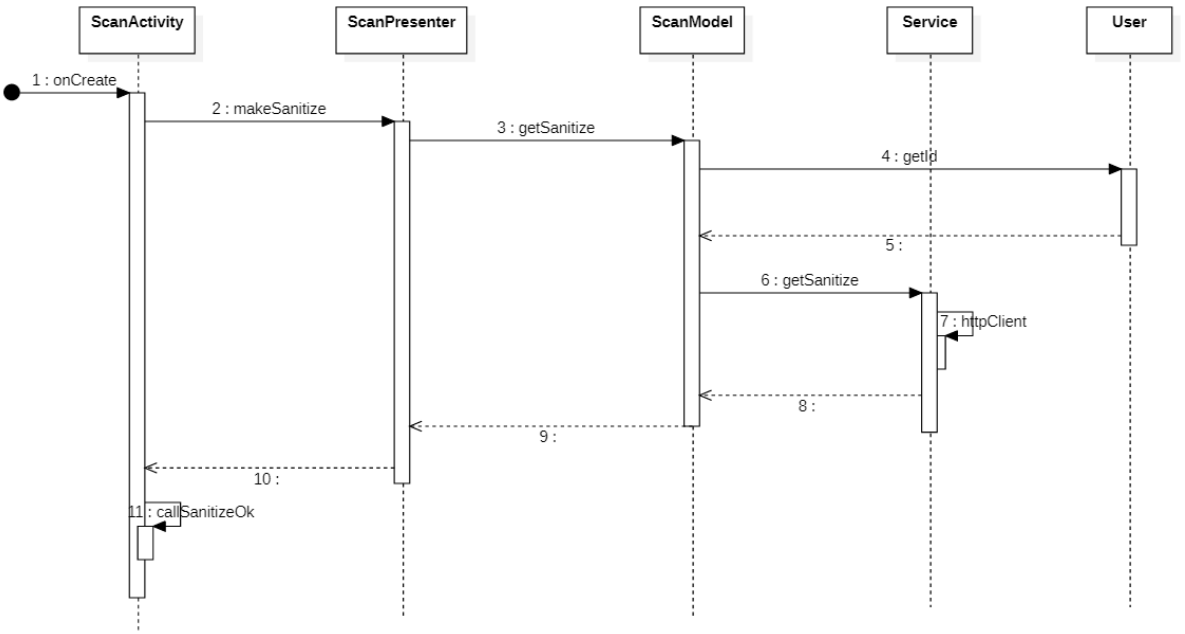

Figura 21: Sanitize-Diagramma di sequenza

Questo diagramma di sequenza rappresenta la serie di azioni svolte dall'applicazione quando viene premuto da un utente il bottone per rendere una postazione igienizzata. La sequenza inizia con la creazione della vista tramite il metodo *onCreate*. Successivamente viene richiesto al Presenter di registrare l'igienizzazione della postazione da parte di un utente tramite il metodo *makeSanitize* che a sua volta lo richiederà al Model attraverso il metodo *getSanitize*. Il Model, per igienizzare una postazione, otterrà prima l'id della Workstation grazie al tag associato ad esso e poi l'id dello user dalla classe User con il metodo *getId*. Successivamente grazie al metodo *getSanitize*, che genera un json con l'id della postazione e l'id dell'utente, andrà a richiedere al Service di registrarla. Per registrarla il Service userà il metodo *httpClient*. In seguito verranno fatte tutte le chiamate di ritorno, ovvero il risultato dell'operazione verrà ritornato al Model che lo ritornerà al Presenter che lo ritornerà a sua volta alla vista. In conclusione per visualizzare le nuove informazioni sulla postazione la vista chiamerà il metodo *callSanitizeOk*.

3 Web-app amministratori

3.1 Introduzione

La web app costituisce l'interfaccia tramite la quale l'amministratore può interagire col sistema. Le funzionalità offerte sono:

- login e logout;
- visualizzazione delle stanze e delle postazioni con i relativi stati;
- aggiunta, rimozione e modifica di stanze e postazioni;
- impostazione di postazioni come guaste;
- impostazione di stanze come inaccessibili;
- visualizzazione delle credenziali degli utenti;
- aggiunta, rimozione e modifica di credenziali;
- visualizzazione e scaricamento report sulle occupazioni e sulle igienizzazioni;
- visualizzazione di notifiche riguardanti il salvataggio dei dati sulla blockchain.

3.2 Requisiti e installazione

3.2.1 Ottenimento codice

I file sorgente della web app si trovano su GitHub all'indirizzo https://github.com/DPCMGroup/bc19-webapp. Essi si possono scaricare compressi in formato zip direttamente dal sito, oppure tramite il comando
`git clone https://github.com/DPCMGroup/bc19-webapp`
se si ha già installato il software di versionamento Git.

3.2.2 Linguaggi

3.2.2.1 Typescript

Superset del linguaggio JAVASCRIPT$_G$. Viene utilizzato per definire la logica dell'applicazione.
Per poterlo utilizzare con ANGULAR$_G$, esso va prima installato col comando
`npm install -g typescript`

3.2.2.2 HTML

Linguaggio di markup utilizzato per definire la struttura delle pagine web. Non necessita di installazione.

3.2.2.3 CSS

Linguaggio per la definizione dello stile grafico delle pagine web. Nella nostra applicazione il codice CSS$_G$ introdotto da noi è molto limitato perché, per questo, ci siamo affidati al FRAMEWORK$_G$ Bootstrap. CSS non necessita di installazione.

3.2.3 Tecnologie

3.2.3.1 Node.js

Programma focalizzato sull'esecuzione di codice javascript al di fuori di un browser. Per l'installazione fare riferimento alla pagina https://nodejs.org/en/download/. La versione utilizzata per lo sviluppo è la 10.24.0.

3.2.3.2 npm

Acronimo di Node Package Manager, permette di ottenere le librerie necessarie allo sviluppo. Si ottiene insieme a NODE.JS$_G$ tramite l'installazione di quest'ultimo. La versione utilizzata per lo sviluppo è la 7.6.2.

3.2.3.3 Angular

Framework per applicazioni web. Per l'installazione fare riferimento alla pagina https://angular.io/guide/setup-local#install-the-angular-cli. La versione utilizzata per lo sviluppo è la 11.2.7.

3.2.3.4 Bootstrap

Framework per la creazione di pagine web. Per l'integrazione di Bootstrap in Angular fare riferimento alla pagina https://www.npmjs.com/package/@ng-bootstrap/ng-bootstrap#installation. La versione utilizzata per lo sviluppo è la 4.5.0.

3.2.4 Test

Per i test abbiamo utilizzato il framework Jasmine. Per utilizzarlo con Angular è necessario installarlo col comando

```
npm install -g jasmine
```

La versione di Jasmine utilizzata per lo sviluppo è la 2.8.0.
Per eseguire i test usare il comando

```
ng test
```

I test verranno eseguiti tramite il programma Karma e i loro risultati saranno visualizzati in una pagina del browser aperta automaticamente.

3.2.5 Ambiente di sviluppo

Per lo sviluppo abbiamo utilizzato l'IDE WebStorm. Per la sua installazione riferirsi alla pagina https://www.jetbrains.com/webstorm/download/. La versione utilizzata per lo sviluppo è la 2020.3.3.

3.2.6 Esecuzione

Una volta installate le tecnologie sopra elencate si potrà eseguire la webapp aprendo un terminale nella root del progetto ed eseguendo il comando
`ng serve`
Verrà eseguito un server locale accessibile all'indirizzo http://localhost:4200

3.3 Architettura

L'architettura della web-app segue il modello a componenti imposto da Angular, che, a sua volta, si basa sul pattern Model-View-ViewModel, descritto dall'immagine sottostante.

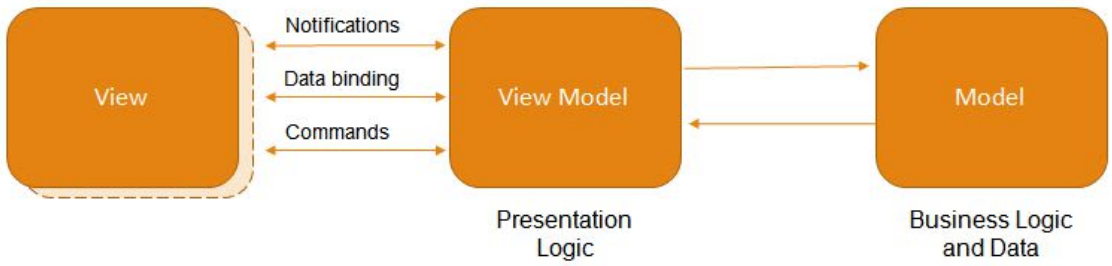

Figura 22: Model-View-ViewModel

Il modello imposto da Angular prevede che ogni funzionalità disponibile all'utente sia costituita da una vista (view) e da una logica (viewmodel) e chiama l'insieme di queste due parti "componente". Gli attributi del componente sono accessibili alla vista. Questa caratteristica permette una sincronizzazione in due versi: la vista, per mano dell'utente, può modificare le variabili del view model, e se quest'ultimo modifica le proprie variabili, la vista si modifica di conseguenza. Oltre ai componenti Angular offre la possibilità di creare dei servizi, che costituiscono la terza componente del pattern: il model. Questi servizi si occupano della gestione dei dati. Per esempio può esserci un servizio per la comunicazione col database, uno per la lettura e scrittura dal local storage e un altro che implementa un algoritmo per la manipolazione dei dati. Un componente è costituito da 4 file. Nel seguente esempio è rappresentato un componente chiamato base.

```
base.component.css
base.component.html
base.component.spec.ts
base.component.ts
```

I file contengono, nell'ordine:

- lo stile grafico della vista del componente;
- la struttura della vista del componente;
- i test definiti per il componente;
- il codice che definisce la logica del componente.

I servizi sono definiti solo da due file. Di seguito sono mostrati i due file prendendo come esempio un servizio chiamato login.

`login.service.spec.ts`
`login.service.ts`

I file contengono, nell'ordine:

- i test definiti per il servizio;
- il codice che definisce la logica del servizio.

La struttura del sito è definita principalmente dal modo in cui i componenti vengono annidati l'uno dentro all'altro. Nel codice HTML$_G$ questo annidamento è descritto dai tag:

- `<app-nomecomponente>`
- `<router-outlet>`

In entrambi i casi, nella pagina visualizzata, verrà inserito il componente specificato al posto del tag. Nel primo caso il componente da inserire è fisso ed è definito dal nome del tag stesso, nel secondo invece il componente inserito può variare. In quest'ultimo caso l'annidamento è definito dal'url col quale si sta visitando la pagina e dalla costante `routes` all'interno del file `src/app/app-routing.module.ts`.

3.4 Estendibilità

Il modello a componenti offerto da Angular, descritto nella precedente sottosezione, facilita l'estendibilità dell'applicazione. Per introdurre una nuova funzionalità sarà infatti sufficiente creare un componente apposito tramite il comando
`ng g c nomecomponente`
Per introdurre invece un servizio si dovrà usare il comando
`ng g service nomeservizio`
Per la struttura dei componenti e dei servizi fare riferimento alla sottosezione precedente.

3.5 Diagrammi dei package

3.5.1 Visione generale

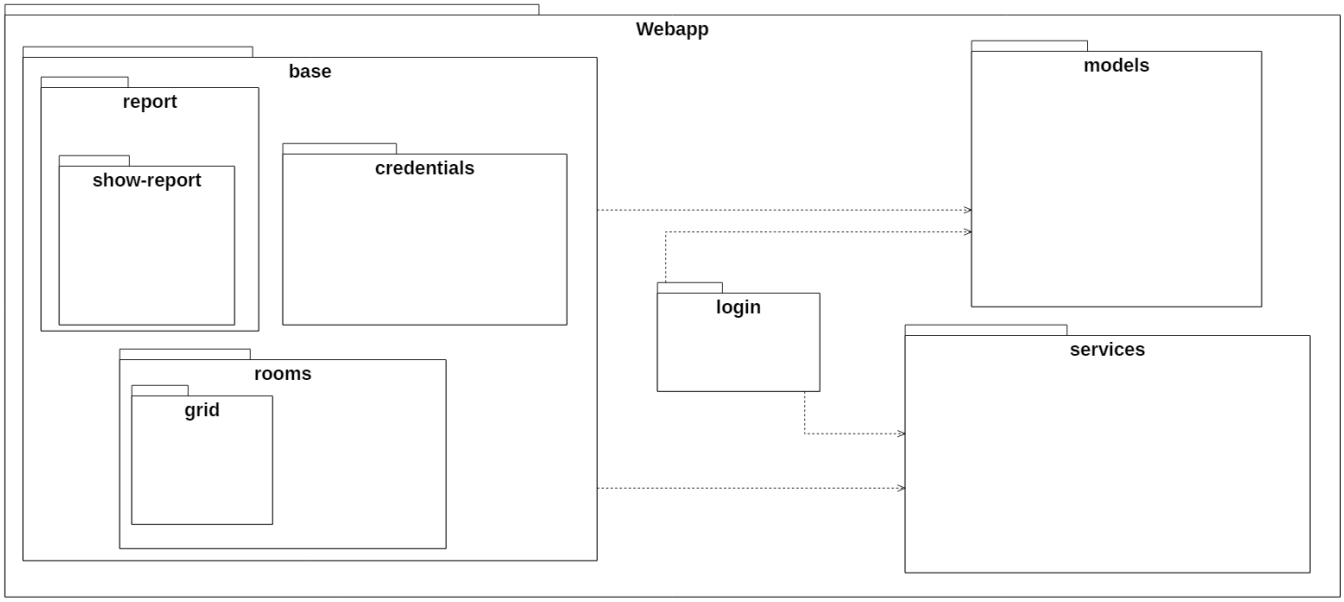

Figura 23: Diagramma dei package della webapp

Il diagramma sovrastante rappresenta la struttura e le relazioni dei package della webapp.

3.5.2 Base

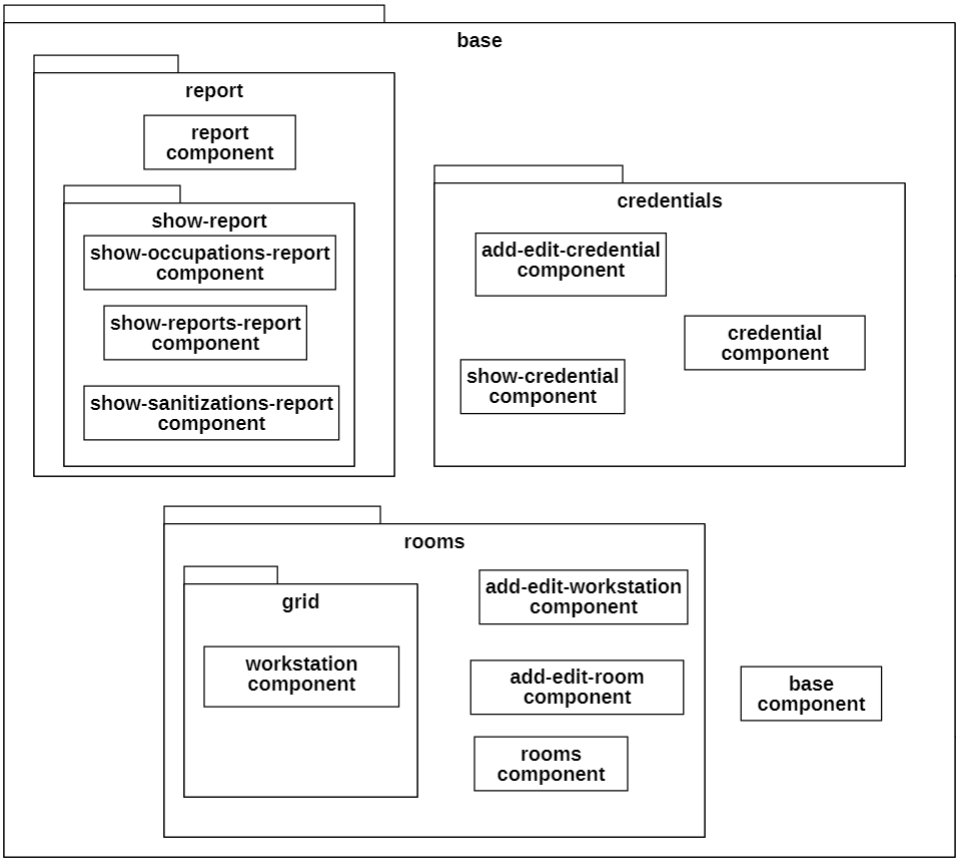

Figura 24: Diagramma del package base della webapp

Il package Base contiene tutte le funzionalità offerte a un amministratore autenticato. Esse sono ulteriormente suddivise in tre package:

- Credentials;
- Rooms;
- Report.

All'interno del package Credentials sono presenti tutti e soli i componenti utilizzati nella pagina di gestione delle credenziali. All'interno del package Rooms sono presenti tutti e soli i componenti utilizzati nella pagina di gestione delle stanze e delle postazioni. All'interno del package Report sono presenti tutti e soli i componenti utilizzati nella pagina di gestione dei report. Le tre pagine appena indicate sono annidate nella vista del componente Base. Quest'ultimo infatti presenta dei pulsanti per la selezione della pagina da visualizzare. Esso presenta inoltre, sempre tramite un pulsante, la funzionalità di logout e di visulizzazione della guida dell'applicazione web.
Il package Base ha una dipendenza verso Models, le cui classi vengono usate per organizzare i dati, e una verso Services, le cui classi vengono usate per interagire con il server e con il local storage. All'interno del package Rooms è presente il package Grid. Esso si occupa di gestire tutto quello che riguarda la visualizzazione schematica delle postazioni all'interno di una griglia.

3.5.3 Login

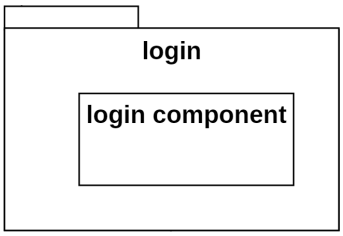

Figura 25: Diagramma del package login della webapp

Il package Login contiene le funzionalità necessarie per l'autenticazione dell'utente.

3.5.4 Models

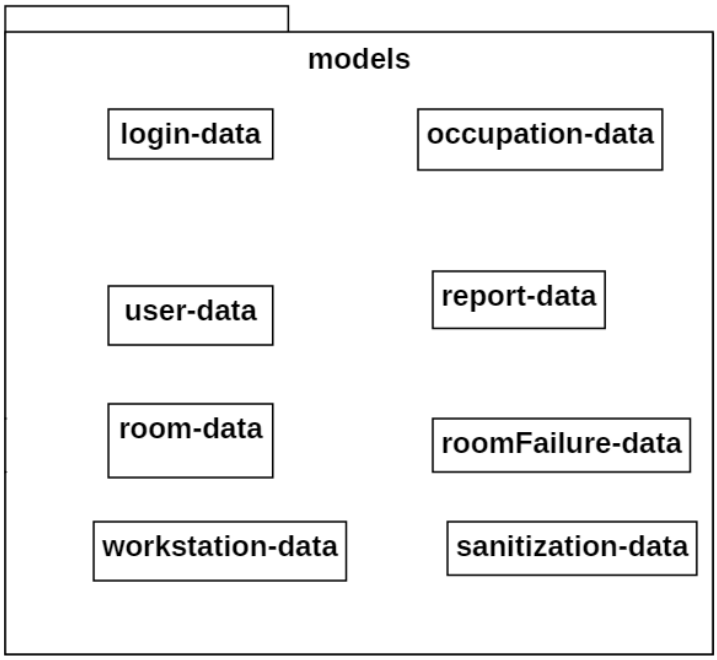

Figura 26: Diagramma del package models della webapp

Il package Models contiene le classi che vengono usate nel resto dell'applicazione per formattare i dati e comunicarli in modo ordinato e coeso.

3.5.5 Services

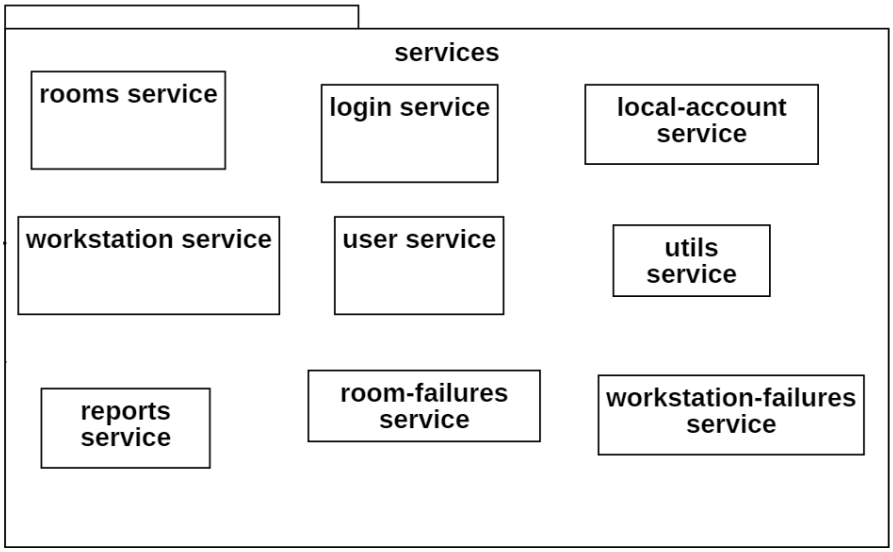

Figura 27: Diagramma del package services della webapp

Il package Services contiene le classi usate nel resto dell'applicazione per interagire col server e con il local storage oppure che fungono da contenitori di algoritmi comuni.

3.6 Diagrammi delle classi

Nei diagrammi e nelle descrizioni che seguono si possono notare alcune mancanze, di cui illustriamo ora le motivazioni:

Diagrammi

- non sono rappresentate le classi HttpClient, Observable, UtilsService e le dipendenze verso di esse perchè sono molto comuni ed è quindi poco utile rappresentarle singolarmente. In particolare:
  - **HttpClient**: classe fornita da Angular per eseguire chiamate http. Da questa classe dipendono tutti i servizi che si connettono al backend.
  - **Observable**: classe fornita dalla libreria rxjs (Reactive Extensions) per ricezione di dati in modo asincrono. Ne sono dipendenti tutti servizi che si connettono al server e tutti i componenti che utilizzano quei servizi.
  - **UtilsService**: servizio che offre alcuni algoritmi comuni. Ne dipendono quasi tutti i componenti e nessun servizio.

Descrizioni

- non sono presenti i costruttori delle classi perché essi non hanno alcun effetto oltre a quello di creare gli oggetti;

- per le classi del package Models non sono descritti gli attributi poiché essi sono semplicemente la copia delle informazioni contenute nel database;
- per le classi legate a un form accessibile all'utente non sono descritti gli attributi inseribili perché essi sono un sottoinsieme di quelli di cui si parla al punto precedente.

3.6.1 Login

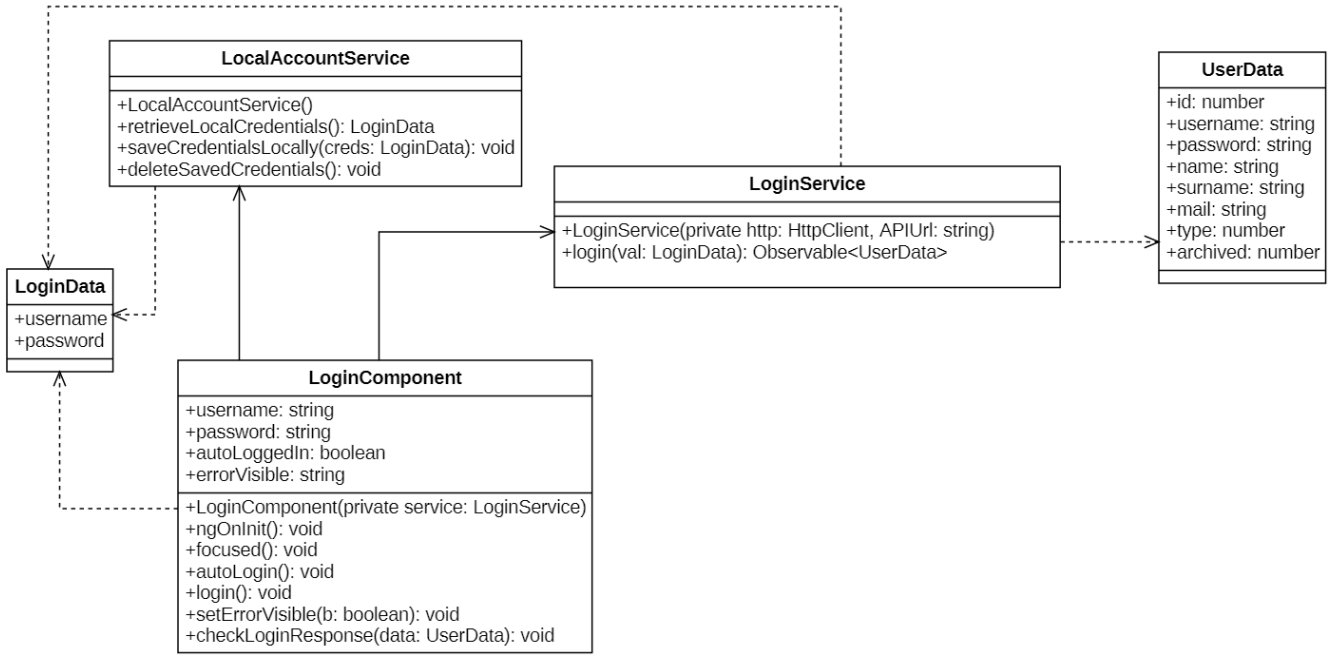

Figura 28: Diagramma delle classi per il login

Nel diagramma sovrastante sono rappresentate le classi utilizzate dalla pagina di login e le dipendenze tra di esse. La classe `LoginComponent` dirige la gestione del login manuale, tramite inserimento delle credenziali da parte dell'utente, e di quello automatico, tramite verifica delle credenziali precedentemente salvate nel local storage del browser. A `LoginService`, in particolare, viene delegata la verifica delle credenziali tramite il server, mentre a LocalAccountService, vengono delegate la scrittura, la lettura e l'eliminazione delle credenziali salvate nel browser. Per comunicare credenziali di accesso da una classe all'altra, viene utilizzata `LoginData`. `UserData` invece viene usata per la ricezione dal server dei dati dell'utente che si è autenticato.

3.6.1.1 Classi

LoginComponent

Classe che si occupa della gestione del login manuale, ovvero tramite l'inserimento delle credenziali da parte dell'utente, e di quello automatico, ovvero tramite l'utilizzo delle credenziali precedentemente salvate nel local storage del browser.
Questa classe contiene i seguenti attributi:

- **autoLoggedIn: boolean**
  Variabile utilizzata per memorizzare il successo o meno del login automatico.
- **errorVisible: string**
  Variabile che determina la visibilità dell'errore mostrato all'utente.

Questa classe presenta i seguenti metodi:

- **ngOnInit(): void**
  Metodo che viene chiamato all'inizializzazione della pagina. Provoca l'evento di login automatico.
- **focused(): void**
  Metodo che viene chiamato nel momento in cui uno dei due campi di inserimento delle credenziali viene selezionata. Provoca l'impostazione dell'invisibilità dell'errore che indica l'incorrettezza delle credenziali inserite.
- **autoLogin(): void**
  Metodo che effettua un tentativo di autenticazione con le credenziali salvate nel local storage. Se l'autenticazione riesce l'utente viene reindirizzato alla pagina home. Altrimenti viene visualizzata la pagina di inserimento delle credenziali.
- **login(): void**
  Metodo che effettua un tentativo di autenticazione con le credenziali inserite dall'utente. Se l'autenticazione riesce l'utente viene reindirizzato alla pagina home. Altrimenti viene mostrato un messaggio di errore.
- **setErrorVisible(b: boolean): void**
  Metodo che imposta la visibilità dell'errore che indica l'incorrettezza delle credenziali inserite.
- **checkLoginResponse(data: UserData): void**
  Metodo che indirizza l'utente alla pagina home se l'autenticazione ha avuto successo e mostra il form d login se l'autenticazione non ha avuto successo.

LoginService

Servizio che permette di verificare tramite il server la validità di credenziali per l'autenticazione. Ogni metodo di questa classe, tranne il costruttore, restituisce un `Observable`. Questo oggetto espone un metodo subscribe con il quale si potrà impostare una funzione di callback che verrà chiamata quando il server fornirà una risposta.
Questa classe contiene i seguenti attributi:

- **APIUrl: string** L'url base dell'API REST a cui il servizio fa riferimento.

Questa classe presenta i seguenti metodi:

- **login(val: LoginData): Observable<UserData>**
  Verifica l'esistenza delle credenziali passate come parametro nel database. Restituisce un Observable che permette di ottenere, in un `UserData`, i dati dell'utente corrispondente alle credenziali.

LocalAccountService

Servizio che permette di gestire il salvataggio delle credenziali dell'utente nel local storage del browser. Questa classe presenta i seguenti metodi:

- **retrieveLocalCredentials(): LoginData**
  Metodo che restituisce le credenziali salvate nel local storage. Se assenti restituisce un oggetto LoginData con i valori entrambi a null.
- **saveCredentialsLocally(creds: LoginData): void**
  Metodo che salva le credenziali passate nel local storage.
- **deleteSavedCredentials(): void**
  Metodo che elimina le credenziali salvate nel local storage.

LoginData

Classe che contiene i valori necessari per l'autenticazione di un amministratore al sistema. Viene usata per la comunicazione di questo tipo di dati all'interno dell'applicazione.

UserData

Classe che contiene tutti e soli i valori necessari per il salvataggio di un utente sul database. È utilizzata per il transito dei dati degli utenti all'interno dell'applicazione.

3.6.2 Aggiunta e modifica stanze

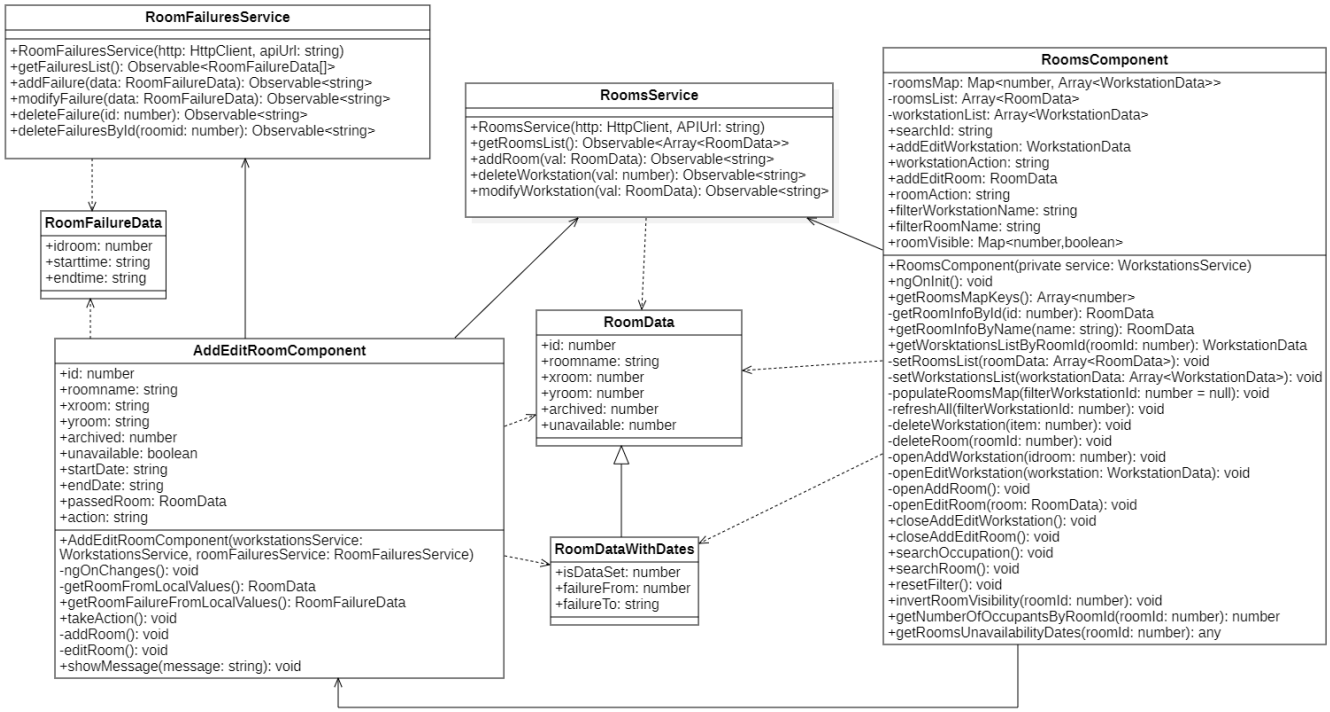

Figura 29: Diagramma delle classi per la gestione delle stanze e delle postazioni

Il diagramma sovrastante rappresenta l'architettura utilizzata per l'aggiunta e modifica delle stanze. `RoomsComponent` si occupa della visualizzazione ed eliminazione delle stanze. Essa delega la creazione e modifica di questi due dati alla classe `AddEditRoomComponent`. Le due componenti contengono riferimenti ai servizi di cui necessitano. I due servizi coinvolti sono `RoomService` e `RoomFailuresService`. Le informazini vengono ottenute dal server tramite la classe `Observable`. In generale i dati riguardanti le stanze sono scambiati utilizzando le due classi `RoomData` e `RoomDataWithDates`.

L'aggiunta e modifica delle postazioni è molto simile al caso qui presentato. Pertanto esse non vengono illustrate.

3.6.2.1 Classi

RoomsComponent

Classe che si occupa della visualizzazione, eliminazione e, tramite i componenti AddEditWorkstation e AddEditRoom, dell'aggiunta e modifica di postazioni e stanze.
Questa classe contiene i seguenti attributi:

- **roomsMap: Map<number, Array<WorkstationData> >**
  Map in cui ad ogni stanza sono assegnate le sue postazioni, ovvero quelle con `idroom` uguale all'`id` della stanza. Se l'utente applica un filtro alla lista, qui verranno contenute solo le stanze e le postazioni selezionate da tale filtro.

- **roomsList: Array<RoomsData>**
  Array utilizzato per contenere le stanze ottenute dal server.
- **workstationList: Array<Workstation>**
  Array utilizzato per contenere le postazioni ottenute dal server.
- **searchId: string**
  Id specificato dall'utente per la ricerca di una postazione.
- **addEditWorkstation: WorkstationData**
  Contiene i dati della postazione che verrà aggiunta o modificata.
- **workstationAction: string**
  Valore specificato dall'utente che provocherà la modifica o l'aggiunta della postazione salvata in `addEditWorkstation`.
- **addEditRoom: RoomData**
  Contiene i dati della stanza che verrà aggiunta o modificata.
- **roomAction: string**
  Valore specificato dall'utente che provocherà la modifica o l'aggiunta della stanza salvata in `addEditRoom`.

Questa classe presenta i seguenti metodi:

- **ngOnInit(): void**
  Questa funzione viene chiamata all'inizializzazione della pagina e provoca l'aggiornamento di roomsMap.
- **getRoomsMapKeys(): Array<number>**
  Restituisce un array con gli id delle stanze contenute in `roomsMap`.
- **getRoomInfoById(id: number): RoomData**
  Restituisce un oggetto `RoomData` con i dati della stanza identificata dall'id passato come parametro.
- **setRoomList(roomData: Array<RoomData>): void**
  Reinizializza `roomList` con i dati ricevuti
- **setWorkstationList(workstationData: Array<WorkstationData>): void**
  Reinizializza `workstationList` con i dati ricevuti
- **populateRoomsMap(filterWorkstationId: number): void**
  Reinizializza `roomsMap` ponendo come chiavi gli id delle stanze contenute in `roomsList` e come valori le postazioni contenute nelle rispettive stanze. In questo modo sarò facile ottenere le postazioni contenute in una stanza conoscendo l'id di quest'ultima.
  Se non viene specificato l'attributo `filterWorkstationId` nella mappa vengono inserite anche le stanze vuote.
  Se viene specificato l'attributo `filterWorkstationId` verrà inserita nella mappa solo la postazione con l'id indicato e la stanza che la contiene.
- **refreshAll(filterWorkstationId: number): void**
  Richiede al server le stanze e le postazioni e le organizza, tramite `populateRoomsMap`, in una mappa. Il parametro `filterWorkstationId` viene passato direttamente a `populateRoomsMap`

- **deleteWorkstation(item: number): void**
  Richiede al server l'eliminazione della postazione con l'id specificato
- **deleteRoom(roomId: number): void**
  Richiede al server l'eliminazione della stanza con l'id specificato
- **openAddWorkstation(idroom: number): void**
  Rende visibile il form per l'aggiunta di una postazione. Inoltre imposta i parametri necessari all'esecuzione di questa azione.
- **openEditWorkstation(workstation: WorkstationData): void**
  Rende visibile il form per la modifica di una postazione. Inoltre imposta i parametri necessari all'esecuzione di questa azione.
- **openAddRoom(): void**
  Rende visibile il form per l'aggiunta di una stanza. Inoltre imposta i parametri necessari all'esecuzione di questa azione.
- **openEditRoom(room: RoomData): void**
  Rende visibile il form per la modifica di una stanza. Inoltre imposta i parametri necessari all'esecuzione di questa azione.
- **closeAddEditWorkstation(): void**
  Provoca l'aggiornamento dei dati contenuti nella pagina.
- **closeAddEditRoom(): void**
  Provoca l'aggiornamento dei dati contenuti nella pagina.
- **searchWorkstation(): void**
  Richiede tutti i dati sulle postazioni e sulle stanze al server specificando il filtro indicato dall'utente.
- **resetFilter(): void**
  Richiede tutti i dati sulle postazioni e sulle stanze al server senza alcun filtro.

AddEditRoomComponent

Classe che si occupa dell'aggiunta e modifica di stanze. I primi 5 attributi sono sempre sincronizzati con il form che viene presentato all'utente. Tramite essi viene definita la stanza da aggiungere o i nuovi dati della stanza da modificare.
Questa classe contiene i seguenti attributi:

- **passedRoom: RoomData**
  L'oggetto che viene passato da `RoomComponent` e che indica la stanza da modificare.
- **action: string**
  L'attributo che viene passato da `RoomComponent` e che indica il tipo di azione da eseguire: aggiunta o modifica della stanza;

Questa classe presenta i seguenti metodi:

- **ngOnChanges(): void**
  Metodo chiamato ogni volta che una proprietà data-bound viene modificata. In questo caso serve per aggiornare i parametri locali in base ai valori di `passedRoom`.
- **getRoomFromLocalValues(): RoomData**
  Restituisce un oggetto `RoomData` ottenuto dai primi 5 attributi.
- **takeAction(): void**
  Chiama il metodo `addRoom()` o `editRoom()` a seconda del valore dell'attributo action.
- **addRoom(): void**
  Richiede al service di aggiungere la stanza specificata.
- **editRoom(): void**
  Richiede al service di modificare la postazione nel modo specificato.
- **showMessage(message: string): void**
  Metodo che mostra all'utente la stringa passata come parametro.

RoomsService

Servizio che permette di ottenere dal server informazioni sulle stanze. Ogni metodo di questa classe, tranne il costruttore, restituisce un `Observable`. Questo oggetto espone un metodo subscribe con il quale si potrà impostare una funzione di callback che verrà chiamata quando il server fornirà una risposta.
Questa classe contiene i seguenti attributi:

- **APIUrl: string**
  L'url base dell'API REST a cui il servizio fa riferimento.

Questa classe presenta i seguenti metodi:

- **getRoomsList(): Observable<Array<RoomData> >**
  Usato per ottenere una lista delle stanze.
- **addRoom(val: RoomData): Observable<string>**
  Usato per tentare di aggiungere la stanza indicata in `val` e di ricevere l'esito dell'operazione.
- **deleteWorkstation(val: number): Observable<string>**
  Usato per tentare di eliminare la stanza con id `val` e di ricevere l'esito dell'operazione.
- **modifyWorkstation(val: RoomData): Observable<string>**
  Usato per tentare di modificare la stanza con id pari a quello contenuto in `val` assegnandole gli altri valori dello stesso parametro. Restituisce, tramite un `Observable`, una risposta in formato stringa che indica l'esito dell'operazione.

RoomData

Questa classe contiene tutti e soli i valori necessari per il salvataggio di una stanza sul database. È utilizzata per il transito dei dati delle stanze all'interno dell'applicazione.

3.6.3 Visualizzazione stanze e postazioni

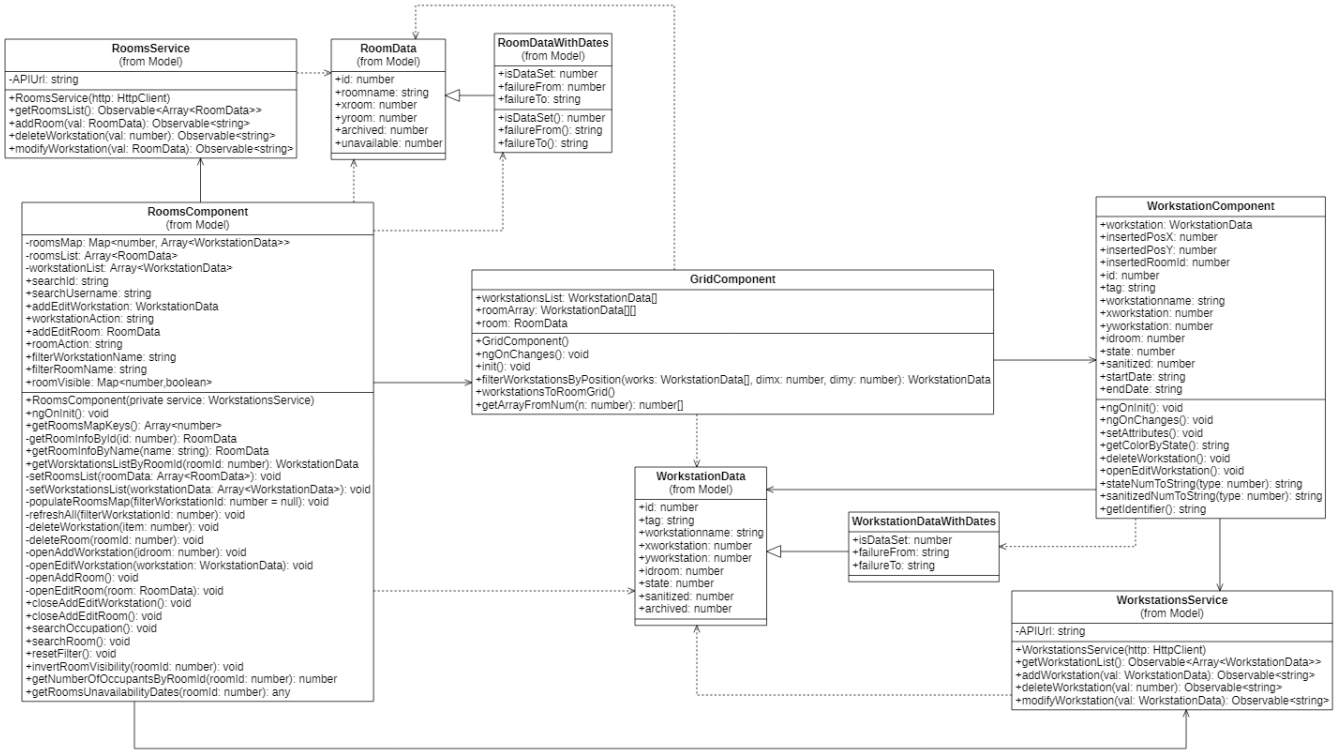

Figura 30: Diagramma delle classi per la visualizzazione delle stanze e delle postazioni

3.6.3.1 Classi

Alla visualizzazione delle stanze e delle postazioni prendono parte le componenti `RoomsComponent`, `GridComponent`, `WorkstationComponent` che gestiscono i dati a tre livelli diversi, dal più alto al più basso. I dati vengono gestiti e trasportati tramite le classi `RoomData`, `RoomDataWithDates`, `WorkstationData`, `WorkstationDataWithDates`. I dati vengono ottenuti dal server tramite i due servizi `RoomsService` e `WorkstationsService`.

Di seguito viene descritto meglio il ruolo delle classi. Vengono esclusi i metodi non strettamente inerenti alla visualizzazione delle stanze e delle postazioni e i metodi il cui funzionamento è facilmente deducibile dal nome.

RoomsComponent

Si occupa di ottenere le stanze e le postazioni dal server, tramite il metodo `refreshAll` e di organizzare le seconde all'interno delle prime tramite il metodo `populateRoomsMap`. Il primo metodo scarica in modo parallelo i dati dal server e li salva negli array `roomsList` e `workstationList`. Il secondo metodo invece prende le postazioni appena raccolte e le suddivide per stanza nella mappa <number, WorsktationData> `roomsMap`. Una volta popolate le stanze, è possibile assegnare ad ogni `GridComponent` le postazioni corrispondenti.

GridComponent

Si occupa di organizzare le postazioni ricevute in una matrice e di visualizzare quest'ultima in un tabella html. Le postazioni sono salvate nell'array `workstationsList` mentre la matrice è `roomArray`. L'organizzazione viene eseguita tramite il metodo `workstationsToRoomGrid`. Una volta eseguita viene assegnata ad ogni WorkstationComponent la rispettiva postazione.

WorkstationComponent

Questo componente si occupa di rappresentare la postazione ricevuta. Questa rappresentazione avviene esponendo nella vista (html) gli attributi della postazione e colorando il componente con un colore consono allo stato e alla situazione di sanificazione. Questo colore viene calcolato tramite il metodo `getColorByState`.

Classi Data

Come suggerito dai nomi, i dati sono rappresentati in due formati: con e senza date. Questa variabilità dipende dallo stato di inaccessibilità della stanza o della postazione. Nel caso in cui essa sia inaccessibile, sono indicate anche le date di inizio e fine del perido in cui tale stato sarà vigente.

Servizi

I due servizi utilizzati forniscono entrambi 4 metodi in formato CRUD (Create, Read, Update, Delete) che rispecchiano le API fornite dal backend.

3.6.4 Gestione credenziali

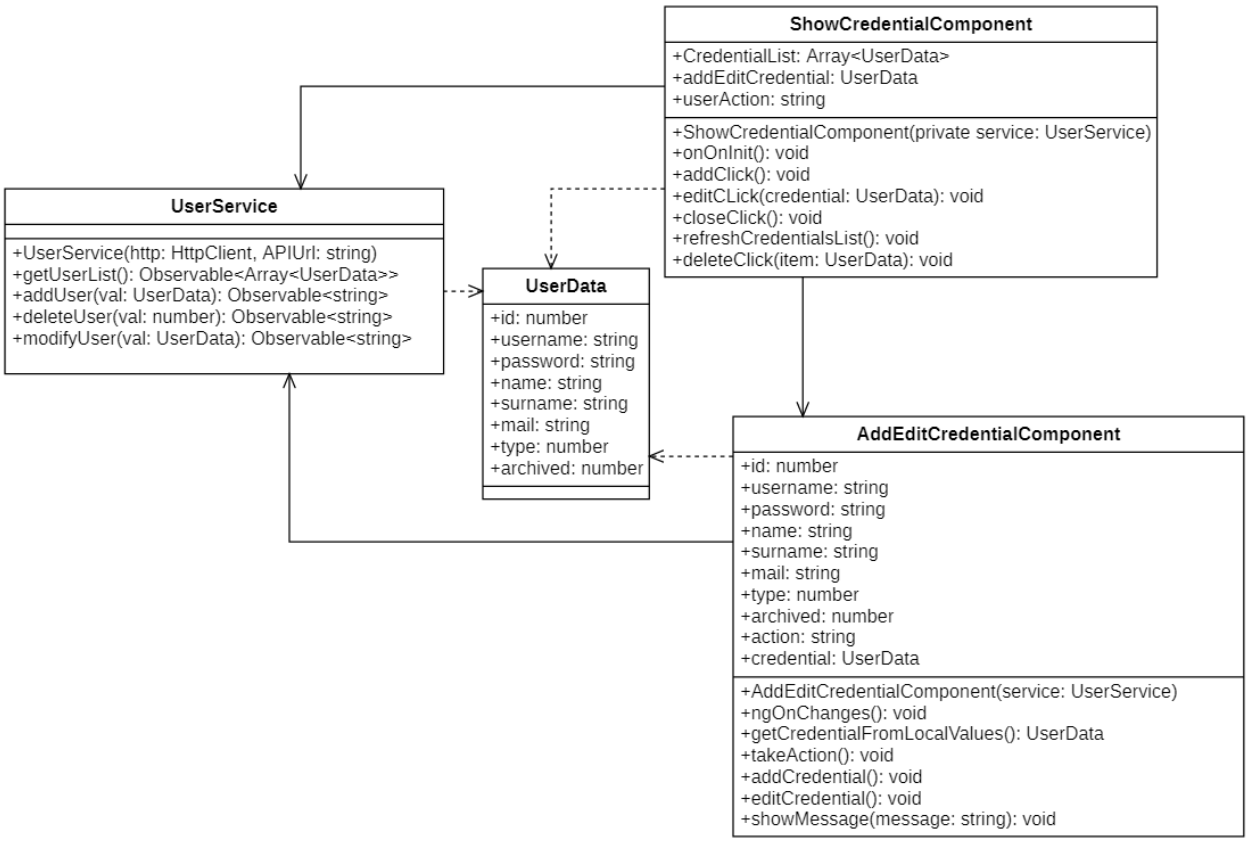

Figura 31: Diagramma delle classi per la gestione delle credenziali

Il diagramma delle classi sovrastante rappresenta le classi coinvolte nella pagina di gestione delle credenziali e le loro relazioni. Il funzionamento di questa pagina è simile a quello della pagina di gestione delle stanze e delle postazioni. Qui la classe `ShowCredentialComponent` si occupa di ottenere la lista delle credenziali dal servizio `UserService` tramite oggetti `Observable`. Essa delega alla classe `AddEditCredentialComponent` l'aggiunta e modifica di credenziali. Il passaggio delle informazioni riguardanti le credenziali viene sempre attuato tramite oggetti della classe `UserData`.

3.6.4.1 Classi

ShowCredentialComponent

Classe che si occupa della visualizzazione di una lista delle credenziali ottenute tramite il service `UserService`.
Questa classe contiene i seguenti attributi:

- **CredentialList: Array<UserData>**
  Array nel quale vengono salvate le credenziali ottenute dal service.

- **addEditCredential: UserData**
  Variabile utilizzata per il passaggio delle informazione da questa classe a `AddEditCredentialComponent`. Il valore di questa variabile viene assegnato alla variabile `credential` di quest'ultima classe.
- **userAction: string**
  Variabile utilizzata per la selezione del tipo di azione (aggiunta o modifica) da eseguire. Il valore di questa variabile viene assegnato alla variabile `action` di `AddEditCredentialComponent`.

Questa classe presenta i seguenti metodi:

- **ngOnInit(): void**
  Metodo chiamato all'inizializzazione della pagina. Provoca l'aggiornamento della lista di credenziali.
- **addClick(): void**
  Metodo chiamato nel momento in cui l'utente apre il form di creazione di una credenziale. Imposta `userAction` e `addEditCredential` in modo che possano essere usate per aggiungere una credenziale al sistema.
- **editClick(credential: UserData): void**
  Metodo chiamato nel momento in cui l'utente apre il form di modifica di una credenziale. Imposta `userAction` e `addEditCredential` in modo che possano essere usate per modificare la credenziale.
- **closeClick(): void**
  Metodo chiamato nel momento in cui l'utente chiude il form di creazione di una credenziale. Provoca l'aggiornamento della lista di credenziali.
- **refreshCredentialsList(): void**
  Provoca l'aggiornamento della lista di credenziali tramite `UserService`.
- **deleteClick(item: UserData): void**
  Metodo chiamato nel momento in cui l'utente preme sul pulsante per l'eliminazione di una credenziale. Provoca l'eliminazione tramite `UserService`.

AddEditCredentialComponent

Classe che si occupa della creazione e modifica di credenziali da parte dell'utente. I primi 8 attributi sono modificabili dall'utente tramite un form e definiscono i valori della credenziale che si sta creando o modificando.
Questa classe contiene i seguenti attributi:

- **action: string**
  Variabile che determina il tipo di azione da eseguire (aggiunta o modifica della credenziale). Viene impostata dalla classe `ShowCredentialComponent`.
- **credential: UserData**
  Variabile attraverso la quale questa classe ottiene i valori iniziali della credenziale che dovrà essere modificata o aggiunta. Viene impostata dalla classe `ShowCredentialComponent`.

Questa classe presenta i seguenti metodi:

- **ngOnChanges(): void**
  Metodo chiamato ogni volta che uno degli attributi data-bound, in questo caso `action` e `credential` viene modificato. Viene utilizzato per aggiornare i primi 8 attributi in base al valore dell'ultimo, che viene impostato da `ShowCredentialComponent`.
- **getCredentialFromLocalValues(): UserData**
  Restituisce un oggetto costruito a partire dai primi 8 attributi della classe.
- **takeAction(): void**
  Avvia l'evento di aggiunta o modifica della credenziale. La scelta tra le due possibilità è determinata dall'attributo action.
- **addCredential(): void**
  Aggiunge la credenziale determinata dai primi 8 valori al server.
- **editCredential(): void**
  Modifica la credenziale presente sul server con id pari all'attributo `id` impostandone gli altri valori con quelli degli altri attributi.
- **showMessage(message: string): void**
  Metodo che mostra all'utente la stringa passata come parametro.

UserService

Servizio che permette di ottenere dal server le credenziali salvate. Ogni metodo di questa classe, tranne il costruttore, restituisce un `Observable`. Questo oggetto espone un metodo subscribe con il quale si potrà impostare una funzione di callback che verrà chiamata quando il server fornirà una risposta.
Questa classe contiene i seguenti attributi:

- **APIUrl: string** L'url base dell'API REST a cui il servizio fa riferimento.

Questa classe presenta i seguenti metodi:

- **getUserList(): Observable<Array<UserData> >**
  Fornisce un'array contenente tutte le credenziali presenti sul server
- **addUser(val: UserData): Observable<string>**
  Permette di aggiungere una credenziale al server. Restituisce l'esito dell'operazione sotto forma di stringa.
- **deleteUser(val: number): Observable<string>**
  Permette di rimuovere una credenziale dal server, specificandone l'id. Restituisce l'esito dell'operazione sotto forma di stringa.
- **modifyUser(val: UserData): Observable<string>**
  Permette di modificare una credenziale presente sul server. La credenziale da modificare è specificata dall'attributo `id` del parametro `val`. I nuovi valori della credenziale sono specificati dagli altri attributi del parametro. Restituisce l'esito dell'operazione sotto forma di stringa.

UserData

Questa classe contiene tutti e soli i valori necessari per il salvataggio di un utente sul database. È utilizzata per il transito dei dati degli utenti all'interno dell'applicazione.

3.6.5 Visualizzazione report

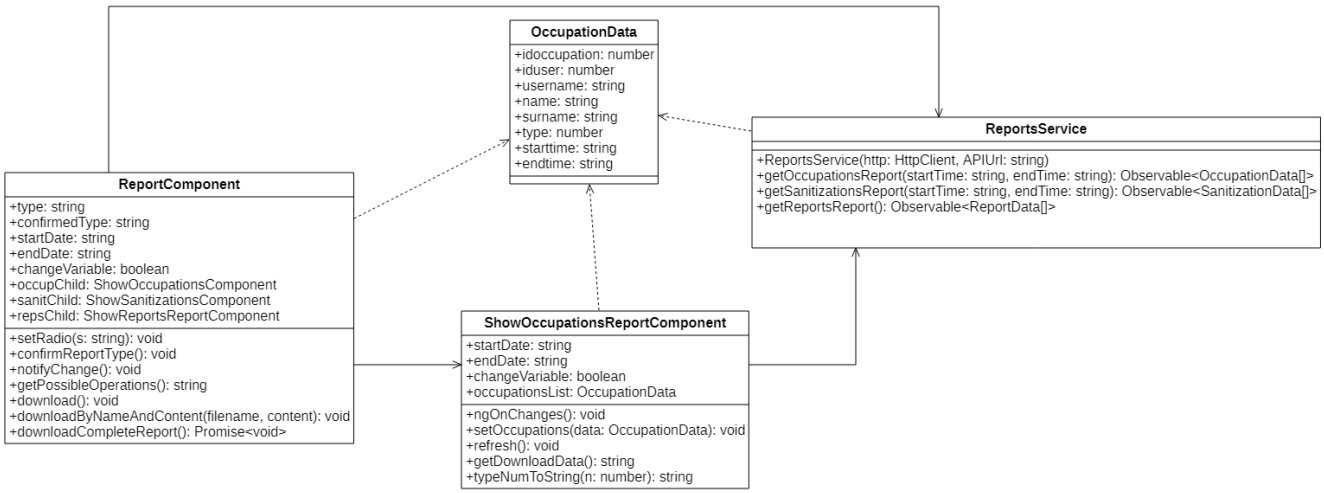

Figura 32: Diagramma delle classi per la visualizzazione dei report

3.6.5.1 Classi

Le classi prese in considerazione nel diagramma servono per la visualizzazione del report delle occupazioni.

ReportComponent

Fornisce un'interfaccia per:

- selezione tipo di report da visaulizzazione o scaricare;
- filtraggio report per periodo;
- scaricamento report.

ShowOccupationsReportComponent

Questo componente si occupa di ottenere una lista delle occupazioni tramite il metodo `refresh` e di mostrarle tramite la vista. Le occupazioni vengono filtrate in base al periodo definito dai valori di `startDate` e `endDate`. Inoltre questo componente fornisce, tramite il metodo `getDownloadData`, la lista delle occupazioni in formato csv. In questo modo `ReportComponent` può ottenere i dati da far scaricare all'utente.

OccupationData

Rappresenta una occupazione singola.

ReportsService

Fornisce tre metodi per ottenere i tre tipi di report possibili. I primi due metodi accettano in input due date che definiscono il periodo all'interno del quale si cercano le informazioni.

L'utilizzo dei componenti `ShowSanitizationsReportComponent` e `ShowReportsReportComponent` è quasi identico a quanto appena descritto. Pertanto esso non viene illustrato esplicitamente

3.7 Diagrammi di sequenza

3.7.1 Login

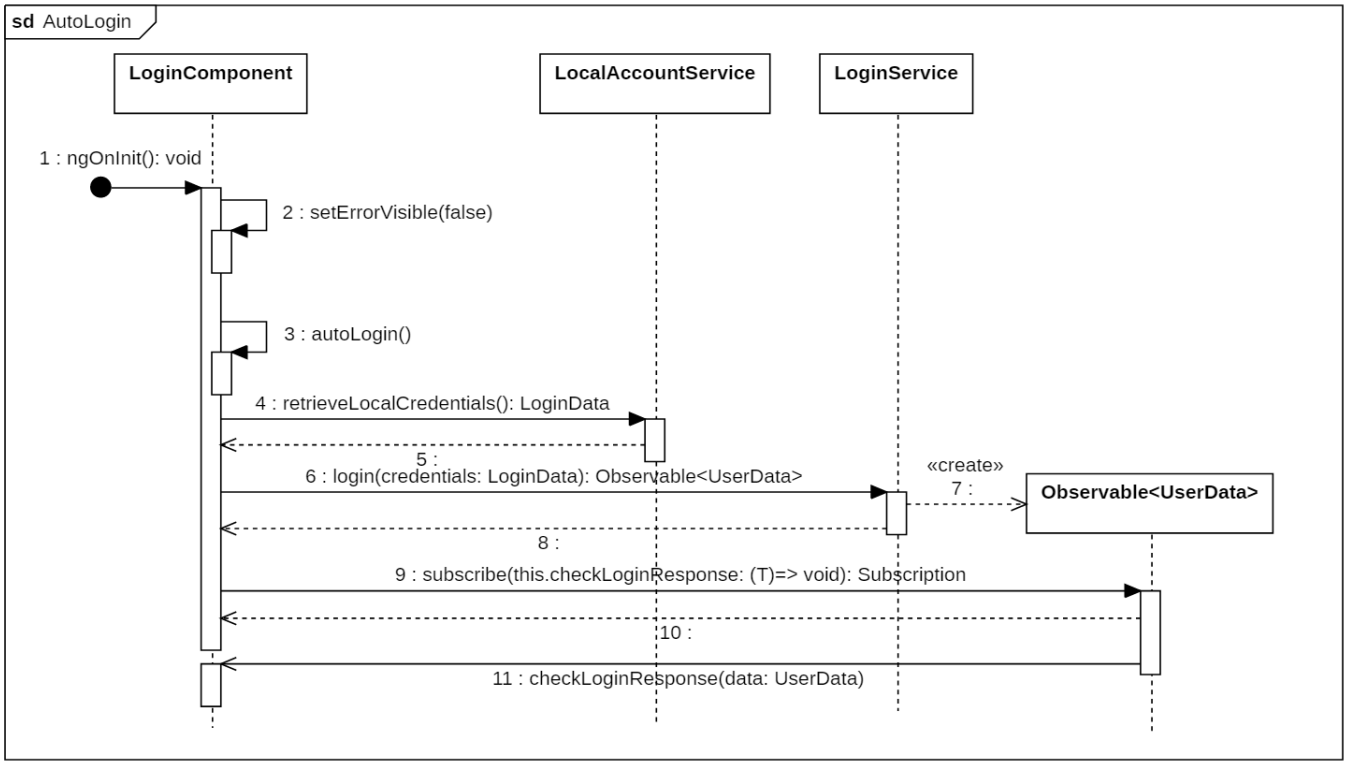

Figura 33: Diagramma di sequenza per l'evento di login automatico

Il diagramma sovrastante rappresenta l'evento di login automatico. L'evento è provocato dall'inizializzazione della pagina (`ngOnInit`). La sequenza consiste nel richiedere le credenziali salvate nel browser tramite il `LocalAccountService` (`retrieveLocalCredentials`) e nel tentare un'autenticazione al server con esse tramite il metodo `login` del `LoginService`. In caso di esito positivo viene resituito da quest'ultimo servizio un oggetto `UserData` con i dati dell'utente che si è autenticato. La funzione `checkLoginResponse` gestisce la risposta del server provocando il reindirizzamento alla home page, in caso di successo, o la visualizzazione del form per l'inserimento delle credenziali.

3.7.2 Aggiunta e modifica stanze

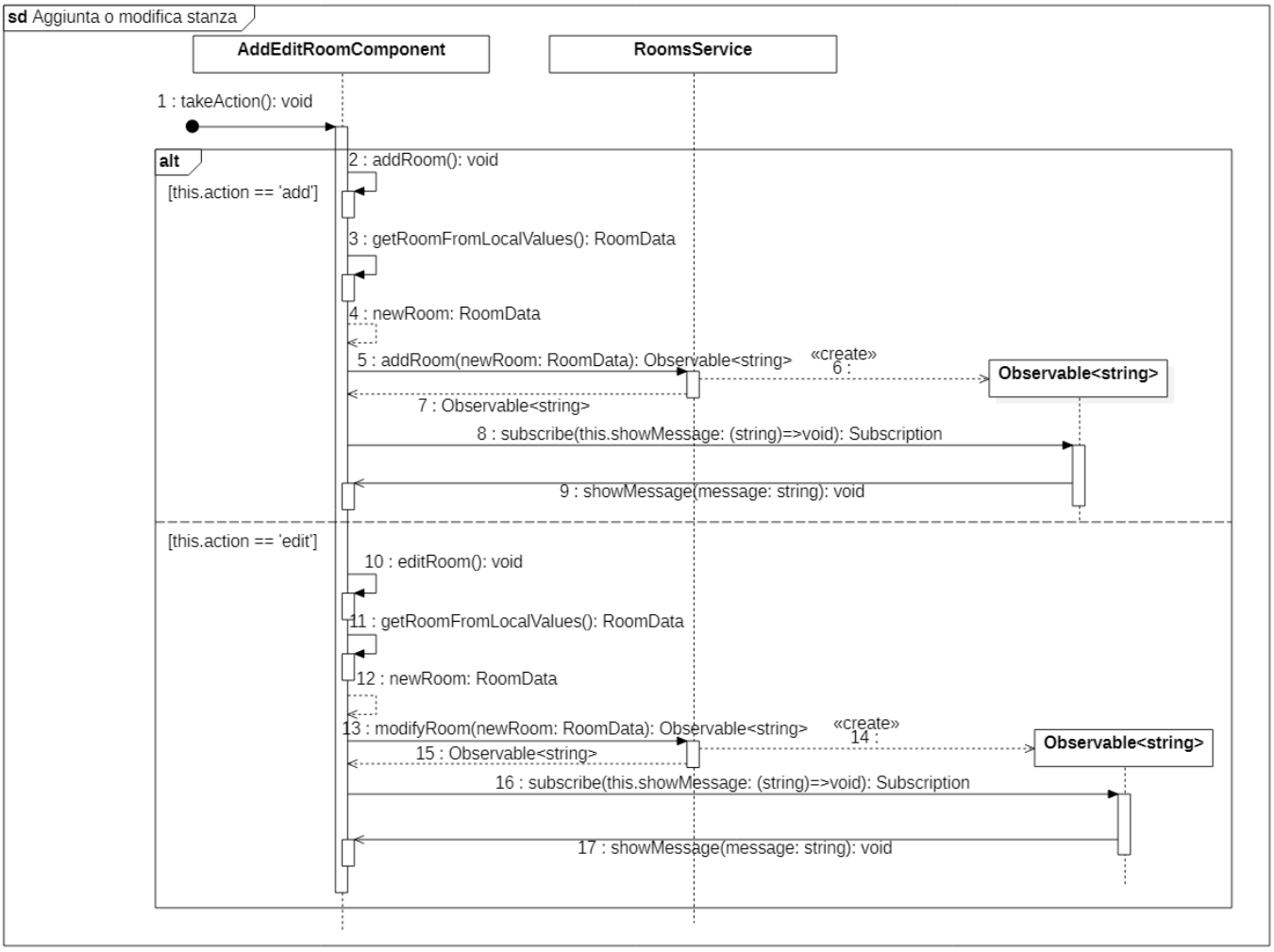

Figura 34: Diagramma di sequenza per l'aggiunta e la modifica di una stanza

Il diagramma sovrastante rappresenta l'evento di aggiunta o modifica di una stanza. L'azione è provocata dalla chiamata del metodo `takeAction` successiva alla pressione di un pulsante da parte dell'utente. A questo punto la scelta dell'una o dell'altra via è determinata dal parametro `action`, che ha assunto valore 'add' o 'edit' a seconda del pulsante premuto. In entrambi i casi l'azione è eseguita inviando i dati da inserire, formattati in un `RoomData`, al servizio. Il servizio restituisce un `Observable` che permette, tramite il metodo `subscribe`, di eseguire la chiamata al server e di impostare una funzione di callback per gestire i dati di ritorno.

3.7.3 Visualizzazione stanze e postazioni

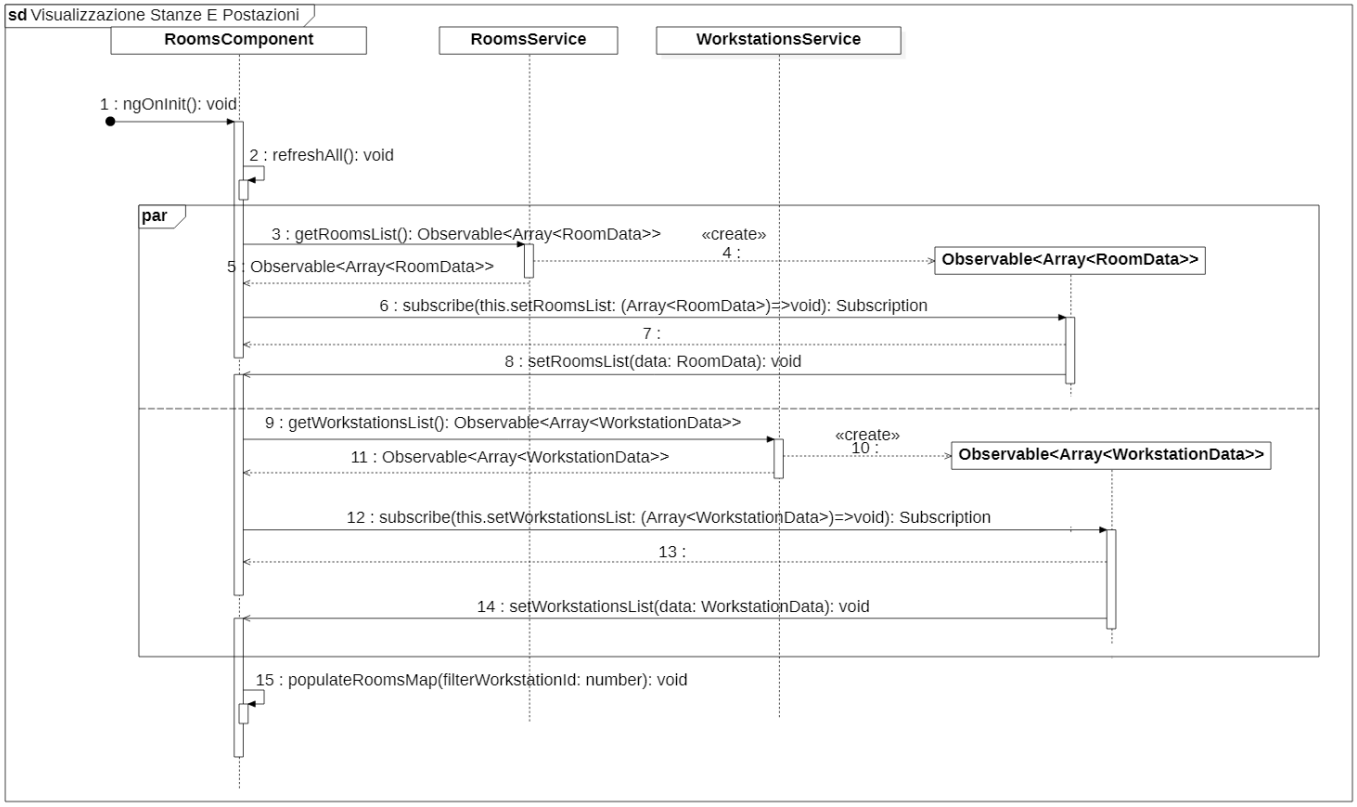

Figura 35: Diagramma di sequenza per la visualizzazione delle stanze e delle postazioni

L'evento rappresentato nel diagramma sovrastante è scatenato dalla chiamata di `ngOnInit`. Come conseguenza viene chiamato il metodo `refreshAll` che fa eseguire parallelamente due richieste di dati al server, la prima per ottenere le stanze e la seconda per ottenere le postazioni. In entrambe le esecuzioni il componente chiama il metodo apposito del servizio e riceve come risposta un `Observable`. Poi, chiamando il metodo `subscribe` di questo oggetto, il componente invia la chiamata al server e imposta la funzione di callback che ne gestirà la risposta. In entrambi i casi i dati di ritorno vengono assegnati a degli attributi del componente. Per le stanze il metodo `setRoomsList` assegna l'array di ritorno all'array locale `roomsList`. Per le stanze, invece, il metodo `setWorkstationsList` assegna l'array di ritorno all'array locale `workstationList`. Quando entrambe le risposte sono arrivate l'esecuzione procede col metodo `populateRoomsMap`, che organizza le stanze e le postazioni in una mappa. Dal diagramma sono esclusi gli eventi successivi al popolamento delle stanze. In sostanza ai diversi GridComponent, rappresentanti le singole stanze, vengono assegnate le rispettive postazioni. All'interno del componente questi oggetti verrano ulteriormente organizzati in una matrice rappresentante la griglia. Per ogni elemento della matrice viene successivamente creato un WorkstationComponent e gli viene assegnata la rispettiva postazione, se presente.

3.7.4 Gestione credenziali

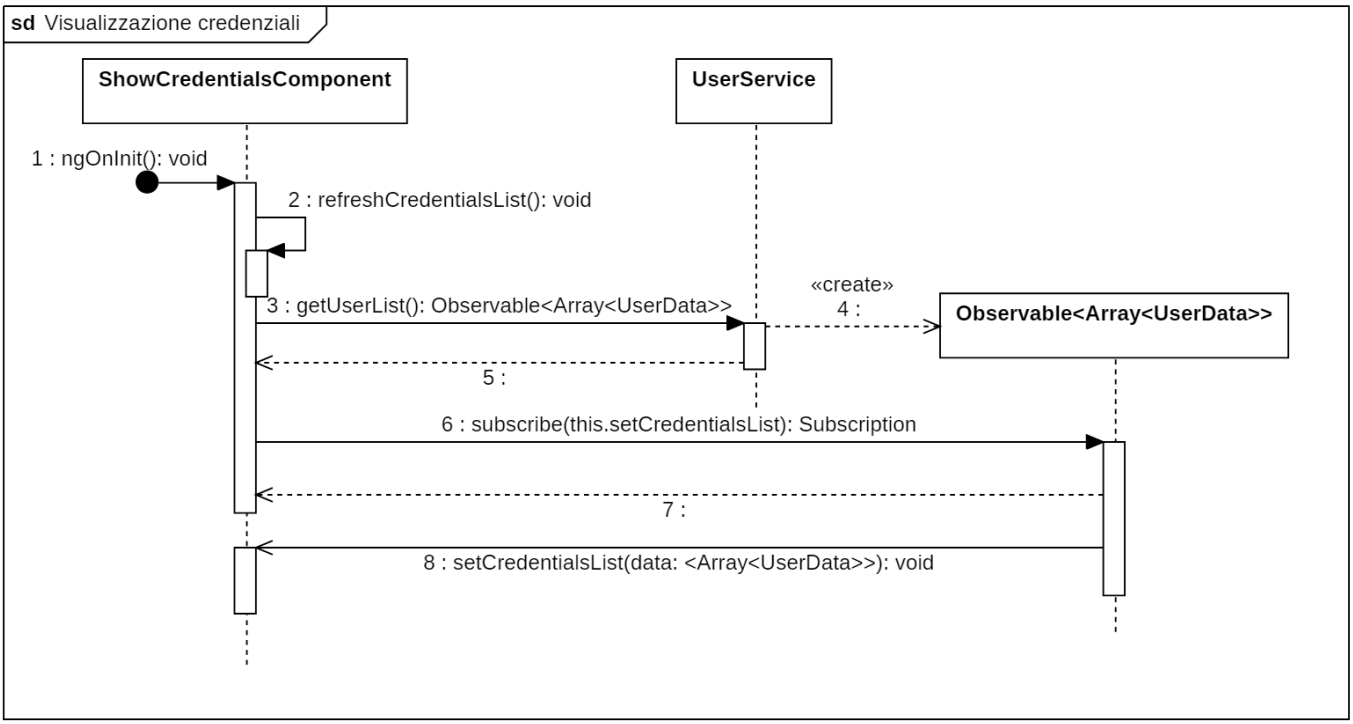

Figura 36: Diagramma di sequenza per la visualizzazione delle credenziali

Il diagramma sovrastante rappresenta l'evento di visualizzazione delle credenziali presenti sul server. L'evento viene scatenato dall'inizializzazione della pagina (`ngOnInit`). Successivamente il componente `ShowCredentialsComponent` chiama il metodo `getUserList` del servizio `UserService` e ottiene un `Observable`. Chiamando il metodo subscribe di quest'ultimo, il componente invia la chiamata al server e, essendo la risposta asincrona, imposta una funzione che gestirà i valori di ritorno. La funzione impostata in questo caso è `setCredentialsList` che, una volta chiamata, assegna all'attributo `CredentialList` i valori ottenuti.

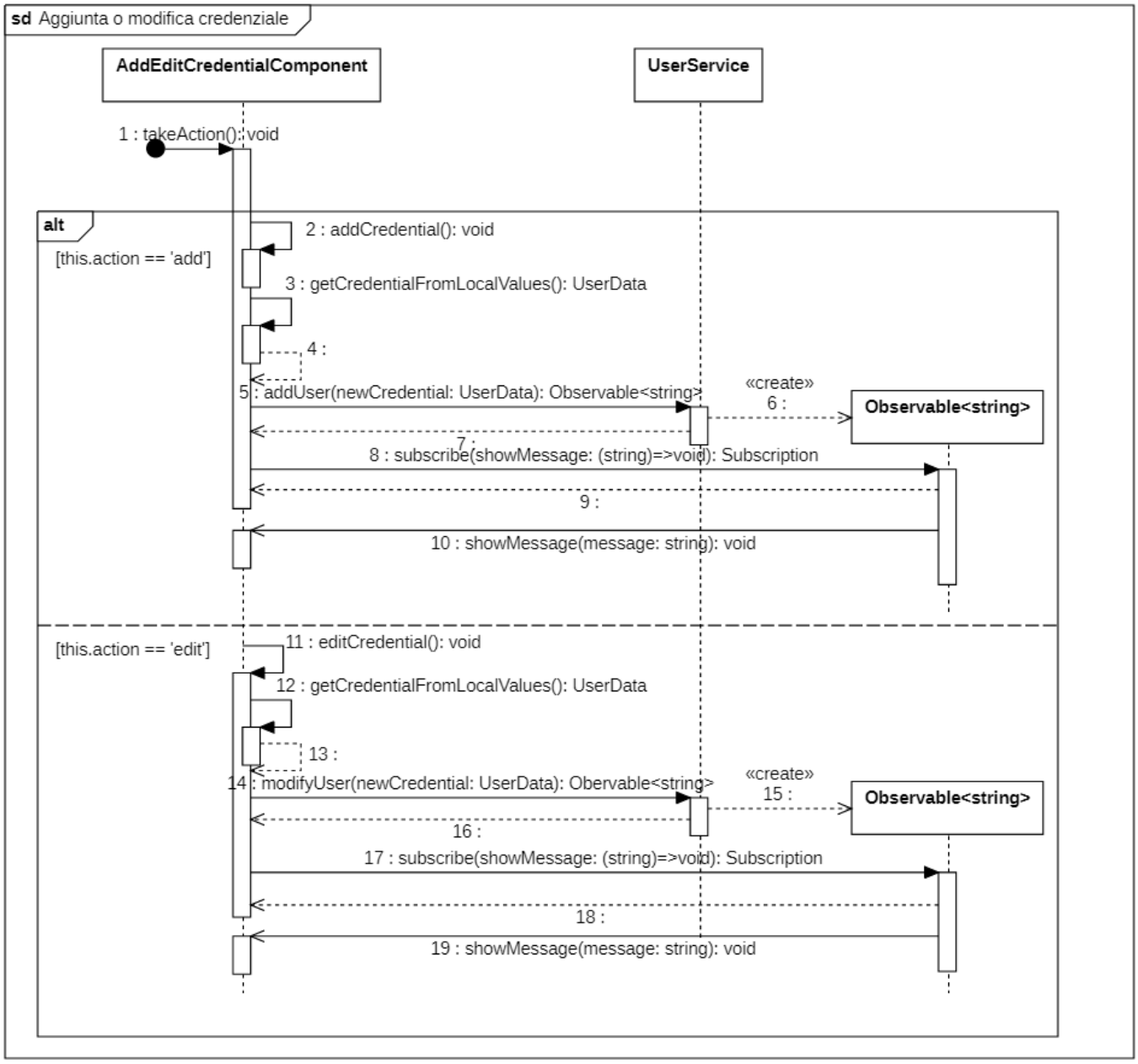

Figura 37: Diagramma di sequenza per l'aggiunta e la modifica delle credenziali

Il diagramma sovrastante rappresenta l'azione di aggiunta o modifica di una credenziale da parte di un'amministratore. L'evento è provocato dalla chiamata a `takeAction`, eseguita dall'utente tramite la pressione di un pulsante. Successivamente viene eseguita l'aggiunta (`addCredential`) o la modifica (`editCredential`) della credenziale definita dai valori inseriti precedentemente dall'utente. In entrambi i casi il componente invia i dati all'`UserService` e riceve un Observable, col quale può, tramite il metodo `subscribe`, inviare la chiamata al server e impostare una funzione di callback per gestirne la risposta.

3.7.5 Visualizzazione report

Non sono stati aggiunti nuovi diagrammi di sequenza in quanto il flusso di esecuzione di questo caso d'uso è lo stesso di quanto descritto in §3.7.3.

4 BackEnd

4.1 Introduzione

Questa parte del prodotto è orientata all'uso da parte di tutto il software che dipenda dalle REST API di BlockCovid. Nel sistema implementato da DPCM2077, dipendono dalle REST API l'applicazione mobile per gli utenti e la web-app per gli amministratori.

4.1.1 Scopo

Il backend permette in BlockCovid, all'app utenti e alla web-app, di fornire ad utenti e amministratori gli strumenti necessari per il funzionamento. Essendo il backend sviluppato in maniera separata ed indipendente dall'applicazione mobile e la web-app, la modalità in cui è implementato non è rilevante per gli sviluppatori *bc19-android* e *bc19-webapp*.

Le funzionalità offerte dal backend sono quelle indicate nella sezione **§7 REST API** del Manuale. Per avere ulteriori informazioni si invita quindi a visitare la sezione indicata.

4.2 Requisiti e installazione

Per poter sviluppare ed aggiungere funzionalità al backend del sistema BlockCovid, sono necessari gli strumenti indicati in questa sezione.

4.2.1 Prerequisiti hardware e software

È consigliato l'utilizzo di un server con risorse sufficienti per il funzionamento del backend. Questo dipende molto dalla mole di utenti e, per non incorrere in problemi in fase di sviluppo, è consigliato avere risorse hardware equivalenti ad un processore quad-core e 8 GB di RAM.

Il sistema operativo di riferimento per lo sviluppo è Ubuntu 18.04 LTS e Ubuntu 20.04 LTS.

4.2.2 Ottenimento codice sorgente

Per scaricare il codice sorgente è necessario **Git**. Se non si dispone di Git è possibile scaricarlo seguendo quanto indicato nella sezione **§4.2.4**. Per scaricare il progetto, invocare il seguente comando da terminale: *git clone https://github.com/DPCMGroup/bc19-api.git*.

4.2.3 Linguaggi utilizzati

4.2.3.1 Python

Il backend è stato sviluppato utilizzando il linguaggio di programmazione PYTHON$_G$, in particolare è stata utilizzata la versione python 3.6 per lo sviluppo del progetto.

Installazione di python

Per l'installazione di python è sufficiente scaricare ed installare il pacchetto, rispettivo al proprio sistema operativo, dal sito ufficiale.

Installazione dependency

Il backend è strutturato secondo quanto messo a disposizione dal framework Django e inoltre utilizza molte altre librerie. I pacchetti utilizzati sono definiti nel file **requirements.txt** e sono necessari per il corretto funzionamento del servizio.
Per installare i pacchetti è necessario posizionarsi con il terminale allo stesso livello del file requirements.txt. Successivamente è sufficiente eseguire il seguente comando: *pip install -r requirements.txt*

4.2.3.2 YAML

Il backend usufruisce dei WORKFLOW$_G$ messi a disposizione da GitHub, chiamati GITHUB ACTIONS$_G$. Il workflow è definito in un file yml, in cui sono specificate le azioni da eseguire per compilare o eseguire gli unit test.
YAML viene utilizzato anche per la definizione del docker-compose su cui verrà eseguito Django.

4.2.4 Source Code Management

Per poter effettuare il versionamento del codice sorgente è richiesto l'utilizzo di **Git**. Se non si dispone di Git è possibile scaricarlo ed installarlo seguendo le istruzioni del sito ufficiale.

4.2.5 Database

Il backend per il suo corretto funzionamento ha bisogno di interfacciarsi ad un database. Per la sua configurazione fare riferimento alla sezione **§5**. Nel caso in cui si volesse cambiare il database a cui Django si interfaccia, è necessario cambiare le impostazioni del file *settings.py* nella sezione *DATABASE*. È consigliato seguire le istruzioni del sito ufficiale di Django.

Django è configurato per l'utilizzo di un database MariaDb e necessita del file my.cnf dove all'interno sono definite le credenziali e l'endpoint di accesso. Il file deve essere allo stesso livello di *manage.py*.

4.2.6 Docker container

DOCKER$_G$ viene utilizzato per l'esecuzione del servizio in produzione.

4.2.6.1 Installazione di Docker su Windows

È possibile installare Docker su Windows visitando il suo sito ufficiale, alla seguente pagina. La guida all'installazione e al primo utilizzo è presente nello stesso link in cui si scarica l'eseguibile per l'installazione.

4.2.6.2 Installazione di Docker su MacOS

L'installazione per MacOS è identica a quella per Windows, ma la pagina a cui scaricarlo si trova a questo link.

4.2.6.3 Installazione Docker Linux

È possibile installare Docker su Ubuntu seguendo le guide presenti sul sito ufficiale, disponibili a questo indirizzo.

4.2.7 Esecuzione

Attualmente sono presenti due modi per eseguire il backend:

- Tramite linea di comando;
- Tramite l'utilizzo di docker (consigliato).

4.2.7.1 Linea di comando

Per eseguire il backend tramite linea di comando, è necessario posizionarsi allo stesso livello del file *manage.py* ed eseguire il comando *python manage.py runserver 8000*.
Questo rende disponibile il servizio sulla porta 8000 dell'host.

4.2.7.2 Docker

Per eseguire il backend tramite docker, è necessario posizionarsi allo stesso livello del file *docker-compose.yml* ed eseguire il comando *docker-compose up*. Docker si occuperà di scaricare tutti i pacchetti necessari per l'esecuzione e renderà disponibile il servizio sulla porta 8000 dell'host.

4.3 Architettura

Nel backend è stato utilizzato il design pattern che utilizza Django, che si ispira al MODEL-VIEW-CONTROLLER (MVC)$_G$, e si chiama Model-Template-View. In questo caso il *Model* rispecchia la gerarchia dei dati presenti nel database, la *View* si occupa della elaborazione di questi dati e il *Template* è, appunto, il template utilizzato come risposta.

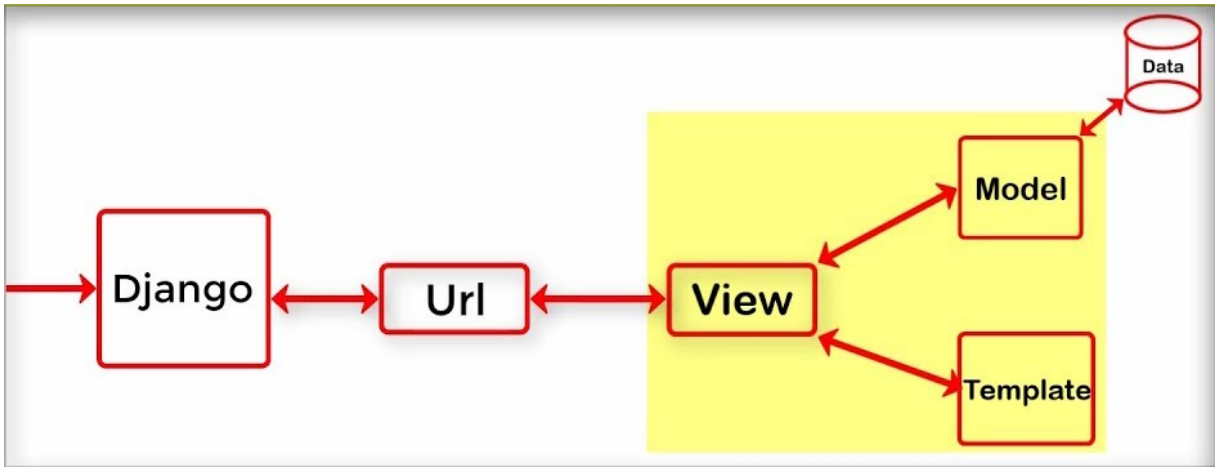

Figura 38: Model-Template-View

4.4 Diagramma delle classi

Viene presentato di seguito il diagramma UML delle classi relativi all'applicazione. Essendo la raffigurazione delle classi simile, viene rappresentata una sezione del diagramma delle classi.

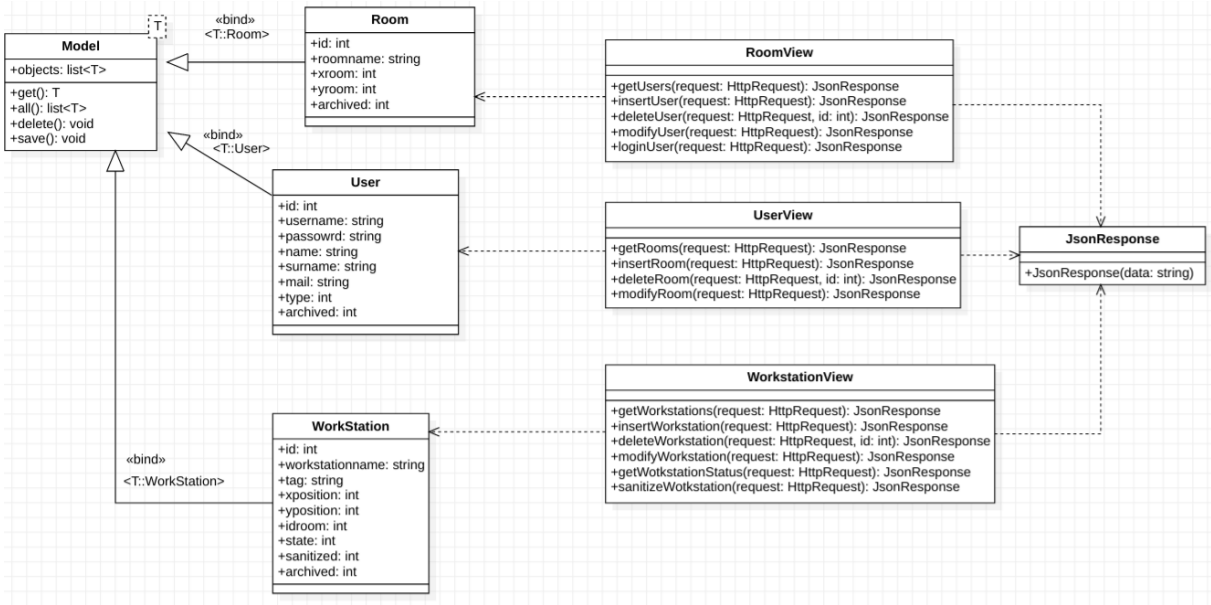

Figura 39: Diagramma delle classi

4.5 Diagramma di sequenza

Viene presentato di seguito il diagramma UML di sequenza relativi all'applicazione. Essendo i diagrammi di sequenza simili, ne viene rappresentata una sezione.

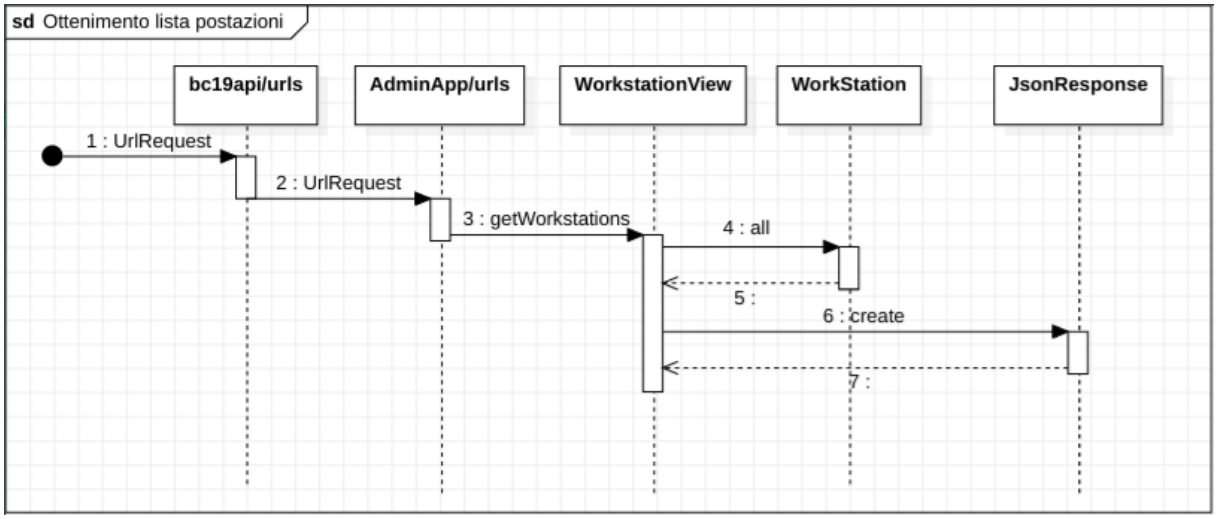

Figura 40: Diagramma di sequenza

5 Database

5.1 Introduzione

Questa sezione tratterà la gestione del database dell'applicazione BlockCovid. Nel sistema implementato da DPCM2077, il database è direttamente collegato al backend.

5.1.1 Scopo

Il database permette al backend di memorizzare i dati che servono per il corretto funzionamento dell'applicazione. Essendo il database implementato nel backend, questa sezione è dedicata a chi gestisce il database e sviluppa il backend.

5.2 Requisiti e installazione

Per poter aggiungere funzionalità al database del sistema BlocKCovid, sono necessari gli strumenti indicati in questa sezione. Per semplicità è consigliato l'utilizzo di docker per l'esecuzione del servizio.

5.2.1 Prerequisiti hardware e software

È consigliato l'utilizzo di un server con risorse sufficienti per il funzionamento del database. Questo dipende molto dalla quantità dei dati e per questo è consigliato avere risorse hardware equivalenti ad un processore quad-core e 8 GB di RAM e sufficiente spazio su disco.

Il database di riferimento è MariaDB 10.5.9.

5.2.2 Ottenimento script

Per scaricare gli script necessari alla creazione e popolamento database è necessario **Git**. Se non si dispone di Git è possibile scaricarlo seguendo quanto indicato nella sezione **§4.2.4**. Per scaricare il progetto, invocare il seguente comando da terminale: *git clone https://github.com/DPCMGroup/bc19-db.git*.

5.2.3 Linguaggi utilizzati

5.2.3.1 SQL

Il linguaggio utilizzato per la creazione e popolamento delle tabelle è stato SQL$_G$.

5.2.4 Docker container

Docker viene utilizzato per l'esecuzione del sevizio in produzione.

5.2.4.1 Installazione di Docker su Windows

È possibile installare Docker su Windows visitando il suo sito ufficiale, alla seguente pagina. La guida all'installazione e al primo utilizzo è presente nello stesso link in cui si scarica l'eseguibile per l'installazione.

5.2.4.2 Installazione di Docker su MacOS

L'installazione per MacOS è identica a quella per Windows, ma la pagina a cui scaricarlo si trova a questo link.

5.2.4.3 Installazione Docker Linux

È possibile installare Docker su Ubuntu seguendo le guide presenti sul sito ufficiale, disponibili a questo indirizzo.

5.2.5 Esecuzione

Attualmente sono presenti due modi per eseguire il database:

- Tramite servizio;
- Tramite l'utilizzo di docker (consigliato).

5.2.5.1 Servizio

Per eseguire il database è necessario averlo installato. Se non lo si ha, si può scaricare e seguire la guida presente a questo link.
Ad installazione completata, il database verrà avviato in automatico ad ogni accensione.

5.2.5.2 Docker

Per eseguire il database tramite docker è sufficiente seguire la seguente guida.

5.3 Schema ER

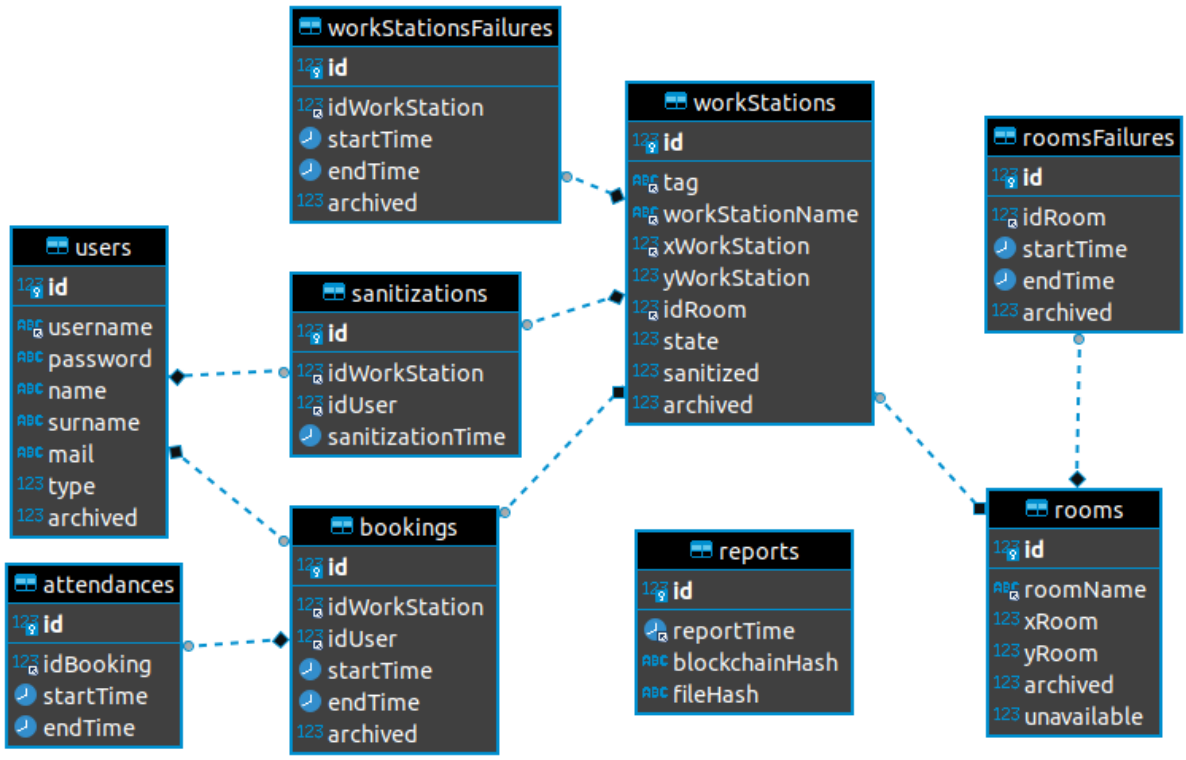

Figura 41: Schema ER

5.3.1 Descrizione

5.3.1.1 Attendances

Entità che raccoglie i dati relativi alle occupazioni di una postazione. È descritta dai seguenti campi:

- id: identificativo dell'occupazione;
- idBooking: identificativo della prenotazione a cui l'occupazione è associata;
- startTime: data e ora di inizio dell'occupazione;
- endTime: data e ora di fine dell'occupazione.

5.3.1.2 Bookings

Entità che raccoglie i dati relativi alle prenotazioni di una postazione. È descritta dai seguenti campi:

- id: identificativo della prenotazione;
- idWorkStation: identificativo della postazione prenotata;

- idUser: identificativo dell'utente che ha effettuato la prenotazione;
- startTime: data e ora di inizio della prenotazione;
- endTime: data e ora di fine della prenotazione;
- archived: segnala se la prenotazione sia stato archiviata: 0 non archiviata, 1 archiviata.

5.3.1.3 Users

Entità che raccoglie i dati relativi agli utenti. È descritta dai seguenti campi:

- id: identificativo dell'utente;
- username: nome dell'utente;
- password: hash della password;
- name: nome proprio dell'utente;
- surname: cognome dell'utente;
- mail: indirizzo e-mail dell'utente;
- type: tipo di utente: 0 amministratore, 1 dipendente, 2 addetto alle pulizie;
- archived: segnala se l'utente sia stato archiviato: 0 non archiviato, 1 archiviato.

5.3.1.4 Sanitizations

Entità che raccoglie i dati relativi alle igienizzazioni. È descritta dai seguenti campi:

- id: identificativo dell'igienizzazione;
- idWorkStation: identificativo della postazione igienizzata;
- idUser: identificativo dell'utente che ha effettuato l'igienizzazione;
- sanitizationTime: data e ora in cui è stata effettuata l'igienizzazione.

5.3.1.5 Reports

Entità che raccoglie le certificazioni generate dal sistema. È descritta dai seguenti campi:

- id: identificativo della certificazione;
- reportTime: data e ora in cui è avvenuta la certificazione;
- blockchainHash: indirizzo della transazione in cui il fileHash è stato salvato;
- fileHash: hash della certificazione.

5.3.1.6 workStationsFailures

Entità che raccoglie i guasti relativi alle postazioni. È descritta dai seguenti campi:

- id: identificativo del guasto;
- idWorkStation: identificativo della postazione soggetta a guasto;
- startTime: data e ora di inizio guasto;
- endTime: data e ora di fine guasto.
- archived: segnala se il guasto sia stato archiviato: 0 non archiviato, 1 archiviato.

5.3.1.7 roomsFailures

Entità che descrive le stanze inaccessibili. È descritta dai seguenti campi:

- id: identificativo dell'inaccessibilità;
- idRoom: identificativo della stanza inaccessibile;
- startTime: data e ora di inizio inaccessibilità;
- endTime: data e ora di fine inaccessibilità.
- archived: segnala se l'inaccessibilità sia stata archiviata: 0 non archiviata, 1 archiviata.

5.3.1.8 WorkStations

Entità che raccoglie i dati relativi alle postazioni. È descritta dai seguenti campi:

- id: identificativo della postazione;
- tag: codice esadecimale descrivente il tag NFC associato alla postazione;
- workStationName: nome della postazione;
- xWorkStation: posizione x (ascissa) in cui si trova la postazione all'interno della stanza;
- xWorkStation: posizione y (ordinata) in cui si trova la postazione all'interno della stanza;
- idRoom: identificativo della stanza in cui si trova la postazione;
- state: descrive lo stato in cui si trova la postazione: 0 libera, 1 occupata, 2 prenotata, 3 guasta;
- sanitized: descrive se la postazione è igienizzata: 0 non igienizzata, 1 igienizzata;
- archived: segnala se la postazione sia stata archiviata: 0 non archiviata, 1 archiviata.

5.3.1.9 Rooms

Entità che raccoglie i dati relativi alle stanze. È descritta dai seguenti campi:

- id: identificativo della stanza;
- roomName: nome della stanza;
- xRoom: dimensione x (lunghezza) della stanza. Si assume che le stanze siano rettangolari;
- yRoom: dimensione y (larghezza) della stanza;
- archived: segnala se la stanza sia stata archiviata: 0 non archiviata, 1 archiviata;
- unavailable: segnala se la stanza sia inaccessibile: 0 accessibile, 1 non accessibile.

6 Blockchain

6.1 Introduzione

Nel nostro sistema la blockchain viene utilizzata per certificare i dati di occupazione e igienizzazione delle postazioni. La tecnologia che abbiamo utilizzato per la nostra blockchain è ETHEREUM$_G$.

6.2 Requisiti e installazione

6.2.1 Ottenimento codice

I file per la configurazione della blockchain si trovano su GitHub all'indirizzo https://github.com/DPCMGroup/bc19-blockchain. Essi si possono scaricare compressi in formato zip direttamente dal sito, oppure tramite il comando
`git clone https://github.com/DPCMGroup/bc19-blockchain`
se si ha già installato il software di versionamento Git.

6.2.2 Tecnologie

6.2.2.1 Geth

Il funzionamento di una rete Ethereum si basa sull'esecuzione di uno o più nodi. Ogni nodo è rappresentato da una Ethereum Virtual Machine (EVM), che manteniene una copia del registro delle transazioni e che si può utilizzare per effettuare transazioni con gli altri nodi. Nel nostro caso la rete è costituita da un solo nodo. Esistono varie implementazioni della EVM e noi abbiamo adottato Geth. Per l'installazione fare riferimento alla pagina https://geth.ethereum.org/docs/install-and-build/installing-geth.
La versione da noi utilizzata è la 1.9.25.

6.2.3 Esecuzione

6.2.3.1 Modifica credenziali

I file di configurazione di Geth forniti presentano un account di gestione della blockchain predefinito. Si consiglia fortemente di modificarne la password prima di usarlo. Per ottenere l'indirizzo dell'account da modificare aprire una shell nella cartella della blockchain ed eseguirvi il comando
`geth account list --datadir .`
All'inizio della riga sarà mostrata la stringa
`Account #0 {...}`
con al posto dei puntini un codice di 40 caratteri. Quel codice è l'indirizzo dell'account.
Per modificare la password dell'account eseguire il comando
`geth account update indirizzo_account --datadir .`
Verrà chiesto di sbloccare l'account con la password "1234". Successivamente si dovrà inserire due

volte la nuova password.
Dopo aver modificato la password scrivere quella nuova nel file `password.sec`, al posto della precedente.

6.2.3.2 Esecuzione blockchain

Linea di comando

Per rendere la blockchain operativa eseguire da riga di comando la seguente stringa:
`geth -networkid 4224 -mine -minerthreads 2 -datadir "." -nodiscover -rpc -rpcport "8545" -port "30303" -rpccorsdomain "*" -allow-insecure-unlock -nat "any" -rpcapi "eth, web3, personal, net, miner" -unlock 0 -password ./password.sec -ipcpath " /.ethereum/geth.ipc"`
La prima inizializzazione della blockchain può richiedere diversi minuti per essere completata. L'inizializzazione è completata quando nella shell compare l'avviso che il primo blocco è stato minato. A questo punto, la blockchain è accessibile all'indirizzo `http://localhost:8545` e può essere utilizzata dal server descritto nella sezione §4. Essa continuerà a minare blocchi, anche se non vengono effettuate transazioni, fino a quando la shell non verrà chiusa o non si eseguirà il comando per fermare il mining. Per effettuare questa azione eseguire la stringa:

```
geth attach url_della_blockchain --exec "miner.stop()"
```

Per far ripartire il mining eseguire:

```
geth attach url_della_blockchain --exec "miner.start()"
```

I dati generati dalla blockchain vengono salvati nella cartella `geth`.
Una documentazione più approfondita per l'utilizzo di Geth si può trovare alla pagina `https://geth.ethereum.org/docs/`.

Docker

Per esguire la blockchain tramite docker, è necessario posizionarsi allo stesso livello del file *docker-compose.yml* ed eseguire il comando *docker-compose up*. Docker si occuperà di scaricare tutti i pacchetti necessari per l'esecuzione, e renderà accessibili le porte 8545 e 30303.
Per poter bloccare il mining, ci si può collegare con geth tramite il comando:

```
geth attach url_della_blockchain --exec "miner.stop()"
```

Per poter riprendere il mining, ci si può collegare con geth tramite il comando:

```
geth attach url_della_blockchain --exec "miner.start(1)"
```

È consigliato avviare il miner con 1 thread (*miner.start(1)*), altrimenti il calcolo del mining può influire molto sulle prestazioni del server.

7 REST API

7.1 Introduzione

Questa parte del documento si riferisce alle API che permettono a BlockCovid di offrire le funzionalità agli utenti e amministratori che usano rispettivamente l'app android e il portale web.

7.1.1 Scopo

Le REST API di BlockCovid permettono all'app utente e alla webapp amministratore di fornire a utenti e amministratori gli strumenti di cui necessitano per l'utilizzo.

7.2 Requisiti e installazione

Per poter utilizzare le REST API è necessario avere un'istanza del backend operativa. Nel caso in cui non si abbia il backend attivo, è possibile consultare la sezione §4.

7.2.1 Formato dei dati

Tutti i dati inviati e ricevuti dalle REST API sono in formato JSON (i.e. application/json).

I tipi di dati e i loro rispettivi valori descritti nelle sezioni che seguono rispecchiano quanto descritto nella sezione **§5.3**, in particolare la sezione **§5.3.1**.

7.3 API

7.3.1 Metodi

7.3.1.1 Occupazione

Richiesta HTTP	Path	Descrizione
POST	/attendences/insert	Inserisce un'occupazione.
POST	/attendences/end	Termina un'occupazione.

7.3.1.2 Prenotazione

Richiesta HTTP	Path	Descrizione
POST	booking/insert	Inserisce una prenotazione.
GET	booking/list	Restituisce la lista di tutte le prenotazioni.
POST	booking/modify	Modifica la prenotazione specificata.
GET	booking/del/{int: id}	Elimina una prenotazione.
POST	booking/gettimetonext	Restituisce quante ore ci sono a disposizione dalla prossima prenotazione.

7.3.1.3 Report

Richiesta HTTP	Path	Descrizione
POST	report/occupations	Ritorna il report delle occupazioni del periodo specificato.
POST	report/sanitizations	Ritorna il report delle sanificazioni del periodo specificato.
GET	report/all	Ritorna la lista dei report presenti nel database.

7.3.1.4 Stanza

Richiesta HTTP	Path	Descrizione
GET	/room/list	Ritorna la lista delle stanze.
GET	/room/del/{int: id}	Cancella la stanza specificata in *id*.
POST	/room/insert	Aggiunge una nuova stanza con i dati specificati.
POST	/room/modify	Modifica una stanza esistente con i dati specificati.
GET	/room/dirty/list	Ritorna la lista delle stanze con almeno una postazione da sanificare.

GET	/room/failure/del/{int: idFail}	Elimina la failure inserita nel database.
GET	/room/failure/delall/{int: idRoom}	Elimina le failure appartenenti alla stanza specificata.
POST	/room/failure/insert	Aggiunge una nuova failure nel database.
POST	/room/failure/modify	Modifica una failure esistente con i dati specificati.
GET	/room/failure/list	Ritorna la lista di tutte le failure.

7.3.1.5 Utente

Richiesta HTTP	Path	Descrizione
GET	/user/list	Ritorna la lista degli utenti.
GET	/user/del/{int: id}	Cancella l'utente specificato in *id*.
POST	/user/insert	Aggiunge un nuovo utente con i dati specificati.
POST	/user/modify	Modifica un utente esistente con i dati specificati.
POST	/user/login	Verifica che i dati inseriti siano corretti per effettuare il login.
GET	/user/bookings/{int: userId}	Ritorna la lista di tutte le prenotazioni eseguite dall'utente specificato in *userId*.

7.3.1.6 Postazione

Richiesta HTTP	Path	Descrizione
GET	/workstation/list	Ritorna la lista delle postazioni.
GET	/workstation/del/{int: id}	Cancella la postazione specificata in *id*.

POST	/workstation/insert	Aggiunge una nuova postazione con i dati specificati.
POST	/workstation/modify	Modifica una postazione esistente con i dati specificati.
POST	/workstation/getInfo	Ritorna le informazioni di una specifica postazione.
POST	/workstation/sanitize	Segna come SANIFICATA$_G$ una specifica postazione.
GET	/workstation/dirty/list	Ritorna la lista delle postazioni da sanificare.
POST	/workstation/bookable/list	Ritorna la lista delle postazioni prenotabili.
POST	/workstation/sanitizeall	Imposta come sanificato tutte le postazioni appartenenti alla stanza specificata.
GET	/workstation/failure/del/{int: idFail}	Elimina la failure inserita nel database.
GET	/workstation/failure/delall/{int: idWorkstation}	Elimina le failure appartenenti alla postazione specificata.
POST	/workstation/failure/insert	Aggiunge una nuova failure nel database.
POST	/workstation/failure/modify	Modifica una failure esistente con i dati specificati.
GET	/workstation/failure/list	Ritorna la lista di tutte le failure.

7.3.2 Descrizione

7.3.2.1 /attendences/insert

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per l'occupazione.

- **idworkstation**: id postazione;
- **iduser**: id utente;
- **time**: ora inizio occupazione in formato "aaaa-MM-gg hh:mm";
- **hour**: numero di ore di occupazione, 0 indica l'occupazione per tutto il tempo a disposizione.

Verrà restituito un JSON contenente le informazioni necessarie riguardante l'occupazione.

```
{
"idattendence": 2,
"idbooking": 20,
"starttime": "2021-06-02 15:00",
"endtime": "2021-06-02 16:59"
}
```

7.3.2.2 /attendences/end

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per il termine dell'occupazione.

- **idattendence**: id occupazione;
- **time**: ora fine occupazione in formato "aaaa-MM-gg hh:mm".

7.3.2.3 /booking/insert

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per l'inserimento di una prenotazione.

- **idworkstation**: id postazione;
- **iduser**: id utente;
- **starttime**: ora inizio prenotazione in formato "aaaa-MM-gg hh:mm";
- **endtime**: ora fine prenotazione in formato "aaaa-MM-gg hh:mm".

7.3.2.4 /booking/list

Chiamata GET. Ritorna la lista delle prenotazioni in formato JSON come segue.

```
[{
"id": 1,
"idworkstation": 1,
"iduser": 2,
"starttime": "2021-06-01 10:30",
"endtime": "2021-06-01 19:00"
}]
```

Rispettivamente i campi corrispondono a:

- **id**: id prenotazione;
- **idworkstation**: nome postazione;
- **iduser**: id utente;
- **starttime**: ora inizio prenotazione in formato "aaaa-MM-gg hh:mm";
- **endtime**: ora fine prenotazione in formato "aaaa-MM-gg hh:mm".

7.3.2.5 /booking/modify

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per la modifica della prenotazione. La prenotazione da modificare è determinata dal parametro *id*:

- **id**: id prenotazione;
- **idworkstation**: nome postazione;
- **iduser**: id utente;
- **starttime**: ora inizio prenotazione in formato "aaaa-MM-gg hh:mm";
- **endtime**: ora fine prenotazione in formato "aaaa-MM-gg hh:mm".

7.3.2.6 /booking/del/{int: id}

Chiamata GET. Elimina la prenotazione identificata da *id*.

7.3.2.7 /booking/gettimetonext

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per il reperimento delle ore disponibili fino alla prenotazione successiva.

- **idworkstation**: id postazione;
- **iduser**: id utente.

Verrà restituita una risposta contenente le ore disponibili.

7.3.2.8 /report/occupations

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per il reperimento del report delle occupazioni.

- **starttime**: ora inizio ricerca in formato "aaaa-MM-gg hh:mm";
- **endtime**: ora fine ricerca in formato "aaaa-MM-gg hh:mm".

Verrà restituito un JSON contenente la lista delle occupazioni e i dati necessari per identificare l'utente.

```
[{
"idoccupation": 1,
"idworkstation": 1,
"iduser": 2,
"username": "averdi",
"name": "alessio",
"surname": "verdi",
"type": 1,
"starttime": "2021-06-01 10:30",
"endtime": "2021-06-01 19:00"
}]
```

7.3.2.9 /report/sanitizations

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per il reperimento del report delle sanificazioni.

- **starttime**: ora inizio ricerca in formato "aaaa-MM-gg hh:mm";
- **endtime**: ora fine ricerca in formato "aaaa-MM-gg hh:mm".

Verrà restituito un JSON contenente la lista delle sanificazioni e i dati necessari per identificare l'utente.

```
[{
"idsanitize": 1,
"idworkstation": 3,
"iduser": 1,
"username": "mrossi",
"name": "mario",
"surname": "rossi",
"type": 0,
"time": "2021-06-02 16:59"
}]
```

7.3.2.10 /report/all

Chiamata GET. Ritorna la lista dei report in formato JSON come segue.

```
[{
"id": 1,
"reporttime": "lab1",
"blockchainhash": 10,
"fileHash": 10
}]
```

Rispettivamente i campi corrispondono a:

- **id**: id report;
- **reporttime**: ora calcolo report inserito;
- **blockchainhash**: hash transazione BLOCKCHAIN$_G$;
- **fileHash**: hash report generato.

7.3.2.11 /room/list

Chiamata GET. Ritorna la lista delle stanze in formato JSON come segue.

```
[{
"id": 1,
"roomname": "lab1",
"xroom": 10,
"yroom": 10,
"archived": 0
}]
```

Rispettivamente i campi corrispondono a:

- **id**: id stanza;
- **roomname**: nome stanza;
- **xroom**: grandezza stanza nell'asse X;
- **yroom**: grandezza stanza nell'asse Y;
- **archived**: indica se la stanza è archiviata.

7.3.2.12 /room/del/{int: id}

Chiamata GET. Cancella una stanza specificandone l'id.

7.3.2.13 /room/insert

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per l'inserimento della nuova stanza:

- **roomname**: nome stanza;
- **xroom**: grandezza stanza nell'asse X;
- **yroom**: grandezza stanza nell'asse Y;
- **archived**: indica se la stanza è archiviata.

7.3.2.14 /room/modify

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per la modifica della stanza. La stanza da modificare è determinata dal parametro *id*:

- **id**: id stanza;
- **roomname**: nome stanza;
- **xroom**: grandezza stanza nell'asse X;
- **yroom**: grandezza stanza nell'asse Y;
- **archived**: indica se la stanza è archiviata.

7.3.2.15 /room/dirty/list

Chiamata GET. Ritorna la lista delle stanze con almeno una postazione non sanificata in formato JSON come segue.

```
[{
"id": 1,
"roomname": "lab1",
"xroom": 10,
"yroom": 10,
"archived": 0
}]
```

Rispettivamente i campi corrispondono a:

- **id**: id stanza;
- **roomname**: nome stanza;

- **xroom**: grandezza stanza nell'asse X;
- **yroom**: grandezza stanza nell'asse Y;
- **archived**: indica se la stanza è archiviata.

7.3.2.16 /room/failure/del/{int: idFail}

Chiamata GET. Cancella un una "roomFailure" specificandone l'id.

7.3.2.17 /room/failure/delall/{int: idRoom}

Chiamata GET. Cancella tutte le "roomFailure" identificate dalla id stanza.

7.3.2.18 /room/failure/insert

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per l'inserimento dell'inaccessibilità della stanza:

- **idroom**: id stanza;
- **starttime**: ora inizio failure in formato "aaaa-MM-gg hh:mm";
- **endtime**: ora fine failure in formato "aaaa-MM-gg hh:mm".

Viene fatto notare che nel caso in cui si voglia specificare una failure a tempo indeterminato, *endtime* viene impostato a *null*.

7.3.2.19 /room/failure/modify

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per la modifica dell'inaccessibilità della stanza:

- **id**: id failure;
- **idroom**: id stanza;
- **starttime**: ora inizio failure in formato "aaaa-MM-gg hh:mm";
- **endtime**: ora fine failure in formato "aaaa-MM-gg hh:mm".

7.3.2.20 /room/failure/list

Chiamata GET. Ritorna la lista delle inaccessibilità delle stanze in formato JSON come segue.

```
[{
"id": 1,
"idroom": 2,
"starttime": "2021-06-01 10:30",
"endtime": "2021-06-01 19:00"
}]
```

Rispettivamente i campi corrispondono a:

- **id**: id failure;
- **idroom**: id stanza;
- **starttime**: ora inizio inaccessibilità in formato "aaaa-MM-gg hh:mm";
- **endtime**: ora fine inaccessibilità in formato "aaaa-MM-gg hh:mm".

7.3.2.21 /user/list

Chiamata GET. Ritorna la lista degli utenti in formato JSON come segue.

```
[{
"id": 1,
"username": "mrossi",
"password": "000",
"name": "mario",
"surname": "rossi",
"mail": "mario.rossi@gmail.com",
"type": 0,
"archived": 0
}]
```

Rispettivamente i campi corrispondono a:

- **id**: id utente;
- **username**: username utente;
- **password**: password utente;
- **name**: nome utente;
- **surname**: cognome utente;
- **mail**: indirizzo email utente;

- **type**: tipo utente;
- **archived**: indica se l'utente è archiviato.

7.3.2.22 /user/del/{int: id}

Chiamata GET. Cancella un utente specificandone l'id.

7.3.2.23 /user/insert

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per l'inserimento del nuovo utente:

- **username**: username utente;
- **password**: password utente;
- **name**: nome utente;
- **surname**: cognome utente;
- **mail**: indirizzo email utente;
- **type**: tipo utente;
- **archived**: indica se l'utente è archiviato.

7.3.2.24 /user/modify

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per la modifica dell'utente. L'utente da modificare è determinato dal parametro *id*:

- **id**: id utente;
- **username**: username utente;
- **password**: password utente;
- **name**: nome utente;
- **surname**: cognome utente;
- **mail**: indirizzo email utente;
- **type**: tipo utente;
- **archived**: indica se l'utente è archiviato.

7.3.2.25 /user/login

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per il login dell'utente:

- **username**: username utente;
- **password**: password utente.

Se il login avviene con successo, viene ritornato l'oggetto *User* in formato JSON come specificato in **§7.3.2.21**

7.3.2.26 user/bookings/{int: userId}

Chiamata GET. Viene richiesta la lista delle prenotazioni dell'utente specificato in *userId*.
Verrà ritornata una lista JSON come segue:

```
[{
"bookId": 1,
"workId": 1,
"workName": "nome",
"roomId": 1,
"roomName": "nomeStanza",
"start": "2021-07-30 10:00",
"end": "2021-07-30 11:00"
}]
```

- **bookId**: id prenotazione;
- **workId**: id postazione prenotata;
- **workName**: nome postazione;
- **roomId**: id stanza;
- **roomName**: nome stanza;
- **start**: ora inizio prenotazione in formato "aaaa-MM-gg hh:mm";
- **end**: ora fine prenotazione in formato "aaaa-MM-gg hh:mm".

7.3.2.27 /workstation/list

Chiamata GET. Ritorna la lista delle postazioni in formato JSON come segue.

```
[
  {
  "id": 1,
  "tag": "00 c0 00 01 8d 91 04",
  "workstationname": "lab1-1x1",
  "xworkstation": 1,
  "yworkstation": 1,
  "idroom": 1,
  "state": 0,
  "sanitized": 1,
  "archived": 0
  }
]
```

Rispettivamente i campi corrispondono a:

- **id**: id postazione;
- **tag**: id tag postazione;
- **workstationname**: nome postazione;
- **xworkstation**: posizione asse x all'interno della stanza;
- **yworkstation**: posizione asse y all'interno della stanza;
- **idroom**: id stanza in cui è presente la postazione;
- **state**: stato attuale della postazione;
- **sanitized**: indica se la postazione è sanificata;
- **archived**: indica se la postazione è archiviata.

7.3.2.28 /workstation/del/{int: id}

Chiamata GET. Cancella una postazione specificandone l'id.

7.3.2.29 /workstation/insert

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per l'inserimento della nuova postazione:

- **tag**: id tag postazione;
- **workstationname**: nome postazione;
- **xworkstation**: posizione asse x all'interno della stanza;

- **yworkstation**: posizione asse y all'interno della stanza;
- **idroom**: id stanza in cui è presente la postazione;
- **state**: stato attuale della postazione;
- **sanitized**: indica se la postazione è sanificata;
- **archived**: indica se la postazione è archiviata.

7.3.2.30 /workstation/modify

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per la modifica della postazione. La postazione da modificare è determinata dal parametro *id*:

- **id**: id postazione;
- **tag**: id tag postazione;
- **workstationname**: nome postazione;
- **xworkstation**: posizione asse x all'interno della stanza;
- **yworkstation**: posizione asse y all'interno della stanza;
- **idroom**: id stanza in cui è presente la postazione;
- **state**: stato attuale della postazione;
- **sanitized**: indica se la postazione è sanificata;
- **archived**: indica se la postazione è archiviata.

7.3.2.31 /workstation/getInfo

Chiamata POST. Nella sezione *body* viene specificato l'idtag della postazione di cui si vogliono avere informazioni.
La risposta è strutturata come segue:

```
{
"workId": 2,
"workName": "lab1-1x2",
"workStatus": 1,
"workSanitized": 1,
"roomName": "lab1",
"bookedToday": 0
}
```

Se la postazione ha delle prenotazioni, allora la risposta avrà dei campi aggiuntivi:

```
{
[...]
"bookedToday": 1
"bookings": [{
  "bookerId": 4,
  "bookerUsername": username,
  "bookerName": name,
  "bookerSurname": surname,
  "from": "2021-07-30 10:00",
  "to": "2021-07-30 11:00"
  }]
}
```

- **workId**: id postazione;
- **workName**: nome postazione;
- **workStatus**: stato attuale della postazione;
- **workSanitized**: indica se la postazione è sanificata;
- **bookedToday**: indica se la postazione ha una prenotazione nella giornata in cui viene scansionata.
- **bookings**: lista prenotazioni presenti per la postazione.
  - **bookerId**: id utente della prenotazione;
  - **bookerUsername**: nome utente di chi ha effettuato la prenotazione;
  - **bookerName**: nome di chi ha effettuato la prenotazione;
  - **bookerSurname**: cognome di chi ha effettuato la prenotazione;
  - **from**: ora inizio prenotazione in formato "aaaa-MM-gg hh:mm";
  - **to**: ora fine prenotazione in formato "aaaa-MM-gg hh:mm";

7.3.2.32 /workstation/sanitize

Chiamata POST. Inserisce una sanificazione per una specifica postazione, indicando chi ha effettuato tale pulizia e a che ora è avvenuta.
Il JSON è strutturato come segue:

- **tag**: id tag postazione;
- **idUser**: id utente di chi ha effettuato la sanificazione;
- **data**: ora sanificazione in formato "aaaa-MM-gg hh:mm".

7.3.2.33 /workstation/dirty/list

Chiamata GET. Ritorna la lista delle postazioni non sanificate in formato JSON come segue.

```
[{
"id": 1,
"tag": "00 c0 00 01 8d 91 04",
"workstationname": "lab1-1x1",
"xworkstation": 1,
"yworkstation": 1,
"idroom": 1,
"state": 0,
"sanitized": 1,
"archived": 0
}]
```

Rispettivamente i campi corrispondono a:

- **id**: id postazione;
- **tag**: id tag postazione;
- **workstationname**: nome postazione;
- **xworkstation**: posizione asse x all'interno della stanza;
- **yworkstation**: posizione asse y all'interno della stanza;
- **idroom**: id stanza in cui è presente la postazione;
- **state**: stato attuale della postazione;
- **sanitized**: indica se la postazione è sanificata;
- **archived**: indica se la postazione è archiviata.

7.3.2.34 /workstation/bookable/list

Chiamata POST. Ritorna le postazioni prenotabili specificato un intervallo di tempo.
Il JSON per la chiamata è strutturato come segue:

- **startTime**: ora inizio intervallo in formato "aaaa-MM-gg hh:mm;
- **endTime**: ora fine intervallo in formato "aaaa-MM-gg hh:mm;
- **idRoom**: id stanza.

7.3.2.35 /workstation/sanitizeall

Chiamata POST. Imposta come sanificate tutte le postazioni appartenenti ad una stanza.
Il JSON per la chiamata è strutturato come segue:

- **time**: ora sanificazione in formato "aaaa-MM-gg hh:mm;
- **iduser**: id utente che ha effettuato la sanificazione;
- **idroom**: id stanza.

7.3.2.36 /workstation/failure/del/{int: idFail}

Chiamata GET. Cancella una "workstationFailure" specificandone l'id.

7.3.2.37 /workstation/failure/delall/{int: idWork}

Chiamata GET. Cancella tutte le "workstationFailure" che appartengono alla postazione identificata da *idWork*.

7.3.2.38 /workstation/failure/insert

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per l'inserimento dell'inaccessibilità della postazione:

- **idworkstation**: id postazione;
- **starttime**: ora inizio failure in formato "aaaa-MM-gg hh:mm";
- **endtime**: ora fine failure in formato "aaaa-MM-gg hh:mm".

Viene fatto notare che nel caso in cui si voglia specificare una failure a tempo indeterminato, *endtime* viene impostato a *null*.

7.3.2.39 /workstation/failure/modify

Chiamata POST. Nella sezione *body* viene specificato il JSON con i dati necessari per l'inserimento dell'inaccessibilità della postazione:

- **id**: id failure;
- **idworkstation**: id postazione;
- **starttime**: ora inizio failure in formato "aaaa-MM-gg hh:mm";
- **endtime**: ora fine failure in formato "aaaa-MM-gg hh:mm".

7.3.2.40 /workstation/failure/list

Chiamata GET. Ritorna la lista delle inaccessibilità delle postazioni in formato JSON come segue.

```
[{
"id": 1,
"idworkstation": 2,
"starttime": "2021-06-01 10:30",
"endtime": "2021-06-01 19:00"
}]
```

Rispettivamente i campi corrispondono a:

- **id**: id failure;
- **idworkstation**: id postazione;
- **starttime**: ora inizio inaccessibilità in formato "aaaa-MM-gg hh:mm";
- **endtime**: ora fine inaccessibilità in formato "aaaa-MM-gg hh:mm".

8 Glossario

A

Android

Sistema operativo per dispositivi mobili.

Angular

Framework per lo sviluppo e il design di applicazioni single-page.

API

Acronimo per Application Programming Interface (API). È un'interfaccia con ruolo di intermediazione fra software diversi.

Applicazione

Programma per calcolatore predisposto per eseguire una categoria di funzioni specifiche.

Architettura software

L'architettura software ci dice come un sistema è organizzato, ossia quali sono i ruoli delle componenti del sistema e quali interazioni esistono fra di esse. Essa definisce, inoltre, le interfacce necessarie all'interazione tra componenti o con l'ambiente. Il tutto non viene fatto in modo estemporaneo, ma seguendo dei paradigmi di composizione precisi.

Asincrono

Dispositivo che opera senza un riferimento temporale di sincronizzazione rispetto a un altro dispositivo.
Protocollo asincrono: modalità di trasmissione dati che non dipende dal compiersi di altri processi.

B

Back-end

Back-end è un termine largamente utilizzato per caratterizzare le interfacce che hanno come destinatario un programma. Una applicazione back-end è un programma con il quale l'utente interagisce

indirettamente, in generale attraverso l'utilizzo di una applicazione front-end.

Blockchain

Una blockchain è una struttura dati condivisa e immutabile. È definita come un registro digitale le cui voci sono raggruppate in blocchi, concatenati in ordine cronologico, e la cui integrità è garantita dall'uso della crittografia.

C

CSS

CSS (Cascading Style Sheets), è un linguaggio usato per definire la formattazione di documenti HTML, XHTML e XML, ad esempio i siti web e le relative pagine.

D

Docker

Il software Docker è una tecnologia di containerizzazione che consente la creazione e l'utilizzo dei container Linux.

E

Ethereum

Blockchain per la creazione e pubblicazione di SMART CONTRACT$_G$.

F

Framework

È una infrastruttura intesa come "struttura o complesso di elementi che costituiscono la base di sostegno o comunque la parte sottostante di altre strutture". Si intende la piattaforma che funge da strato intermedio tra un sistema operativo e il software che lo utilizza.

G

GitHub

GitHub è un servizio di hosting per progetti software.

GitHub Actions

GitHub Actions è uno strumento fornito da GitHub che permette l'automazione di compiti di varia natura.

H

HTML

È un linguaggio di markup per la strutturazione delle pagine web.

I

Interfaccia

Servizi offerti da una entità a un'altra entità.

J

Javascript

JavaScript è un linguaggio di scripting orientato agli oggetti e agli eventi, comunemente utilizzato nella programmazione Web.

K

Kotlin

Kotlin è un linguaggio di programmazione versatile, multi-paradigma e open-source creato da JetBrains.

L

Linux

È una famiglia di sistemi operativi open-source di tipo Unix-like.

M

MacOS

Sistema operativo Unix-like dei personal computer di Apple Inc..

MVC

Acronimo di Model-View-Controller (MVC, talvolta tradotto in italiano con la dicitura modello-vista-controllo). È un "pattern architetturale" soprattutto utilizzato per la programmazione orientata agli oggetti, in grado di separare la rappresentazione interna dei dati da ciò che viene presentato e accettato dall'utente.

N

NFC

Acronimo di Near Field Communication (Comunicazione a Corto Raggio). Indica una tecnologia di trasmissione di dati senza fili a distanze tipicamente inferiori ai 10 cm.

Node.js

Ambiente open-source per l'esecuzione di codice Javascript a run-time, al di fuori dei browser. Ciò permette l'esecuzione di codice Javascript server-side.

P

Postazione

Spazio fisico identificato da un tag RFID univoco dove l'utilizzatore appoggia il cellulare mentre sta svolgendo il suo lavoro. Ciascuna postazione di lavoro è inserita in una stanza dell'organizzazione (laboratorio, ufficio, biblioteca, etc. . .).

Python

Python è un linguaggio di programmazione di più "alto livello" rispetto alla maggior parte degli altri linguaggi, orientato a oggetti, adatto, tra gli altri usi, a sviluppare applicazioni distribuite, scripting, computazione numerica e system testing.

R

Repository

Spazio di archiviazione remoto. È utilizzato per la condivisione di software e documenti.

RFID

Radio-frequency identification, in telecomunicazioni ed elettronica, si intende una tecnologia per l'identificazione e/o memorizzazione automatica di informazioni inerenti a oggetti, animali o persone basata sulla capacità di memorizzazione di dati da parte di particolari etichette elettroniche, chiamate tag, e sulla capacità di queste di rispondere all'interrogazione a distanza da parte di appositi apparati fissi o portatili, chiamati reader.

S

Sanificazione

Viene inteso l'atto di pulizia delle postazioni o stanze.

Server

Calcolatore che svolge funzioni di servizio per tutti i calcolatori collegati oppure programma, generalmente sempre attivo, che esegue determinate funzioni quando queste sono richieste da altri programmi.

SQL

È un linguaggio standardizzato per database basati sul modello relazionale.

T

Tag NFC

Sono dei transponder RFID, ovvero dei minuscoli chip collegati a un'antenna. Il chip ha un codice univoco e una parte di memoria riscrivibile. L'antenna permette al chip di interagire con un lettore NFC, come uno smartphone NFC.

Test

Esperimento variamente espletato allo scopo di saggiare, mediante determinate reazioni, l'entità o la consistenza di un'attitudine o di una capacità individuale.

V

Versionamento

È la gestione di versioni multiple di un insieme di informazioni: gli strumenti software per il controllo versione sono ritenuti molto spesso necessari per la maggior parte dei progetti di sviluppo software o documentali gestiti da un team collaborativo di sviluppo o redazione.

W

Windows

Sistema operativo creato da Microsoft.

Workflow

È l'automazione totale o parziale di un processo aziendale, in cui documenti, informazioni o compiti passano da un partecipante a un altro per svolgere attività, secondo un insieme di regole definite.