



Università degli Studi di Camerino

SCUOLA DI SCIENZE E TECNOLOGIE

Corso di Laurea in Informatica (Classe L-31)

Rete di tracciamento anonimo via Bluetooth

Elaborato finale - Tesi Sperimentale

Laureando
Damiano Serpetta

Matricola 100636

Relatore
Rosario Culmone

Indice

1	Introduzione	9
2	Analisi e Progettazione	11
2.1	Descrizione del progetto	11
2.1.1	Asincronia	13
2.1.2	Anonimato	13
2.1.3	Obiettivo	13
2.1.4	Requisiti fondamentali	14
2.2	Ricerca	14
2.2.1	Bluetooth	14
2.2.2	Ambienti di sviluppo	17
2.3	App Android	17
2.3.1	Requisiti funzionali	17
2.3.2	Casi d'uso	19
2.3.3	Progettazione	20
2.3.4	Pattern strutturale: MVVM	20
2.4	Server	21
2.4.1	Requisiti funzionali	21
2.4.2	Use Case	22
2.4.3	Progettazione	23
2.4.4	Pattern architetturale: MVC	23
3	Implementazione	25
3.1	Applicazione Android	25
3.1.1	MVVM	26
3.1.2	Librerie	27
3.1.3	Activity	28
3.1.4	Coroutines	28
3.1.5	View Model	29
3.1.6	Binding dei dati	30
3.1.7	Permessi	30
3.1.8	Bluetooth	32
3.1.9	Rilevamento dei dispositivi	34
3.1.10	Tracciamento	34

3.1.11 Anonimato	35
3.1.12 Dati	35
3.1.13 Scambio dati con il Server	36
3.1.14 User Interface	39
3.1.15 Gestione degli errori	40
3.2 Server	42
3.2.1 Librerie	42
3.2.2 Server Web	43
3.2.3 Express	44
3.2.4 View	44
3.2.5 Database	45
3.2.6 Routes API	47
3.2.7 Controller	48
3.2.8 Service	49
3.2.9 Repository	51
3.2.10 JWT	51
3.2.11 Routes di visualizzazione	51
3.2.12 Visualizzazione dei dati	52
3.2.13 User Interface	54
3.2.14 Gestione degli errori	56
3.2.15 Autenticazione	57

Elenco delle figure

2.1	Versioni del Bluetooth nel tempo a confronto.	15
2.2	Architettura per il rilevamento dei dispositivi Bluetooth	16
2.3	Casi d'uso dell'App	19
2.4	MVVM Pattern in Android	21
2.5	Casi d'uso del Server	22
2.6	MVC Pattern	24
3.1	Architettura dell'App Android.	27
3.2	Ottenimento dell'istanza di Retrofit.	28
3.3	Esempio di implementazione degli oggetti MutableLiveData e LiveData.	29
3.4	Implementazione del Data Binding.	30
3.5	Workflow per l'uso dei permessi in Android.	31
3.6	Funzione di callback quando i permessi vengono soddisfatti.	32
3.7	Classe per la gestione dell'interfaccia Bluetooth.	33
3.8	Classe per la cattura degli eventi Bluetooth.	34
3.9	Aggiunta delle informazioni del tracciamento tramite Observer.	35
3.10	Funzione di Hash SHA-256 per l'anonimato.	35
3.11	Esempio di classe dati per la rappresentazione dei contatti.	36
3.12	Invio dei contatti al Server.	37
3.13	Funzione POST implementata nell'interfaccia per la connessione con il DB.	38
3.14	Esempio di funzione per l'ottenimento della lista dei contatti effettuati da un device.	38
3.15	User interface dell'App.	39
3.16	Configurazione Express	43
3.17	Configurazione Routes	44
3.18	Pagina web di visualizzazione dati	44
3.19	Connessione al DBMS MongoDB	46
3.20	Implementazione delle operazioni CRUD per gli utenti	47
3.21	Implementazione delle operazioni CRUD dei Contatti	48
3.22	Controller dei dati in ingresso per le operazioni degli utenti	49
3.23	Controller dei dati in ingresso per le operazioni dei contatti	49
3.24	Business logic relativa all'operazione POST dei contatti	50
3.25	Operazione POST nel database attraverso il repository.	51

3.26 Funzione di validazione token.	52
3.27 Routes per la visualizzazione delle pagine web.	52
3.28 Esempio di pagina di visualizzazione dei contatti di un determinato utente.	53
3.29 Funzione AJAX della pagina web di visualizzazione dati.	54
3.30 Layout HTML utilizzato per Handlebars.	55
3.31 Codice jQuery per l'animazione della Navigation Bar.	56
3.32 Gestione degli errori HTTP.	57

Elenco delle tabelle

3.1	User Schema del Database	46
3.2	Contact Schema del Database	47

1. Introduzione

Nel corso degli ultimi mesi il mondo è stato colpito da un'improvvisa malattia che ha sovvertito la vita, le abitudini e le relazioni tra le persone di qualsiasi nazione, indistintamente, rendendoci così distanti ma al contempo **vicini e uniti nella lotta a questa battaglia**. Questo ha portato inevitabilmente a ridimensionare il proprio stile di vita, il rapporto con le altre persone e il modo di vedere il mondo, a far fronte alla paura e a combattere il virus anche con nuovi strumenti che prima non erano di uso comune. Tra gli strumenti nuovi proposti in ambito informatico vi è la nota applicazione "*Immuni*" per smartphone, nata come soluzione al problema del tracciamento dei contagi, di fondamentale importanza per il contrasto della pandemia e per il controllo delle zone più contagiate. Si è visto, oggi più che mai, quanto sia forte la responsabilità individuale di ogni cittadino per potersi proteggere e per proteggere l'altro.

Nasce da qui, da uno strumento innovativo per il **tracciamento dei contagi** quale l'app "*Immuni*", l'idea di un'alternativa che fosse completamente anonima, scalabile e senza la necessità di uno scambio di informazioni tra i dispositivi affinché il tracciamento sia efficace.

L'idea alla base di questo progetto è lo sviluppo di una **rete di tracciamento**, che funziona tramite la raccolta dei dati **anonimi** in modo **asincrono** tra diversi dispositivi, i quali inviano anonimamente i propri dati sul tracciamento. In questo modo, non vi è la necessità che ci sia uno scambio di informazioni tra i dispositivi e ciò fa sì che non si necessita che le due parti abbiano contemporaneamente installato l'applicazione sul proprio *smartphone*.

Questo progetto di Tesi di Laurea si compone di **due progetti separati**. Il **primo progetto** riguarda lo sviluppo di un'**applicazione** per sistemi con *Android*, essa è lo strumento di tracciamento che ogni utente usa per la rilevazione dei dispositivi adiacenti. Il **secondo progetto** concerne la creazione e sviluppo di un **Server** che gestisca i dati online dei suddetti dispositivi.

La tesi è di tipo **sperimentale**, il progetto inizia da una proposta di sperimentazione di diverse applicazioni dell'Informatica per la semplificazione della quotidianità e come strumento ausiliare, come si è fatto esempio, per la prevenzione dei contagi.

Il progetto in questione, è stato ideato e sviluppato singolarmente con la supervisione del relatore **Rosario Culmone**, come nuovo progetto.

2. Analisi e Progettazione

Il progetto in questione nasce da una serie di **colloqui** svolti in sede universitaria e a distanza con il professor **Rosario Culmone** in cui sono stati attraversati i passaggi necessari per l'**analisi** e la **progettazione** di un software.

Nel corso degli incontri, è nata l'idea di un **applicativo** che potesse ereditare alcune funzionalità delle attuali applicazioni già pubblicate in materia di tracciamento, si consideri l'applicazione Immuni, ma che potesse estenderle con altre funzionalità, in modo da rendere l'esperienza utente e collettiva completa. Insieme al professor Culmone, sono stati poi definiti i **requisiti funzionali** e **non funzionali** che interessano lo sviluppo del progetto. I requisiti software del progetto sono stati redatti e descritti per integrità e completezza delle informazioni in un successivo momento: essi hanno costituito una solida base per la progettazione del sistema.

La specifica dei requisiti software ha preceduto la successiva fase di modellazione degli stessi per la produzione di diagrammi **UML**¹ (*Unified Modeling Language*), astrazioni dei concetti fondamentali del progetto, fornendo una visualizzazione grafica del dominio dell'applicazione e dei requisiti che lo compongono.

La fase di definizione dei requisiti software è stata accompagnata da un lungo percorso di ricerca di fattibilità, infatti, uno dei requisiti funzionali fondamentali per lo sviluppo del software è l'utilizzo della scheda **Bluetooth** dei dispositivi mobili, e in particolare l'architettura *Bluetooth Low Energy*², la quale permette non solo la **rilevazione** dei dispositivi adiacenti al proprio Smartphone - funzionalità concessa da tutte le schede Bluetooth - ma anche di registrare il **quantitativo di energia necessaria** per rilevare un dispositivo vicino, e ciò consente di poter approssimare la vicinanza ad esso.

2.1 Descrizione del progetto

Il progetto riguarda lo sviluppo di una rete anonima di tracciamento asincrono attraverso la rilevazione dei dispositivi via Bluetooth.

Analizzando il problema, esso si può scomporre in due progetti principali:

- 1 La **prima parte**, che interessa lo sviluppo di una rete di tracciamento asincrono, è individuata nella produzione di un **Server** che riceva, invii e mantenga i dati in un *database in rete*.

¹Linguaggio di modellazione e di specifica basato sul paradigma orientato agli oggetti

²Tecnologia wireless progettata per un minor consumo energetico e costo ridotto

- 2 La seconda parte**, che concerne la rilevazione dei dispositivi via Bluetooth, si risolve tramite lo sviluppo di un'**applicazione per dispositivi mobili** con sistema operativo Android.

La rete di tracciamento è costituita da un database che contiene tutti i tracciamenti effettuati dagli utenti che hanno avviato la scansione attraverso l'applicazione. Il Server riceve e gestisce le chiamate **HTTP³** (*Hypertext Transfer Protocol*) tramite le proprie **API⁴** (*Application programming interface*) e, tramite la connessione al database, provvede a fornire i dati richiesti.

Per quanto riguarda il tracciamento, all'interno del database vengono mantenute le seguenti informazioni:

- 1.** ID del contatto
- 2.** ID del dispositivo che ha tracciato
- 3.** ID del dispositivo che è stato tracciato
- 4.** Data di rilevamento
- 5.** Distanza tra i dispositivi
- 6.** Geo-localizzazione (facoltativa)

Per quanto concerne il dispositivo, invece, vengono salvati solo i seguenti dati:

- 1.** ID del dispositivo
- 2.** Data di registrazione, ovvero la data della prima volta che il dispositivo traccia un altro dispositivo o viene tracciato

Tramite la raccolta dei dati descritti, si crea una **rete di tracciamento** tra dispositivi rilevati in modo asincrono, ovvero senza che avvenga uno scambio di dati tra le due parti. Questo permette inoltre di tenere una storia di tracciamento di un dispositivo anche senza che quest'ultimo abbia installato il software nel proprio dispositivo.

La rilevazione dei dispositivi avviene con il software per dispositivi Android, che provvede, con l'ausilio della scheda Bluetooth, a rilevare, elaborare e inviare i dati al Server, che li convalida e memorizza. La raccolta dei dati relativi al tracciamento fa uso del BLE (*Bluetooth Low Energy*) per permettere non solo il **risparmio del quantitativo di energia** necessario per rilevare gli altri dispositivi, ma anche alla **registrazione della distanza** che vi è tra il dispositivo che esegue la scansione e il dispositivo rilevato.

I dati registrati in forma **anonima** dei dispositivi contribuiscono a creare una serie di dati elaborabili, eventualmente utili in diversi ambiti, come la **prevenzione** dei contagi di un virus, cui a tal proposito si è rilevato fondamentale quanto essa sia importante e quanto l'ausilio di strumenti anche informatici possa dare un **contributo** positivo alla questione.

³Protocollo a livello applicativo usato come principale sistema per la trasmissione d'informazioni sul web ovvero in un'architettura tipica client-server

⁴Set di definizioni e protocolli con i quali vengono realizzati e integrati software applicativi

2.1.1 Asincronia

Con il termine **asincrono** si intende un sistema che non richiede una sincronia diretta tra le parti, ovvero non vi è uno scambio di informazioni né tantomeno un accordo comune per la raccolta di informazioni relative al tracciamento. Infatti, come descritto nei precedenti paragrafi, il progetto ha lo scopo di poter creare e fare crescere una rete di tracciamento grazie all'apporto degli utenti singoli che daranno alla rete, e *non a coppie di utenti*.

2.1.2 Anonimato

Il fattore **anonimato** costituisce una prerogativa importante nello sviluppo del progetto. Il tema del tracciamento dei dispositivi è fonte di discussione in termini di riservatezza e protezione dei dati sensibili di ciascun utente. Il dato personale con cui si potrebbe risalire al dispositivo e all'utente utilizzatore è l'indirizzo MAC⁵ (*Media Access Control*).

MAC Address

Il **MAC Address** è un indirizzo di rete che viene assegnato in fase di produzione a ciascuna interfaccia di rete (*NIC o Network Interface Controller*) e la identifica globalmente in modo univoco. La scheda Bluetooth di ogni dispositivo presenta quindi il proprio indirizzo MAC, che non identifica direttamente il dispositivo possessore, ma che comunque è un elemento strettamente collegato ad esso, e quindi una possibile **sorgente di dati personali**.

La risoluzione del problema della tutela della privacy avviene memorizzando non il MAC Address del dispositivo mobile, ma un codice identificativo unico ottenuto tramite una funzione di crittografia **hash** sul MAC Address. La funzione hash è progettata per essere **unidirezionale**: si può conoscere il valore crittografato partendo dall'indirizzo MAC, ma non si può conoscere il MAC Address partendo dal valore crittografato, se non verificando tutte le combinazioni possibili con metodi, in termini tecnici, '*brute-force*'.

2.1.3 Obiettivo

L'obiettivo del progetto di Tesi sperimentale è l'**analisi, progettazione e sviluppo** di un sistema che soddisfi tutti i requisiti funzionali e non, rilevati nei colloqui di analisi. Vi è inoltre la ricerca delle tecnologie e gli strumenti necessari per il completamento dei requisiti software. Uno dei passi principali e fondamentali è rappresentato dallo **studio** della tecnologia Bluetooth, le sue versioni, funzionalità e modi di implementazione per lo sviluppo del progetto. Inoltre, da una prima analisi del software si sono ricercate e definite le tecnologie di sviluppo da utilizzare, in base alle necessità che sono state proposte per questo progetto.

⁵Codice di 48 bit (6 byte) assegnato in modo univoco dal produttore ad ogni scheda di rete ethernet o wireless prodotta al mondo

2.1.4 Requisiti fondamentali

I requisiti **fondamentali** rappresentano quei requisiti che definiscono in modo significativo le tecnologie che devono esser utilizzate per lo sviluppo del sistema. Infatti, le tecnologie sono state selezione in base principalmente ai requisiti fondamentali che caratterizzano il sistema.

Qui riportati i **requisiti fondamentali** del sistema:

1. Il **Sistema** deve poter accedere alla scheda Bluetooth del dispositivo
2. Il **Sistema** deve poter rilevare i dispositivi Bluetooth nelle circostanze
3. Il **Sistema** deve poter registrare informazioni dei dispositivi rilevati
4. Il **Sistema** deve poter calcolare la distanza tra esso e il dispositivo rilevato
5. Il **Sistema** deve poter immagazzinare i dati rilevati in un database in rete
6. Il **Sistema** deve poter fornire la possibilità di una visualizzazione online del tracciamento
7. Il **Sistema** deve richiedere l'autorizzazione per l'accesso alle azioni ristrette nel dispositivo
8. Il **Sistema** deve richiedere una conferma per l'avvio del Bluetooth.
9. Il **Sistema** deve poter inviare i dati ad un Server online.
10. Il **Sistema** deve poter ricevere i dati da un Server online.
11. L'**Utente** deve poter visualizzare i dati relativi al proprio dispositivo
12. L'**Utente** deve poter avviare la rilevazione dei dispositivi

2.2 Ricerca

La ricerca è il processo in cui le tecnologie da utilizzare vengono studiate per favorirne un'implementazione efficiente. Vengono studiati tutti gli strumenti messi a disposizione dalle tecnologie e si analizza la sua **architettura**.

2.2.1 Bluetooth

Il **Bluetooth** è uno standard di comunicazione *wireless* (senza fili) per lo scambio di dati tra dispositivi a corto raggio, che utilizza la banda radio a onde corte UHF⁶ (*Ultra high frequency*). Il Bluetooth opera su onde radio tra 2.4 e 2.485 Ghz e, secondo il più recente standard, permette la connessione fino a 7 dispositivi contemporaneamente. Ogni scheda Bluetooth crea una piccola rete sicura in cui scambiare dati tra due dispositivi a intervalli temporali alternati, con architettura *master-slave*⁷.

⁶Segnali a radiofrequenza trasmessi nella banda che va da 300 MHz a 3 GHz, ossia lunghezza d'onda tra 1 e 0,1 m

⁷Tipo di architettura che crea un rapporto gerarchico tra due hardware, il master controlla il canale di comunicazione mentre lo slave risponde.

L'intervallo temporale con cui un dispositivo può inviare o ricevere i dati è pari a 312,5 microsecondi, il *master* invia le informazioni negli intervalli dispari e riceve negli intervalli pari, mentre lo *slave* ricevono negli intervalli dispari e inviano i dati negli intervalli pari.

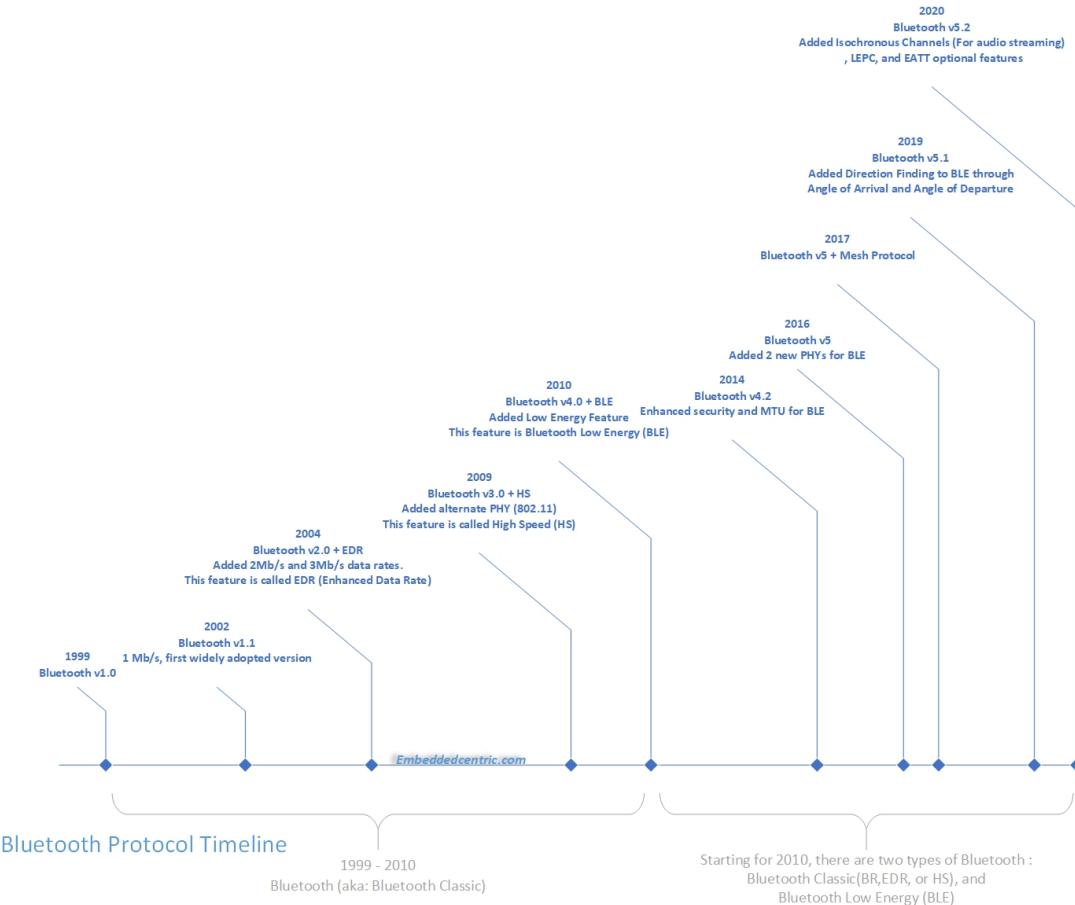


Figura 2.1: Versioni del Bluetooth nel tempo a confronto. Fonte ⁸

Il Bluetooth ha un raggio massimo di azione che spazia tra **10 e 100 metri** in base alla classe Bluetooth di riferimento, questo fornisce un elemento importante per lo sviluppo del progetto, poiché permette di identificare e rilevare i dispositivi vicini.

Le diverse versioni rilasciate nel tempo, dal 1998 al 2021, hanno portato sempre più caratteristiche innovative per la connessione di dispositivi a breve distanza: tra queste, vi è il Bluetooth Low Energy, tecnologia in grande parte presente nei dispositivi moderni, che ha permesso di reinventare l'uso del Bluetooth, con un apporto significativo per i dispositivi IoT⁹ (*Internet of Things*).

Bluetooth Low Energy

Dal Bluetooth 4.0 è stato introdotto il **Bluetooth Low Energy**, una particolare tecnologia che permette di *modulare* l'intensità del segnale che viene inviato per comunicare

⁸<https://www.embeddedcentric.com/introduction-to-bluetooth-low-energy-bluetooth-5/>

⁹Processo di connessione di oggetti di uso quotidiano a Internet, con intervento manuale limitato.

con gli altri dispositivi: in questo modo, oltre a risparmiare un grande quantitativo di energia, è possibile conoscere l'intensità del segnale inviato, e approssimare una distanza fra i dispositivi.

L'intensità del segnale radio viene misurato in **decibel** (db), indicata come Received signal strength indication (o *RSSI*), e spazia tra un valore di 0db e -120db.

Stima della distanza tramite RSSI

L'approssimazione della distanza in **metri** avviene secondo la seguente formula:

$$Distanza = 10^{((MeasuredPower-RSSI)/(10*N))}$$

- *Measured Power*: quantitativo di energia, in decibel, corrispondente a 1 m di distanza.
- *RSSI*: valore in decibel dell'energia misurata.
- *N*: costante dipendente dall'ambiente in cui viene utilizzato, valore da 2 a 4.

Applicando la formula sopracritta si ricava la distanza, espressa in metri, tra due diversi dispositivi.

Scansione Bluetooth

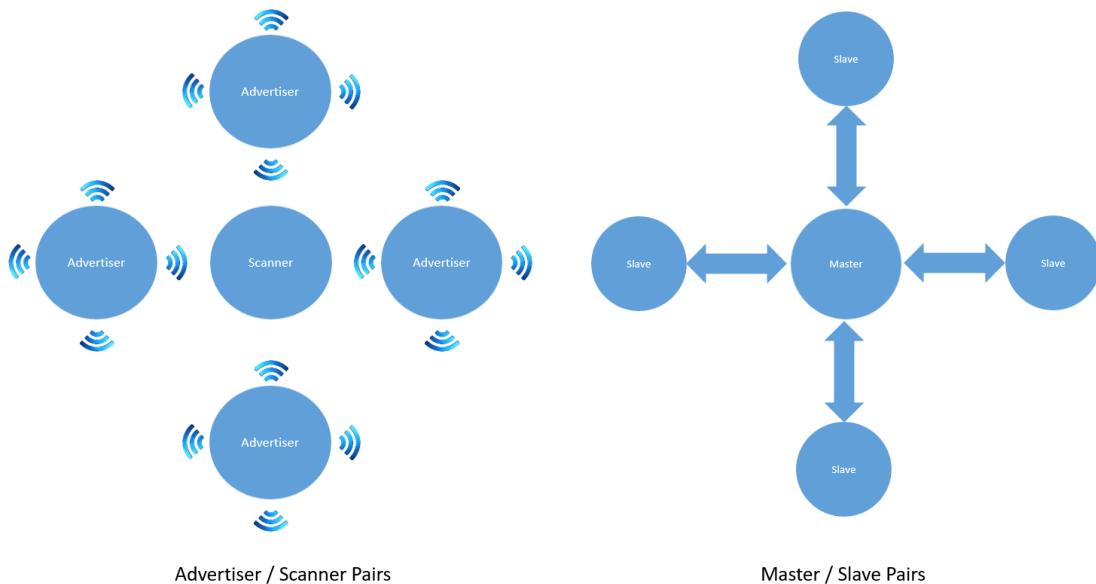


Figura 2.2: Architettura per il rilevamento dei dispositivi Bluetooth. Fonte ¹⁰

La scansione Bluetooth rileva tutti i dispositivi Bluetooth nell'area circostante che sono in modalità '*Discoverable*'. In questa modalità, i dispositivi Bluetooth possono essere scovati dagli altri dispositivi vicini e questo è permesso grazie all'invio in **broadcast**¹¹ di pacchetti da parte del dispositivo che vuol essere rilevato. L'invio dei pacchetti avviene a intervalli regolari nei canali 37, 38, 39 del **canale radio** Bluetooth. Gli intervalli sono definiti dalla modalità in cui sono configurati, possono essere a bassa

¹⁰<https://embeddedcentric.com/introduction-to-bluetooth-low-energy-bluetooth-5/>

¹¹Invio di un messaggio elettronico a tutti gli utenti della rete cui si è collegati

latenza (con un consumo maggiore di energia), ad alta latenza (con un netto risparmio di energia) o in una modalità bilanciata tra le due.

2.2.2 Ambienti di sviluppo

Android

L’ambiente di sviluppo dell’applicativo software per Android è l’IDE (*Integrated development environment*) **Android Studio**. Android Studio permette lo sviluppo in Java, Kotlin e C++ di applicazioni utilizzando componenti e librerie native di Android. **Kotlin** è attualmente il linguaggio di programmazione che, oltre a essere il più recente ad essere stato ideato e sviluppato, è consigliato direttamente da Google per lo sviluppo di Applicazioni per Android.

Server

Per quanto concerne l’applicativo che funge da Server, si è optato per **Express.js**, un framework¹² per applicazioni web per *Node.js*. **Node.js** è un sistema di runtime che esegue codice Javascript, il linguaggio di programmazione utilizzato per lo sviluppo del progetto, lato server. Il runtime di Node.js è orientato ad eventi, multipiattaforma, che rende possibile le operazioni di Input e Output del Server in modo asincrono.

Node.js inoltre, utilizza **npm**, il gestore di pacchetti Javascript open source, pubblici e privati, il quale contiene una collezione di librerie utilizzabili nei progetti dello stesso.

2.3 App Android

2.3.1 Requisiti funzionali

I requisiti funzionali per l’App:

1. Il **Sistema** deve poter gestire la scheda Bluetooth del dispositivo per la scansione.
2. Il **Sistema** deve poter rilevare i dispositivi Bluetooth nelle circostanze.
3. Il **Sistema** deve poter registrare informazioni dei dispositivi rilevati.
4. Il **Sistema** deve poter calcolare la distanza tra esso e il dispositivo rilevato.
5. Il **Sistema** deve poter inviare e ricevere dati dal Server.
6. Il **Sistema** deve poter mostrare le informazioni del tracciamento nell’interfaccia utente dell’App.
7. Il **Sistema** deve poter fornire la possibilità di una visualizzazione online del tracciamento
8. Il **Sistema** deve richiedere l’autorizzazione per l’accesso alle azioni ristrette nel dispositivo

¹²Sistema o infrastruttura che estende funzionalità del linguaggio di programmazione su cui è progettato, al fine di fornire porzioni di codice e classi riutilizzabili.

9. Il **Sistema** deve richiede una conferma per l'avvio del Bluetooth.
10. L'**Utente** deve poter avviare e fermare la scansione dei dispositivi Bluetooth.
11. L'**Utente** deve poter accedere a una pagina web dove visualizzare i dati rilevati.

2.3.2 Casi d'uso

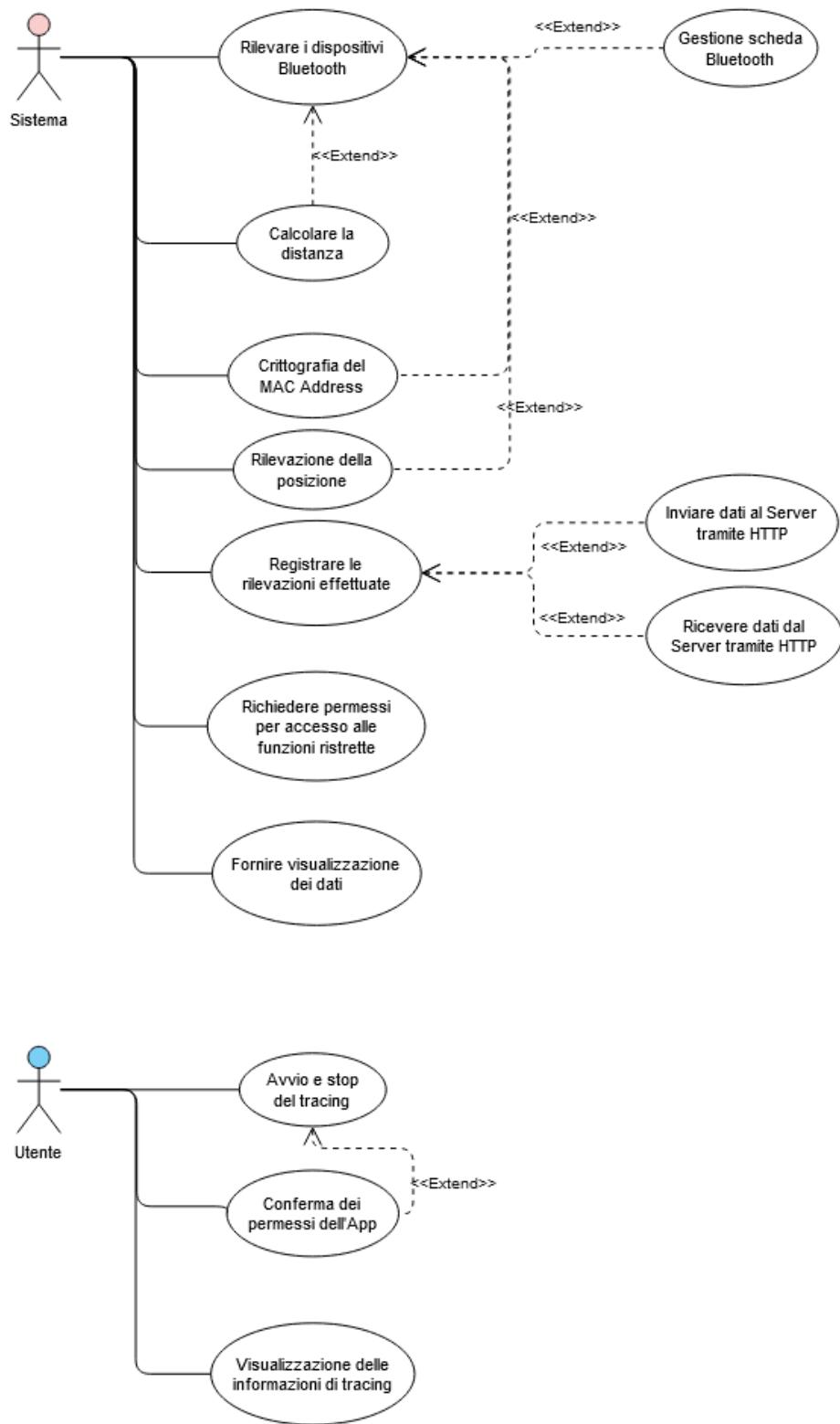


Figura 2.3: Casi d'uso dell'App

2.3.3 Progettazione

Partendo dall'analisi, dalla redazione dei requisiti, dalla modellazione UML e la scelta delle tecnologie di sviluppo, si passa alla progettazione e implementazione del software. I **modelli UML** sono una base su cui partire per la progettazione del software, infatti, da essi poi si passa a implementare le suddette classi nel modello in linguaggio di programmazione, attraverso i paradigmi proposti dal linguaggio scelto.

In questa fase, vengono studiati i **paradigmi** e gli **strumenti** messi a disposizione dal linguaggio di programmazione per far fede allo studio e all'analisi effettuata antecedentemente, infatti ogni linguaggio di programmazione e ambiente di sviluppo è strutturato e struttura il progetto in modi diversi, e mette a disposizione tecniche di programmazione diverse. In questo caso, Android Studio permette e consiglia la programmazione in Kotlin, il quale è molto simile a Java e ne condivide la logica di programmazione. Kotlin è orientato agli oggetti, la base e le fondamenta di questo linguaggio, oltre ad avere delle prerogative per lo sviluppo *multi-threading* e **mobile** per Android.

Nella fase di progettazione si va a definire un **layout grafico** per l'interfaccia utente, semplice, intuitivo e di facile utilizzo. E' importante la definizione del layout grafico per l'implementazione poiché definisce come l'utente interagisce con il software, e perciò adattarlo di conseguenza.

2.3.4 Pattern strutturale: MVVM

Il Model-View-ViewModel è un **pattern architetturale** che separa l'interfaccia utente e tutto ciò che non riguarda l'interfaccia utente. L'interfaccia utente, che è scritta in **XML** (*eXtensible Markup Language*), viene associata ai dati tramite il **Data Binding**, una particolare procedura per lo scambio dei dati tra interfaccia utente e *business logic*. Le dipendenze tra i vari moduli dell'applicazione sono così ridotte, questo ne facilita e migliora il mantenimento e l'estensione. Il **MVVM** è un esempio di **Separation of Concerns**¹³, un concetto importante per tutti i pattern architetturali.

Strati del MVVM:

1. Il **Model** definisce la parte di applicazione che implementa il modello di dominio, incluso *business logic* e tutto il **core** del software. Questo strato è completamente indipendente dalla View e dal ViewModel.
2. La **View** definisce l'interfaccia utente tramite XML. Tutto ciò che è presente in questo strato ha il solo scopo della visualizzazione dei dati che si trovano nel ViewModel. Può anche essere presente della logica, ma al solo scopo rappresentativo.
3. Il **ViewModel** è lo strato intermedio che unisce il Model con la View, ovvero i dati elaborati con la visualizzazione degli stessi. Questo strato si occupa della preparazione dei dati per la View, e in questo modo può contenere funzionalità ad esempio di ordinamento o di conversione dei dati in dati presentabili per la View.

¹³La separazione degli interessi è un principio di progettazione per separare un programma per computer in sezioni distinte.

MVVM in Android

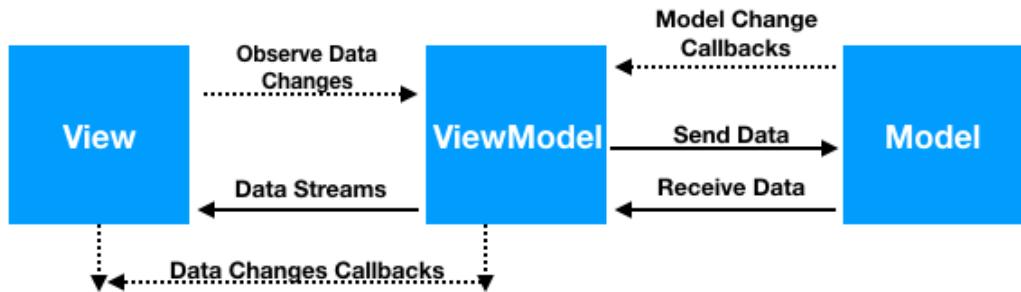


Figura 2.4: MVVM Pattern in Android. Fonte¹⁴

Il MVVM è supportato nello sviluppo Android, in particolare esso fornisce gli strumenti per l'implementazione dei vari componenti e dello **scambio dei dati** tra essi. Android permette l'implementazione del pattern attraverso due importanti strumenti: *Data Binding* e la classe *ViewModel*.

2.4 Server

2.4.1 Requisiti funzionali

I requisiti funzionali per il Server sono:

1. Il **Sistema** deve ricevere e verificare i dati da altre applicazioni in rete.
2. Il **Sistema** deve fornire un'autenticazione per le proprie funzioni.
3. Il **Sistema** deve inviare e ricevere dati dal Database.
4. Il **Sistema** deve visualizzare i dati in una pagina web.
5. Il **Sistema** deve fornire un'interfaccia grafica per l'utente.
6. Il **Sistema** deve poter esser raggiungibile in rete.
7. L'**Utente** deve poter accedere alla pagina web per la visualizzazione dei dati.

¹⁴<https://www.journaldev.com/20292/android-mvvm-design-pattern>

2.4.2 Use Case

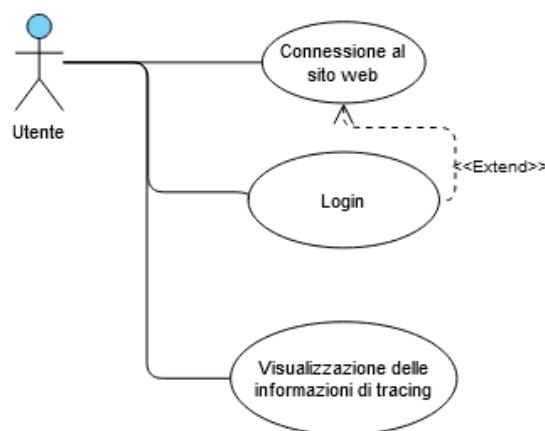


Figura 2.5: Casi d'uso del Server

2.4.3 Progettazione

La progettazione del Server è l'insieme delle procedure per la creazione di un software a partire dalla sua analisi. A partire dai modelli UML (Use-case e Classi) creati e i requisiti funzionali in prima istanza si ricerca un ambiente di sviluppo che copra tutti i requisiti richieste. Nel caso di questo progetto, si è optato per la scelta di un framework **stabile** ed **efficente** per la creazione di un Server Web e di una applicazione web associata per la visualizzazione dei dati.

La scelta è ricaduta su Express.js, che come si vedrà nei prossimi paragrafi, è un framework per Node.js che fornisce gli strumenti, classi e interfacce per la creazione, supporto e mantenimento di un Server Web e un'interfaccia Web. Node.js è un runtime javascript orientato agli eventi, in esso è possibile costruire sopra un Server HTTP e implementare delle API per soddisfare i requisiti.

Per quanto riguarda il database, il nocciolo del software da sviluppare, si è scelto un database non relazione, gestito da **MongoDB**: esso è indicato quando si lavora con una grande mole di dati e si vuol fare anche analisi su di essi. Questa scelta è una scelta fatta anche con una visione a **lungo termine**, infatti una volta che la rete di tracciamento si popola e cresce con i dati dei trackers MongoDB è la **soluzione più efficace** per la loro gestione.

2.4.4 Pattern architetturale: MVC

Il **Model-View-Controller** è un pattern architetturale che separa l'interfaccia utente dalla modellazione dei dati (*business logic* o *dominio*). L'elemento che lega i due componenti separati è il *Controller*, esso si occupa di gestire le interazioni dell'utente e scatenare delle azioni del Model, che elabora i dati e li mette a disposizione per la View.

MVC Architecture Pattern

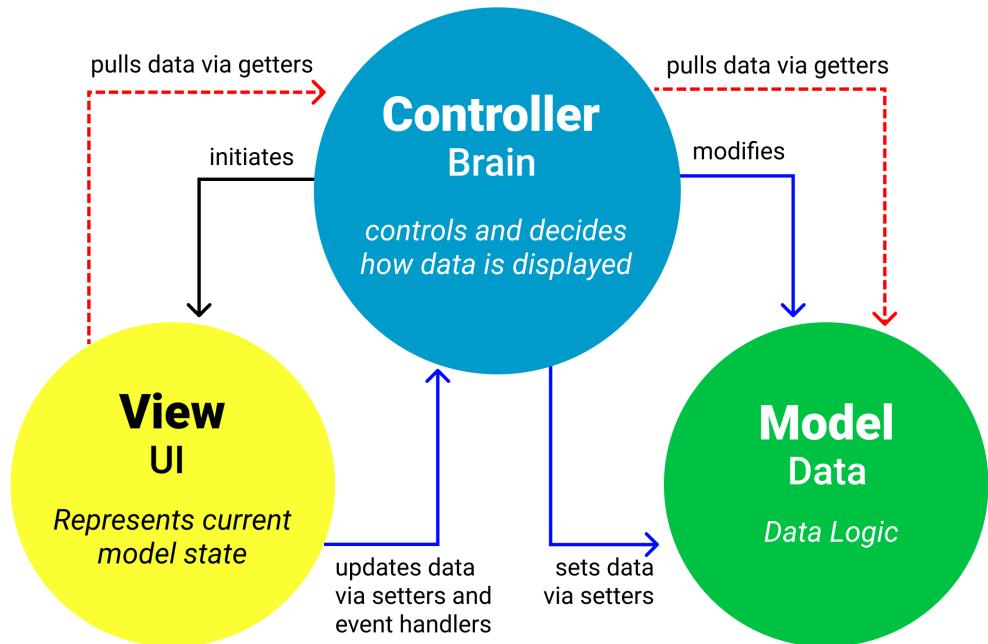


Figura 2.6: MVC Pattern. Fonte¹⁵

In **Express.js** si è optato per l'implementazione del pattern MVC, in questo modo si ha un progetto e un applicativo che è innanzitutto più mantenibile nel tempo ma soprattutto estendibile.

¹⁵ <https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained>

3. Implementazione

3.1 Applicazione Android

L'applicazione per Android è sviluppata mediante l'IDE **Android Studio** e il linguaggio di programmazione **Kotlin**. Kotlin rappresenta l'ultimo standard consigliato da Google per lo sviluppo di applicazioni Android.

Lo sviluppo di applicazioni per Android si divide in due processi fondamentali: la costruzione e implementazione, attraverso **XML**, di un'interfaccia grafica per ogni componente e 'pagina' dell'applicazione e lo sviluppo vero e proprio del **core** dell'applicazione con *Kotlin*.

La configurazione dell'ambiente è molto importante in questo caso, Android Studio prevede l'installazione dell'Android SDK¹ (*Software Development Kit*), del JDK² (*Java Development Kit*), Gradle³ e degli emulatori con Android per l'esecuzione del codice, o alternativamente un dispositivo reale in modalità debug connesso al computer, in cui sono installati i driver ADB⁴ (*Android Debug Bridge*).

Suddivisione del progetto

Android Studio divide il progetto in tre diversi moduli:

1. Il modulo **manifests**, che comprende un file di nome '*AndroidManifest.xml*', che contiene la configurazione globale del progetto. Infatti, nel file, troviamo i permessi che l'applicazione richiede per il suo funzionamento, il nome dell'applicazione, l'icona, il tema, le impostazioni del traffico di rete, le *Activities* che sono comprese nel progetto e i font. Android Studio provvede quindi alla **configurazione** del progetto sulla base dei dati che sono forniti nel file.
2. Il modulo **java**. Esso comprende tutto il codice Kotlin che riguarda la **logica di business** dell'applicazione e tutto ciò che non sia la visualizzazione. In questo modulo sono contenuti i **package** che compongono l'architettura dell'applicazione, essi sono:
 - a. *config*: il package **config** contiene le classi che contengono dati e funzioni per la configurazione delle varie componenti dell'applicazione, come il bluetooth, il database, i token api e le variabili d'ambiente.

¹Insieme di strumenti per lo sviluppo e la documentazione di software Android.

²Insieme degli strumenti per sviluppare programmi da parte dei programmatori Java

³Sistema open source per l'automazione dello sviluppo.

⁴Strumento compreso all'interno del software SDK e usato per mettere in comunicazione un dispositivo Android ed un computer.

- b. *data*: il package **data** aggrega le classi per la gestione dei dati elaborati dall'applicazione, in esso sono contenute le funzioni per la comunicazione con i database locali e remoti (*api, datasource e repository*) e le classi che definiscono gli oggetti con cui l'applicazione rappresenta le istanze degli utenti e dei contatti.
- c. *domain*: il package **domain** comprende tutte le classi che si usano per l'elaborazione dei dati da parte dell'applicazione, vediamo la classe per l'*hashing*⁵, la classe per l'ottenimento dell'istanza per la gestione delle chiamate HTTP, la classe per la gestione dei database all'interno dell'applicazione e infine, la classe che si occupa delle funzioni CRUD rispetto agli utenti e ai contatti.
- d. *viewmodel*: il package in questione, contiene tutti i viewmodel che vengono usati nel progetto. I viewmodel sono particolari classi, secondo l'architettura di Android Studio, che si implementano per la gestione e visualizzazione dei dati all'interno dell'applicazione. Essi rappresentano sostanzialmente lo 'stato' dell'applicazione.
- e. *functions*: il package **functions** contiene le classi statiche per le funzioni di uso comune all'interno dell'applicazione, che non necessitano di oggetti d'istanza.
- f. *la directory principale*: nella **directory principale** vi è la classe relativa all'Activity, il componente principale dell'applicazione Android che funge dal classico **main** nei paradigmi dei linguaggi di programmazione standard. Android inizializza il codice nell'istanza di un'Activity invocando metodi di callback che corrispondono alla specifica fase nel proprio ciclo di vita. Le Activity sono le componenti principali e fondamentali delle applicazioni in Android, e l'implementazione di quest'ultime permettono la cooperazione tra diverse app all'interno del sistema: ad esempio, risulta semplice con le Activity, il lancio di un'Activity di un'altra App tramite l'App sviluppata.

Gradle

Gradle è uno strumento di **automazione della compilazione** del codice sorgente dell'applicazione. Esso è uno strumento, configurato di *default* in Android Studio, che permette di definire e automatizzare gli *step* per la compilazione e l'esecuzione del codice. Gradle permette di configurare i passi che, dai vari file e packages contenuti nel progetto, produce un eseguibile per Android autonomamente. Esso si occupa anche di gestire le dipendenze del progetto, come framework e librerie.

3.1.1 MVVM

Le applicazioni Android hanno una struttura molto complessa, con molte componenti, *activities*, *fragments*, *services*, *content providers* e *broadcast receivers*. Trovare il modo di connettere tutte queste componenti al fine di sviluppare l'App è un **processo complesso e delicato**, per questo si ricorre a pattern architettonici che aiutano nel design e nello sviluppo. Il pattern architettonale implementato in Android Studio è il MVVM, Model-View-ViewModel, il quale rispecchia a pieno il concetto del *separation of concerns*, cioè la suddivisione del progetto in moduli diversi per ogni scopo dello

⁵Funzione non invertibile che mappa una stringa di lunghezza arbitraria in una stringa di lunghezza predefinita.

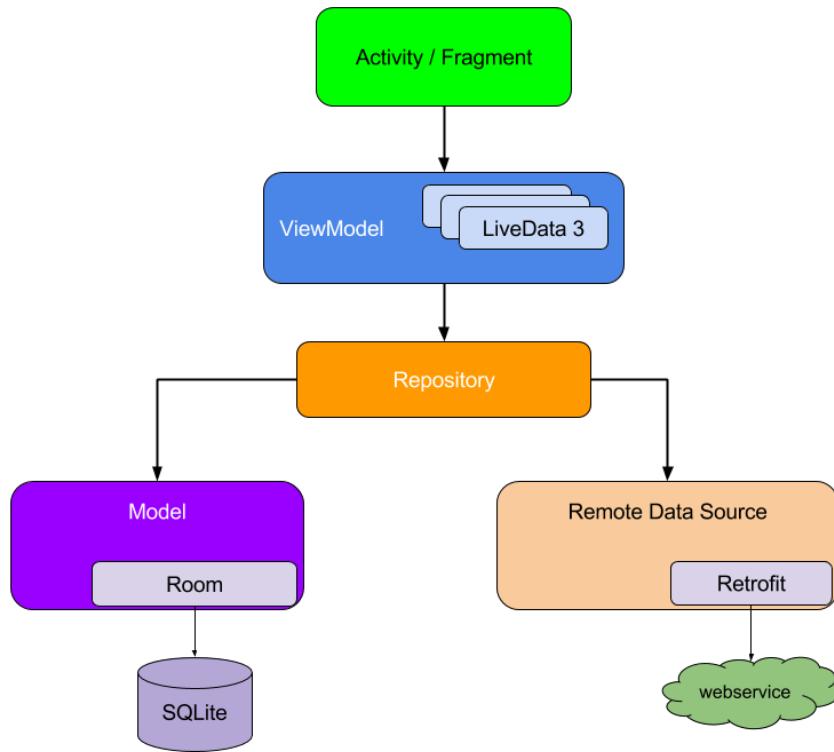


Figura 3.1: Architettura dell'App Android.

sviluppo. L'idea principale del MVVM è quella di guidare e **aggiornare** l'interfaccia utente attraverso il **Model**, ovvero la parte che gestisce i dati persistenti dell'app.

3.1.2 Librerie

Le librerie incluse e utilizzate nel progetto sono definite nella configurazione di Gradle.

Retrofit

Retrofit è un *client HTTP* per Android, semplifica il lavoro della creazione delle API per lo scambio dei dati via HTTP con il Server. La creazione delle API per la comunicazione con il Server remoto avviene attraverso la creazione di interfacce in Kotlin, a cui vengono aggiunte delle *annotazioni* che definiscono l'**end-point**, i metodi e tutti i parametri per l'esecuzione di chiamate HTTP, come *query*, *header* e *body*. Retrofit inoltre, ha un supporto completo per le *coroutines* di Kotlin, i costrutti per l'implementazione del **multi-threading** e **asincronia** nel progetto.

Nel progetto viene importata la classe di Retrofit e istanziato un oggetto per l'esecuzione di esso. L'oggetto viene configurato definendo quali sono le API che devono essere eseguite e il modo per la conversione degli oggetti **JSON** in oggetti Kotlin.

Gson

Gson è la libreria che permette la **conversione** degli oggetti in JSON in oggetti di Kotlin. Essa viene utilizzata, come visto nel paragrafo appena precedente, in correlazione

```
fun getRetrofitInstance(): Retrofit {
    return Retrofit.Builder().baseUrl(BASE_URL)
        .addConverterFactory(GsonConverterFactory.create(GsonBuilder().create()))
        .build()
}
```

Figura 3.2: Ottenimento dell'istanza di Retrofit.

con Retrofit, per la conversione delle risposte HTTP in dati interpretabili da Kotlin.

3.1.3 Activity

Le **Activity**, come descritto nei paragrafi precedenti, oltre ad implementare il codice dell'App, provvedono a le finestre dove l'App disegna la propria User Interface. Ogni finestra in un'applicazione Android corrisponde a una differente Activity.

Le Activity hanno 7 diversi stati che cambiano durante l'esecuzione dell'App:

1. `onCreate()` - Rappresenta lo stato, obbligatorio, che viene eseguito quando l'applicazione viene creata, qui si inizializzano i componenti essenziali dell'App. Qui viene eseguita la funzione `setContentView()` che definisce il layout dell'interfaccia
2. `onStart()` - Si tratta della fase appena successiva alla creazione, è l'ultima fase prima dell'avvio.
3. `onResume()` - E' lo stato in cui l'utente interagisce con l'App e dove la maggior parte del *core* dell'App è contenuto.
4. `onPause()` - E' lo stato in cui l'applicazione entra nello stato di pausa. Questo può essere dovuto ad un'azione dell'utente, ad esempio di tornare indietro alla schermata principale di Android. Lo stato di pausa significa che l'utente non sta interagendo con l'applicazione ma comunque essa si deve aggiornare poiché ci si aspetta un ritorno di interazione.
5. `onStop()` - Rappresenta lo stato in cui l'App si ferma e non è più visibile all'utente. Questo può succedere perché l'App si sta per chiudere in modo definitivo o ad esempio una nuova attività si sta per aprire.
6. `onRestart()` - Si tratta della fase in cui un'App viene ripristinata dopo uno stato di Stop.
7. `onDestroy()` - E' la fase in cui l'App sta per essere chiusa e la memoria viene liberata.

Le Activity si dichiarano nel file Manifest del progetto.

3.1.4 Coroutines

Le **coroutines** in Kotlin sono dei costrutti per la programmazione concorrente in Kotlin e in Android. Kotlin rende il meccanismo del multi-threading e asincronia semplice tramite i coroutines. Le coroutines sono gestite dallo sviluppatore, ed è lo sviluppatore che implementa e definisce come le coroutines devono essere eseguite e in che ambito. Infatti, le coroutines in Kotlin possono avere diversi 'scopes' in base a dove esse vengono

lanciate. Le coroutines vengono utilizzate ampiamente nel progetto per permettere lo **scambio dei dati in maniera asincrona**, così da incrementare le performance dell'App e non permettere che l'interfaccia utente si blocchi mentre aspetta di caricare dei dati.

3.1.5 View Model

Le classi **View Model** sono progettate per immagazzinare e gestire i dati relativi all'interfaccia utente dell'applicazione. Le classi View Model sono importanti anche per mantenere i dati quando la configurazione dell'App cambia, cui senza verrebbero eliminati. Un esempio di cambio di configurazione è la rotazione dello schermo, e senza l'implementazione dei View Model, i dati che compongono l'App verrebbero eliminati. La classe View Model permette l'esecuzione di funzioni asincrone per il recupero e aggiornamento bi-direzionale dei dati, così si può separare la parte dei dati dell'interfaccia utente dalla parte del controller dell'UI (*User Interface*).

I View Model vengono implementati attraverso l'estensione di una classe con la classe *ViewModel()*. Essa fornisce delle classi per la realizzazione dei View Model: infatti, l'implementazione di essi si basa sull'istanziamento di oggetti di tipo **MutableLiveData** e **LiveData**. Questi oggetti, permettono l'immagazzinamento dei dati in una struttura che poi viene *osservata* tramite le funzioni *Observer*⁶ dagli altri componenti e classi. In Figura 3.3 vi è un'implementazione degli oggetti MutableLiveData e LiveData.

```
private var tracingState = MutableLiveData<Boolean>(value: false)
val tracingStateData: LiveData<Boolean>
|   get() = tracingState

private var mIsBluetoothSupported = MutableLiveData<Boolean>(value: false)
val isBluetoothSupported: LiveData<Boolean>
|   get() = mIsBluetoothSupported

private var mIsLocationSupported = MutableLiveData<Boolean>(value: false)
val isLocationSupported: LiveData<Boolean>
|   get() = mIsLocationSupported
```

Figura 3.3: Esempio di implementazione degli oggetti MutableLiveData e LiveData.

Data corrispondenti al View Model principale dell'Activity. In questo caso, sono dati riguardati lo stato del tracciamento e il supporto delle funzionalità di Bluetooth e della rilevazione della posizione. L'oggetto MutableLiveData contiene i veri e propri dati ed è privato, così da non poter essere acceduto dall'esterno. L'oggetto LiveData invece, non fa altro che restituire il valore dell'oggetto che contiene i dati, favorendo un **mecanismo di incapsulamento**, dove i dati possono essere letti direttamente da questo oggetto e impostati tramite le funzioni *create*.

Gli altri View Model implementati nel progetto:

1. Tracing View Model: Questo View Model mantiene una lista di oggetti di classe '*discoveredDevice*', oggetti che mantengono i dati di un dispositivo rilevato du-

⁶Design pattern utilizzato come base architettonica di molti sistemi di gestione di eventi.

```
binding = DataBindingUtil.setContentView(activity: this, R.layout.activity_main)
binding.lifecycleOwner = this
```

Figura 3.4: Implementazione del Data Binding.

rante la scansione. Questa classe comprende anche una lista temporanea parallela e uguale alla lista principale dove vengono aggiunti i nuovi dispositivi, dopodiché il valore viene assegnato alla lista principale: questo è un meccanismo che permette, dato che il valore cambia, che l'Observer intercetti il cambiamento. Questa funzionalità viene implementata nei View Model in cui sono presenti liste.

2. Contacts View Model: Questa classe View Model contiene una lista (sempre di tipo *MutableLiveData*) dove vengono aggiunti gli oggetti di classe *Contact*, ovvero oggetti con soli dati che si riferiscono a un contatto con un altro dispositivo. Questi dati, a differenza del View Model precedente, sono i dati elaborati, contenenti tutte le informazioni aggiunte per il tracciamento, che verranno poi inviati al Server. Anch'esso implementa la lista temporanea per l'innesto dell'Observer.

Implementazione nell'Activity

L'implementazione dei View Model avviene attraverso una classe fornita da Kotlin '*ViewModelProvider*'. Quest'ultima può istanziare i ViewModel senza parametri in ingresso e per questo, si provvede a utilizzare una classe Factory per l'istanziamento dei View Model. La classe Factory rispecchia il medesimo design pattern creazionale 'Factory⁷'.

3.1.6 Binding dei dati

Il **Data Binding** è una libreria di supporto contenuta nell'*Android Jetpack*⁸ che permette di legare i componenti dell'interfaccia utente con i dati dell'applicazione. Il Data Binding permette di eliminare le chiamate ai metodi per la modifica degli elementi nell'interfaccia utente, e permette la **semplificazione** dell'aggiornamento dei dati nell'UI rispetto ai dati dell'App. L'utilizzo del Data Binding migliora anche le **performance** e l'utilizzo della **memoria**, prevenendo anche errori di tipo *Null Pointer Exception*. La libreria per la gestione del Data Binding fornisce anche strumenti per l'osservazione dei cambiamenti dei dati, così da scatenare degli **eventi** in relazione ad essi.

I componenti dell'UI vengono mappati nel binding, così le loro proprietà sono direttamente accessibili dall'oggetto *binding*. Nella Figura 3.4 vi è l'implementazione del Data Binding nel progetto.

3.1.7 Permessi

Android pone molta attenzione alla privacy e alla **gestione dei permessi** da garantire per l'uso dell'applicazione. Infatti, vi sono azioni che hanno delle **restrizioni** sul proprio utilizzo e richiedono una specifica conferma e approvazione da parte dell'utente. L'utente si ritrova a dover accettare delle *richieste esplicite* per ogni azione

⁷Approccio di programmazione con il quale creare oggetti senza bisogno di dover specificare la loro classe.

⁸Suite di librerie per lo sviluppo Android

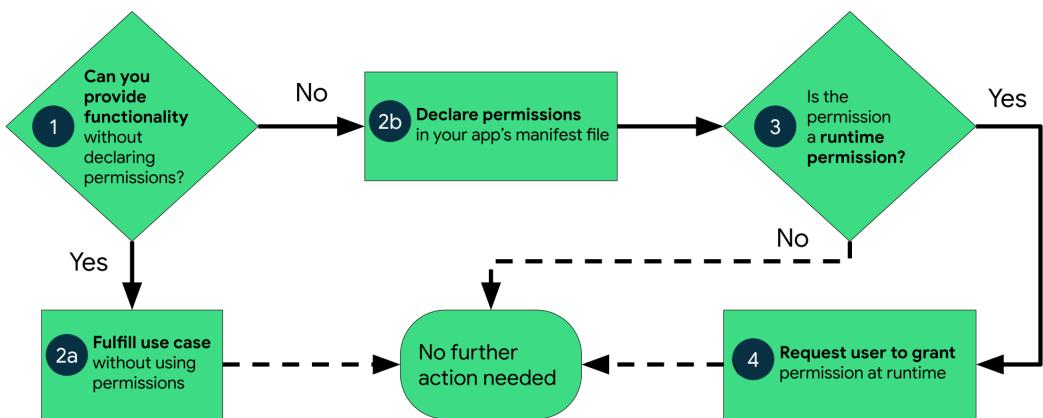


Figura 3.5: Workflow per l'uso dei permessi in Android.

ristretta, in fase di installazione dell'App o all'interno dell'applicazione (chiamati *runtime permissions*), in base al tipo di permesso. E' importante che lo sviluppatore sia **chiaro** con l'utente delle azioni protette che l'App eseguirà ed il perché di queste azioni.

I permessi richiesti per lo sviluppo del progetto sono:

1. Accesso alla **geo localizzazione**, essi permettono l'ottenimento della localizzazione, attraverso il modulo GPS⁹ (*Global Positioning System*), del dispositivo in un determinato istante. Il permesso, viene richiesto con una richiesta esplicita quando l'utente prova ad avviare la rilevazione dei dispositivi. I permessi all'interno del progetto sono identificati in '*ACCESS_FINE_LOCATION*', '*ACCESS_COARSE_LOCATION*' e '*ACCESS_BACKGROUND_LOCATION*'.
2. Accesso all'utilizzo del modulo Bluetooth, essi chiaramente sono i permessi principali all'interno dell'applicazione e si rifanno all'uso normale e in modalità amministratore del Bluetooth, che si necessita per l'uso del Bluetooth per gli scopi proposti dal progetto. Essi, come i permessi relativi alla geo localizzazione, vengono richiesti nell'App quando l'utente prova ad avviare una rilevazione. I permessi all'interno del progetto sono identificati in '*BLUETOOTH*', '*BLUETOOTH_SCAN*' e '*BLUETOOTH_ADMIN*'
3. Accesso all'utilizzo di internet per la comunicazione con il Server, questo è un permesso che viene accettato automaticamente in fase di installazione dell'App. All'interno del progetto, i permessi si riferiscono a '*INTERNET*'

Implementazione

All'interno del progetto, sono implementate due funzioni che gestiscono la **richiesta dei permessi** e il controllo della soddisfazione di essi: esse sono le funzioni invocate nelle parti in cui vi è la necessità di questi permessi, ovvero nel bottone per l'avvio del rilevamento dei dispositivi.

La funzione di controllo dei permessi, controlla se i permessi che abbiamo definito nelle variabili d'ambiente sono stati tutti soddisfatti, e il suo valore di ritorno è un valore

⁹Sistema di posizionamento e navigazione satellitare statunitense.

```

requestPermissionLauncher = registerForActivityResult(
    ActivityResultContracts.RequestPermission()
) { isGranted: Boolean ->
    if (isGranted) {
        Toast.makeText(context: this@MainActivity, text: "Granted!", Toast.LENGTH_SHORT).show()
        if (checkPermissions()) {
            enableBluetooth()
            viewModel.setTracingState(true)
            mBtManager.startScan()
        }
    } else {
        viewModel.setTracingState(false)
        Toast.makeText(context: this@MainActivity, text: "Not Granted!", Toast.LENGTH_SHORT).show()
    }
}

```

Figura 3.6: Funzione di callback quando i permessi vengono soddisfatti.

booleano. Android mette a disposizione delle funzioni di sistema per il **controllo di un permesso**, con in input la stringa di riferimento al permesso.

La funzione di richiesta dei permessi, per ogni permesso definito nella lista nelle variabili d'ambiente, controlla se il permesso è soddisfatto, se non è soddisfatto esegue l'operazione di sistema per la richiesta del permesso e registra una funzione di **callback**¹⁰ da eseguire quando il permesso viene soddisfatto, e in questo caso rappresenta la richiesta di avvio di bluetooth e di conseguenza l'avvio della scansione, se tutti i permessi sono soddisfatti. Nella Figura 3.6 è mostrato il codice per l'implementazione della funzione di callback quando i permessi vengono soddisfatti.

3.1.8 Bluetooth

Il sistema Android include il supporto per il Bluetooth, esso mette a disposizione le **API** necessarie per l'accesso alle funzionalità e all'accesso al Bluetooth. Android fornisce le classi e le interfacce che permettono la **comunicazione** tra l'app e la scheda Bluetooth del dispositivo. Le funzionalità messe a disposizione da Android per quanto riguarda il Bluetooth sono molteplici e riguardano principalmente la **connessione e lo scambio dati** tra due dispositivi. Ciò che si necessita nel progetto, rispettando l'obiettivo e i requisiti, è che il tracciamento sia asincrono, ovvero senza scambio di dati tra le due parti. Per questo motivo, si necessita solo la parte inherente alla rilevazione dei dispositivi attorno (*Discovery*). Android fornisce la classe '*AndroidAdapter*' che rappresenta l'**interfaccia radio** del Bluetooth: essa è la porta per tutte le interazioni con il Bluetooth e tutte le funzioni associate.

Gestore Bluetooth

Nel progetto si è creata una classe per l'istanziamento e la gestione dell'interfaccia Bluetooth, in questa classe vi è l'oggetto che comunica **direttamente** con la scheda Bluetooth del dispositivo. La classe è *TracingBluetoothManager*, in Figura 3.7, e l'oggetto corrispondente viene istanziato nel *MainActivity* in fase (nel ciclo di vita dell'app) di creazione dell'App.

La classe provvede a controllare l'oggetto *BluetoothAdapter* e, come visto in figura, vi

¹⁰Funzione trasmessa a un'altra funzione come parametro.

```

class TracingBluetoothManager(_context: Context) {

    // Bluetooth
    private var mBluetoothManager: BluetoothManager? = null
    private var mBluetoothAdapter: BluetoothAdapter? = null

    init {
        // Inizializza oggetto per la gestione del bluetooth.
        mBluetoothManager = _context.getSystemService(Context.BLUETOOTH_SERVICE) as BluetoothManager
        mBluetoothAdapter = mBluetoothManager?.adapter
    }

    fun isBluetoothSupported(): Boolean {
        return (mBluetoothAdapter != null)
    }

    fun isBluetoothEnabled(): Boolean? {
        return mBluetoothAdapter?.isEnabled
    }

    fun startScan(): Boolean? {
        return mBluetoothAdapter?.startDiscovery()
    }
}

```

Figura 3.7: Classe per la gestione dell’interfaccia Bluetooth.

è la funzione *startScan* per avviare la scansione dei dispositivi Bluetooth.

Quando si avvia una scansione, il Bluetooth inizia il rilevamento dei dispositivi vicini che vengono salvati in un **buffer locale**: il buffer locale su cui vengono salvati i dispositivi viene poi catturato da un’altro oggetto nell’App, che provvede a salvarli su di un ViewModel. L’oggetto che cattura l’evento di rilevamento di un nuovo dispositivo ha il nome di Broadcast Receiver: esso implementa una funzione chiamata ‘*onReceive()*’ che viene eseguita quando una specifica azione o evento si soddisfa. In questo caso l’evento che si deve soddisfare per l’esecuzione del metodo callback è l’evento Bluetooth ‘*ACTION_FOUND*’, esso rappresenta la rilevazione di un dispositivo e viene innescato dalla classe che gestisce il dispositivo Bluetooth.

In questo modo è possibile favorire il funzionamento **asincrono** del sistema che si è creato con l’App, nessun componente ha bisogno di aspettare attivamente nessun’altro componente, ma si scatena solo quando un determinato evento viene soddisfatto.

Broadcast Receiver

In figura 3.8 si mostra l’implementazione della classe che cattura l’evento Bluetooth relativo alla ‘scoperta’ di un nuovo dispositivo bluetooth. Lo scopo della classe è quello di intercettare i dispositivi bluetooth e salvarne i dati quali: *MAC Address* e *RSSI*, che si ricorda essere l’intensità di energia che il dispositivo rilevante ha impiegato per rilevare questo dispositivo.

La classe in questione estende la classe *BroadcastReceiver* e sovrascrive il metodo che fornisce la classe: *onReceive*.

```

class TracingBluetoothBR(_viewModel: TracingViewModel) : BroadcastReceiver() {

    private var viewModel = _viewModel

    override fun onReceive(context: Context, intent: Intent) {
        when (intent.action) {
            BluetoothDevice.ACTION_FOUND -> {
                val btDevice: BluetoothDevice =
                    intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE)!!
                val deviceHardwareAddress = btDevice.address // MAC address
                val deviceRSSI: Short =
                    intent.getShortExtra(BluetoothDevice.EXTRA_RSSI, Short.MIN_VALUE)

                val device = DiscoveredDevice(deviceHardwareAddress, deviceRSSI)
                viewModel.addDeviceInfo(device)
            }
        }
    }
}

```

Figura 3.8: Classe per la cattura degli eventi Bluetooth.

3.1.9 Rilevamento dei dispositivi

Il rilevamento dei dispositivi, come visto nei paragrafi precedenti, avviene tramite una cooperazione asincrona tra l’interfaccia Bluetooth e un oggetto che intercetta i dati che l’interfaccia produce. L’oggetto che intercetta i dati e ne ricava le informazioni utili, crea un oggetto di classe ‘*DiscoveredDevice*’ e lo aggiunge a una lista di dispositivi rilevati che implementa il modello ViewModel, così che gli altri oggetti possano vederne il contenuto. L’oggetto di classe ‘*DiscoveredDevice*’ è una classe di tipo dati che contiene il MAC Address e l’RSSI del dispositivo.

3.1.10 Tracciamento

Dopo l’ottenimento una lista dei dispositivi incontrati, il prossimo passo da fare è quello di incapsulare questi dati con altri dati utili al fine del tracciamento: si crea quindi, un’altro ViewModel che conterrà tutti i dispositivi con dati completi, pronti per essere mandati al Server. Il ViewModel corrispondente, nel progetto, ha il nome di *contactsViewModel* e implementa la stessa struttura del ViewModel per il rilevamento dei dispositivi. Questa struttura però, conterrà oggetti di tipo ‘**Contact**’, ovvero oggetti con i dati completi per ogni contatto.

I dati aggiunti ad ogni dispositivo rilevato, vengono aggiunti mediante la classe **ContactTracer** che implementa una funzione chiamata *createContactOnDiscoveredDevice*. La funzione elabora i dati già presenti e ne aggiunge altri:

1. Distanza: la distanza viene calcolata con un’apposita funzione, la cui espressione viene descritta nel Capitolo 2 riguardante la Progettazione.
2. Data: la data di rilevamento viene impostata con la data corrente.
3. Posizione: la geo localizzazione sfrutta l’oggetto locationManager della classe LocationManager, direttamente fornita da Android. Tuttavia, la funzione di geolo-

calizzazione non è ancora implementata nel progetto ma esso è impostato per il completo supporto all'aggiunta di un servizio di locazione.

```
tracingViewModel.discoveredDevices.observe( owner: this, { it: MutableList<DiscoveredDevice>!
    try{
        val contact =
            mContactTracer.createContactOnDiscoveredDevice(viewModel.getMacAddress(), it.removeLast())
        contactsViewModel.addContact(contact)
    } catch (err : NullPointerException){
        Log.i( tag: "Discover Error", msg: "Null pointer exception")
    }
})
```

Figura 3.9: Aggiunta delle informazioni del tracciamento tramite Observer.

I dati così, una volta aggiunti i dati del tracciamento, vengono inseriti in un nuovo ViewModel 'contactsViewModel', una lista che contiene tutti i contatti che sono pronti per essere elaborati e immagazzinati nel Server.

3.1.11 Anonimato

L'anonimato nel progetto viene assicurato dall'hashing **SHA-256** (*Secure Hash Algorithm 256*) che viene eseguito nelle stringhe dei MAC Address. Lo SHA-256 è un hash, una funzione di crittografia unidirezionale che permette di creare una stringa che **non può essere inversa**. Questo significa che solo chi conosce la stringa originale, ovvero il proprio MAC Address, riesce a identificare la stringa crittografata. In questo modo, nessun utente sarà a conoscenza delle persone che ha rilevato nei propri tracciamenti.

Hash

La funzione di Hash SHA-256 viene implementata nel progetto tramite una classe statica. La funzione, che prende in input una stringa, restituisce il valore crittografato, a livello **client**.

```
private const val algorithm : String = "SHA-256";
fun toHash(_input : String):String {
    return MessageDigest.getInstance(algorithm)
        .digest(_input.toByteArray())
        .fold( initial: "", { str, it -> str + "%02x".format(it) })
}
```

Figura 3.10: Funzione di Hash SHA-256 per l'anonimato.

3.1.12 Dati

I dati nell'applicazione vengono gestiti attraverso le apposite classi che rappresentano ognuno di essi. Le classi per rappresentare i dati non hanno funzioni, esse si chiamano **data class** nel linguaggio Kotlin. Nel progetto vi sono tre classi per i dati:

```
data class Contact(
    @SerializedName( value: "deviceDetector")
    val deviceDetector: String,
    @SerializedName( value: "deviceDetected")
    val deviceDetected: String,
    @SerializedName( value: "distance")
    val distance: Double,
    @SerializedName( value: "position")
    val position: String,
    @SerializedName( value: "contactDate")
    val contactDate: Date
)
```

Figura 3.11: Esempio di classe dati per la rappresentazione dei contatti.

1. DiscoveredDevice: la classe che definisce gli oggetti rilevati dal bluetooth, essi hanno solo due attributi: MAC Address e RSSI. Gli oggetti di questa classe sono gli oggetti che compongono i ViewModel atti a contenere la lista di tutti i dispositivi incontrati.
2. Device: la seguente classe è una rappresentazione dei dispositivi, e come tali hanno solo un attributo: MAC Address.
3. Contact: questa classe definisce il contatto tra il proprio dispositivo e il dispositivo incontrato, esso contiene i dati elaborati e completi: MAC Address del proprio dispositivo, MAC Address del dispositivo rilevato, Distanza, Posizione e Data del contatto.

Le classi definite per la rappresentazione dei dati implementano un'annotazione sopra a essi che permette di definire qual è il parametro corrispondente nel database. In questo modo si è dato uno **standard** per il nome dei parametri, che corrisponde a quelli che il Server accetta e comprende e, di riflesso, corrispondono ai parametri definiti nello Schema del database.

Le annotazioni fanno parte della libreria Gson, proposta da google per la conversioni di oggetti Kotlin in oggetti Json. In questo modo, gli oggetti vengono convertiti e il nome della chiave corrisponderà a quello scritto nelle annotazioni.

Gson

Gson è una **libreria** fornita da Google per la serializzazione e deserializzazione degli oggetti Java (e Kotlin) su JSON. La libreria mappa e converte i dati di una classe in formato JSON, e viceversa. In questo modo, si possono preparare i dati per l'invio al Server, in un formato standard comune ai due software.

3.1.13 Scambio dati con il Server

I dati, una volta elaborati e completati, sono pronti per essere inviati al Server, che li elaborerà e immagazzinerà nel database non relazionale. I contatti che si trovano nel ViewModel '*contactsViewModel*' possono essere intesi come un buffer in cui vi sono i

contatti che ancora debbono essere inviati al Server.

Il MainActivity si occupa quindi di osservare questa lista tramite Observer e, quando vengono aggiunti i dati, inviarli tramite HTTP al Server.

Si è istanziata una classe per la gestione dello scambio dati con il database nel Server remoto: **TracerDB**

```
contactsViewModel.contacts.observe(owner: this, {
    val lastContact = it.removeLast()
    mTracerDB.insertContact(lastContact)
})
```

Figura 3.12: Invio dei contatti al Server.

Il sistema di invio dati al server è stato suddiviso in più classi raggruppate in moduli in modo da aumentare l'astrazione del sistema, con un'architettura a strati.

Strati del sistema per l'invio dei dati al Server:

API

Il modulo API contiene l'interfaccia *ContactTracerDBService*, essa è l'interfaccia usata dall'oggetto **Retrofit**¹¹ che definisce come esso debba comunicare con il Server. Nell'interfaccia si definiscono tutti i metodi HTTP con cui si comunica con il Server per ogni collezione del database con intestazione (*Header*) della richiesta, corpo (*Body*) e le query URL associate alla richiesta.

Nel dettaglio:

1. Il metodo HTTP definisce l'operazione CRUD da eseguire nel Server. Ogni metodo HTTP
2. L'header è l'intestazione della chiamata HTTP. Nel caso del progetto in questione contiene il Token (Stringa) di autorizzazione che permette l'autenticazione al Server.
3. Il corpo della richiesta contiene tutti i dati effettivi, in formato JSON, da inviare.
4. La Query URL è una stringa che viene posta all'interno dell'Url, che il Server può intercettare per prendere come paramentro per le proprie azioni.

Remote Data Source

Il package **datasource** contiene le interfacce e classi che implementano le funzioni per il recupero dei dati da parte delle diverse sorgenti dei dati. Ogni classe estende un'interfaccia definita in precedenza, in modo da incrementare la possibilità di future estensioni. Le classi implementate riguardano per ora solo lo scambio dei dati con il Server remoto in quanto è ciò che si necessita. Le classi implementate sono quindi: **ContactRemoteDataSource** e **DeviceRemoteDataSource**, ognuna per ogni Schema del database. Queste classi, si limitano a eseguire funzioni API per l'ottenimento dei dati dal Server, in formato JSON.

¹¹Client HTTP per lo sviluppo su Android.

```
@POST(DBConfig.contactsEndpoint)
suspend fun insertContact(
    @Header( value: "Authorization") accessToken: String,
    @Body contact: Contact
): Response<Contact>
```

Figura 3.13: Funzione POST implementata nell’interfaccia per la connessione con il DB.

Repository

Il modulo **repository** comprende le interfacce e le classi che si occupano di scambiare i dati tra i possibili tipi di database, remoti e locali, convertire i dati, e restituire gli oggetti necessari in base al tipo di richiesta. Le classi *repository* implementano le classi descritte nei vari package per lo scambio dati, al fine di decisione sul come e dove scambiare i dati tra le varie sorgenti di dati. In Figura 3.14 vi è l’implementazione

```
override suspend fun getAllContact(_deviceId : String): List<Contact>? {
    lateinit var contactList: List<Contact>

    try {
        val response = contactRemoteDataSource.getAllContact(_deviceId)
        val body = response.body()
        if (body != null) {
            contactList = body
            return contactList
        }
    } catch (exception: Exception) {
        Log.i( tag: "Error", exception.message.toString())
    }

    return null
}
```

Figura 3.14: Esempio di funzione per l’ottenimento della lista dei contatti effettuati da un device.

di una funzione della classe *repository* che si occupa dell’ottenimento di una lista di contatti effettuati da un certo dispositivo. Si può notare come la funzione sia asincrona e esegua la funzione corrispondente per lo scambio dei dati con il Server. I dati poi ricevuti verranno convertiti tramite Gson in oggetti Kotlin.

Use cases

Il package **usecases** contiene le classi che implementano letteralmente i casi d’uso relativi all’inserimento e l’ottenimento dei dati. Nel package vi è una classe per ogni

Schema (o *collezione*) del database, e in questo caso sono due: relative agli utenti e ai dispositivi. Le classi contenute nel package sono le classi che vengono istanziate direttamente per la gestione dello scambio dei dati nell'App: esse infatti fanno uso delle classi e interfacce di ogni diverso strato per lo scambio dei dati. In modo da mantenere generale e estendibile l'implementazione, le classi sono indipendenti dal tipo di database implementato, e se esso si trova in locale o in rete.

3.1.14 User Interface



Figura 3.15: User interface dell'App.

L'interfaccia utente in Android viene sviluppata mediante il linguaggio XML, definendo i componenti e le loro proprietà connesse. Android Studio fornisce anche un'ambiente grafico per lo sviluppo dell'interfaccia grafica, chiamato *Layout Editor*, il quale è un'insieme di **componenti** specifici e **relazioni** al codice sorgente dell'App.

Componenti

Le componenti dell’interfaccia utente sono molteplici e adatte ad ogni esigenza dell’App. Esempi di componenti sono: bottoni, testi, immagini, barre di progresso.

Constraints

Lo sviluppo di interfacce utenti responsive, ovvero che si adattano al tipo di schermo e risoluzione del dispositivo, avviene tramite l’uso di vincoli chiamati **constraints**. I *constraints* sono vincoli di un componente che viene ancorato ad un’altro componente o allo schermo (view) direttamente. In questo modo si ha il risultato di un layout che si adatti in base alla risoluzione e non abbia solo delle misure fisse.

Binding

Il Binding dei dati nell’interfaccia non è altro che la visualizzazione dei dati nel Model nell’interfaccia utente. Viene definita la variabile del ViewModel corrispondente e si richiama nella proprietà che si vuol associare nell’UI.

Eventi

Gli eventi dell’interfaccia utente rappresentano le interazioni che ha l’utente con l’App. Infatti, ogni componente dell’ui può registrare un’azione che può essere eseguita a seguito di un determinato evento compiuto dall’utente. L’evento dell’utente può essere ad esempio la pressione di un bottone, lo slide della schermata o la scelta multipla di un elemento in un componente.

Resources

Il package *Resources* contiene tutte le risorse statiche necessarie per la visualizzazione, esse contengono i font, le immagini e le risorse multimediali che contribuiscono alla composizione dell’UI.

Material Design

Il Material design è uno stile, un codice, un linguaggio di design sviluppato da Google per fornire degli strumenti utili per la progettazione e sviluppo dell’interfaccia utente.

3.1.15 Gestione degli errori

La gestione degli errori avviene in due modi: a livello di sviluppo, tramite i costrutti **’try-catch’** e **’throw’**, mentre a livello di testing, tramite le classi di Test in Kotlin. Il primo costrutto a livello di sviluppo, viene usato letteralmente per eseguire una porzione di codice e ‘catturare’ le possibili eccezioni che si potrebbero verificare in certe operazioni: in questo modo si riesce a gestire tutte le eccezioni e risolvere, per restituire un risultato corretto. Il secondo strumento di sviluppo, throw, è una funzione che permette l’esecuzione di un’eccezione che viene sollevata nel metodo che ha chiamato la funzione in cui vi è il throw. Con il throw si riesce a prevenire le eccezioni e il blocco del sistema.

Per quanto riguarda il **testing**, si implementano delle classi che permettono di eseguire dei metodi e aspettarsi un risultato definito, per tutte le possibili casistiche di utilizzo dell'applicazione, in modo da confermare il corretto funzionamento dell'App.

3.2 Server

3.2.1 Librerie

Le librerie utilizzate appartengono al gestore dei pacchetti **npm**, esse costituiscono una parte fondamentale per lo sviluppo del Server in quanto permettono di utilizzare moduli consolidati per funzioni comuni.

Express.js

Express, come descritto il precedenza, è il framework utilizzato per lo sviluppo del Server, che fornisce una serie di funzioni avanzate per le applicazioni web. Express fornisce gli strumenti necessari per la creazione di un Server Web e per la gestione delle chiamate HTTP, formando una solida base per lo sviluppo di **REST** (*Representational State Transfer*) **API** nell'applicativo.

Handlebars

Handlebars è un linguaggio di **templating** che compila il template in funzioni Javascript, e che mantiene la visualizzazione separata dalla logica del business. Handlebars viene utilizzato nel progetto come '**view engine**', ovvero si occupa della visualizzazione dei dati nel sito web. Handlebars permette la divisione in **moduli** (*partials*) del sito web, in modo da favorire il riuso degli stessi in diverse pagine web. Il linguaggio di formattazione è comunemente HTML e CSS, mentre per ciò che riguarda il templating, Handlebars ha un proprio semplice linguaggio, per la gestione dei *partials*, dei moduli **riutilizzabili** e dei dati **dinamici** all'interno della view.

Mongoose

Mongoose è una libreria javascript per la modellazione degli oggetti di MongoDB. La modellazione di mongoose degli oggetti è basata sul concetto degli '**Schema**', e fornisce delle soluzioni che includono: validazione dei dati, conversione del tipo di dati, costruzioni delle query ed altre funzioni per la semplificazione della gestione del database.

Express-jwt

Express-jwt è un pacchetto e libreria npm che fornisce l'implementazione della **validazione** dei JSON Web Tokens per l'accesso alle funzioni ristrette. Questo **middleware** valida il token di autorizzazione attraverso un servizio di validazione, che nel caso di questo progetto è Auth0.

Dotenv

La libreria **dotenv** è una libreria semplice e senza dipendenze che permette di immagazzinare le variabili di ambiente, che si definiscono nel progetto, all'interno di un file di nome '*process.env*'. In questo modo le variabili d'ambiente appartengono a tutto il progetto.

Cors

Cors (*Cross-origin resource sharing*) è un meccanismo che permette la richiesta di risorse **ristrette** in una pagina web da parte di un’altro dominio, diverso da quello dove le risorse sono salvate. In questo modo è possibile la richiesta di risorse da parte dell’App verso il Server. La libreria **cors** contenuta in *npm* si occupa dell’implementazione di questo meccanismo all’interno di Express.js, attraverso l’uso di middlewares.

Librerie di debugging e sviluppo

Vi sono diverse librerie di debugging e sviluppo all’interno del progetto. Una di queste, ad esempio **nodemon**, è una libreria che permette il *restart* del Server quando vengono apportate modifiche al codice.

3.2.2 Server Web

Il Server Web viene implementato in Node.js e messo in ascolto nella porta *3000* attraverso due moduli fondamentali.

Il **primo modulo** comprende l’istanza dell’oggetto express ”*app*”, la quale viene successivamente inizializzato e configurato con le varie classi che costituiranno l’applicativo. Express permette la configurazione dell’oggetto tramite la funzione *use()*. In particola-



```
● ● ●
export default async ({ _app, _projectPath }) => {
  _app.use(cors());
  _app.use(logger("dev"));
  _app.use(express.json());
  _app.use(express.static(path.join(_projectPath, "/views/public")));
};
```

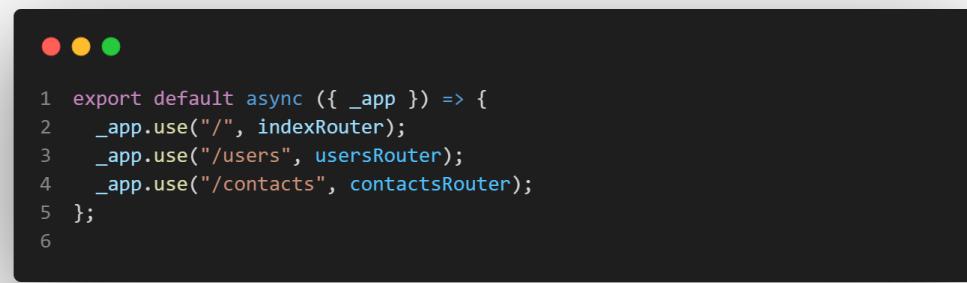
Figura 3.16: Configurazione Express

re, in prima istanza, l’oggetto viene configurato per tutto ciò che riguarda il framework express, con alcune delle librerie descritte: **cors**, **morgan**, il supporto per il formato **JSON** e con il percorso dei file statici della view.

L’oggetto, una volta indicati i moduli da utilizzare per le funzioni relative alla configurazione Web, viene impostato con le *routes* da utilizzare per ricevere le chiamate HTTP negli **end-point** definiti, relativi alla pagina web e alle REST API. Una volta impostate le **routes** del Server, esso sarà configurato per intercettare chiamate negli end-point definiti, tramite url, e restituire un risultato in base al tipo di richiesta.

Viene impostato poi tutto ciò che riguarda la ’view’, Express viene configurato per utilizzare Handlebars, vengono impostate le directory su cui sono salvati i file relativi alla view, e in quest’ultimo vengono registrati i componenti riutilizzabili appartenenti alla view, tramite la funzione *registerPartials()*.

Come ultima istanza, Express viene configurato per la gestione degli **errori** HTTP:



```
● ● ●
1 export default async ({ _app }) => {
2   _app.use("/", indexRouter);
3   _app.use("/users", usersRouter);
4   _app.use("/contacts", contactsRouter);
5 };
6
```

Figura 3.17: Configurazione Routes

tramite la medesima funzione `use()`, relativa all'oggetto Express, si istanziano le funzioni che intercettano gli errori HTTP con il relativo codice di errore, e si mostra la pagina di errore con una specifica descrizione in base al tipo di errore.

3.2.3 Express

Express.js, come descritto in precedenza, è un framework ideato per la creazione e il mantenimento dei Server. Esso fornisce strumenti e metodi per la **gestione** del Server, e implementa anche moduli estensivi di terze parti che forniscono funzionalità, sicurezza e velocità, in aggiunta a ciò che è già compreso nella **suite** di Express.

3.2.4 View



Figura 3.18: Pagina web di visualizzazione dati

Handlebars

Handlebars è il motore grafico che si occupa della visualizzazione delle pagine web scritte in **HTML e CSS**. Esso organizza il progetto della *view* in modo strutturato, ogni parte della view fa parte di un preciso package. I file statici vengono immagazzinati in un package di nome '*Public*', le componenti della view hanno il loro package in '*partials*', così come il layout e le pagine web della view sono separate dagli altri package.

Partials

I partials, in Handlebars, rappresentano i **componenti** di template che è possibile **riutilizzare** nelle pagine web. Nel progetto, data la semplicità della visualizzazione, è stato creato un solo partial, che rappresenta il *footer* nelle pagine web. Questo modo di strutturare la view ci permette di scrivere codice generico, dinamico in base ai dati che si necessita in una determinata pagina web, che è possibile riutilizzare e soprattutto modificare: in questa maniera, la modifica si applicherà a tutte le pagine della view su cui è dichiarato quel determinato partial.

Formattazione

Handlebars esegue il render della view tramite pagine scritte negli ormai noti HTML e fogli di stile CSS.

Bootstrap è il framework front-end CSS che si è utilizzato per le pagine web del progetto. Bootstrap fornisce strumenti consolidati per la realizzazione di pagine web *responsive*¹², un modello di progettazione per le componenti HTML e CSS della view.

3.2.5 Database

Il Database utilizzato nel Server per mantenere i dati è un database **non relazionale**, impostato a **documenti** in stile JSON, con tipi di schema (*le tradizioni tabelle*) dinamico. MongoDB è il DBMS¹³ (*Database Management System*) usato nel caso di questo progetto. Si è scelto MongoDB in quanto le applicazioni di database non relazionali in ambiti con grandi quantità di dati sono risultate più efficaci rispetto al tradizionale SQL.

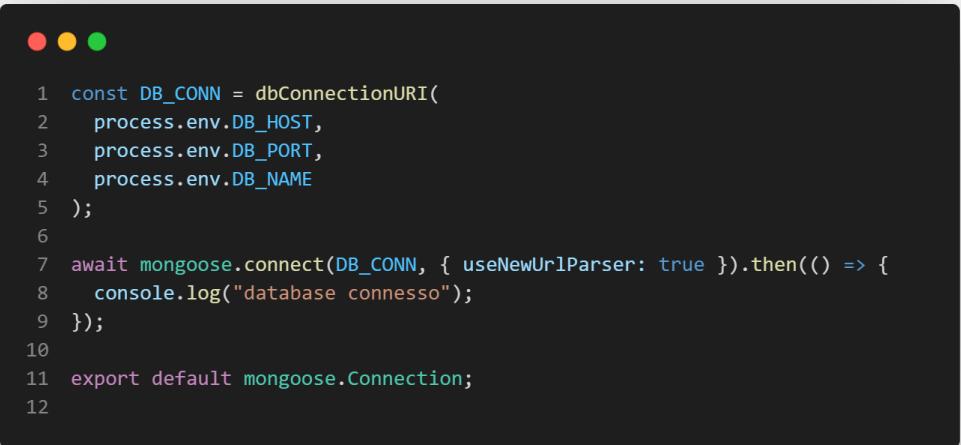
MongoDB è eseguito dal sistema operativo come **servizio di rete** con esecuzione continua e si occupa di gestire i documenti, i dati e le interrogazioni sul database. Esso si integra con Node.js attraverso **mongoose**, una libreria di strumenti di modellazione degli oggetti per MongoDB.

Connessione a MongoDB

La connessione a MongoDB, tramite **mongoose**, avviene con l'ausilio di una modulo **helper**, che provvede alla connessione al DBMS e restituisce un'oggetto corrispondente alla connessione effettuata. La funzione di connessione invoca una funzione di *callback* a connessione eseguita correttamente, che in questo corrisponde alla stampa a video di una stringa di log.

¹²Tecnica di web design per la realizzazione di siti in grado di adattarsi graficamente in modo automatico al dispositivo coi quali vengono visualizzati.

¹³Sistema software progettato per consentire la creazione, la manipolazione e l'interrogazione efficiente di database.



```

● ● ●
1 const DB_CONN = dbConnectionURI(
2   process.env.DB_HOST,
3   process.env.DB_PORT,
4   process.env.DB_NAME
5 );
6
7 await mongoose.connect(DB_CONN, { useNewUrlParser: true }).then(() => {
8   console.log("database connesso");
9 });
10
11 export default mongoose.Connection;
12

```

Figura 3.19: Connessione al DBMS MongoDB

I parametri corrispondenti alla composizione dell'**URI** alla posizione del database nella rete, viene recuperato dalle variabili di ambiente precedentemente impostati su Node.js. La funzione `dbConnectionURI()` provvede ad assemblare l'URI da utilizzare nella connessione.

Schema

Uno **Schema** è un oggetto che definisce la forma e il contenuto dei documenti e dei documenti incorporati in una collezione.

Il database utilizzato ha una struttura di due collezioni:

La prima collezione definisce e contiene i dati degli utenti che hanno eseguito delle rilevazioni o che sono stati rilevati, infatti la politica di gestione del database prevede che il server, quando riceve i dati di una rilevazione di contatto, registri il contatto nei corrispondenti utenti, e se questi non sono già esistenti nel database, si provvede a crearne di nuovi.

Schema User

ID	Stringa
Data di registrazione	Data

Tabella 3.1: User Schema del Database

La seconda collezione identifica i dati dei contatti avvenuti, ricevuti dai dispositivi mobili, e contiene i dati per l'identificazione dei corrispondenti utenti.

Schema Contact:

Ogni Schema è un oggetto della classe Schema di mongoose, che corrisponde a una collezione di MongoDB e viene aggiunto come *model* nell'oggetto mongoose. Le operazioni sul database si eseguono direttamente con le funzioni dell'oggetto Schema.

ID Contatto	Stringa
ID Dispositivo rilevatore	Stringa
ID Dispositivo rilevato	Stringa
Distanza	Numero
Posizione	Stringa
Data del contatto	Data

Tabella 3.2: Contact Schema del Database

3.2.6 Routes API



```

1 // READ
2 router.get("/", async (req, res) => {
3   if (req.query.id) res.send(await getUserService(req.query.id));
4   else res.send(await getAllUsersService());
5 });
6 // CREATE
7 router.post("/", postUserController, async (req, res) => {
8   res.send(await postUserService(res.userInfo));
9 });
10 // UPDATE
11 router.patch("/", patchUserController, async (req, res) => {
12   res.send(await patchUserService(res.id, res.update));
13 });
14 // DELETE
15 router.delete("/", async (req, res) => {
16   res.send(await deleteUserService(req.query.id));
17 });
18

```

Figura 3.20: Implementazione delle operazioni CRUD per gli utenti

Express fornisce una classe per la gestione delle **routes** in Node.js, le routes permettono al server di gestire le chiamate HTTP e di restituire un oggetto negli specifici end-point.

L'oggetto **router** ha diverse funzioni per la gestione dei metodi HTTP, e quelle utilizzate per l'implementazione delle operazioni CRUD¹⁴ nel progetto sono: **POST**, **GET**, **PATCH**, **DELETE**.

Le funzioni scritte, si compongono da due parametri fondamentali.

Il primo, di tipo Stringa, definisce l'end-point su cui il server intercetta la richiesta. Esso, banalmente, rappresenta l'URL in cui il Server risponde.

Il secondo parametro definisce invece l'azione e le operazioni compiute dal Server quando si raggiunge il determinato end-point. Esso, quindi, è una funzione di callback

¹⁴CRUD

che a sua volta richiede almeno due parametri:

1. Request (o *req*), un oggetto che comprende i parametri della richiesta HTTP, come le query nell'URL o i dati nel body.
2. Response (o *res*), un oggetto che contiene i dati di risposta dalle operazioni effettuate nella route, create dal Server.

Un terzo elemento delle funzioni CRUD è facoltativo e riguarda come le funzioni possono essere inserite nei parametri come funzioni di callback: i **middleware**. I Middleware sono particolari funzioni che si pongono nel '**mezzo**' di una funzione, per controllare un flusso di dati e applicare altre funzioni.

Nel progetto, i middleware vengono applicati alle funzioni CRUD e usati come *controller* dei dati in ingresso.

```

● ● ●

1 // READ
2 router.get("/", async (req, res) => {
3   if (req.query.id) res.send(await getContactService(req.query.id));
4   else if (req.query.deviceId)
5     res.send(await getContactsForDevice(req.query.deviceId));
6   else res.send(await getAllContactsService());
7 });
8 // CREATE
9 router.post("/", postContactController, async (req, res) => {
10   res.send(await postContactService(res.contactInfo));
11 });
12 // PATCH
13 router.patch("/", patchContactController, async (req, res) => {
14   res.send(await patchContactService(res.id, res.update));
15 });
16 // DELETE
17 router.delete("/", async (req, res) => {
18   res.send(await deleteContactService(req.query.id));
19 });

```

Figura 3.21: Implementazione delle operazioni CRUD dei Contatti

3.2.7 Controller

I Controller vengono implementati nelle operazioni sia relative agli utenti che ai contatti, in questo modo, secondo il pattern architettonale MVC, si **divide** la parte del **controllo** e del **filtraggio** dei dati in ingresso rispetto alla logica di business. Si ha un **controllo** completo sulle informazioni che la route riceve in ingresso, aperto a possibili estensioni future, e **riutilizzabile**.

In particolare, i controller mappano i dati che sono necessari alla route su oggetti che il Server **comprende** e riesce ad elaborare, aggiunti poi all'oggetto Response della route.

Esempio di controller degli Utenti nella Figura 3.22.

Esempio di controller dei Contatti nella Figura 3.23.

```

1  async function postUserController(req, res, next) {
2    try {
3      res.userInfo = userInfoConverter(req.body._id);
4    } catch (err) {
5      res.code = err.code;
6    }
7    next();
8  }
9
10 async function patchUserController(req, res, next) {
11   res.id = req.query.id;
12   res.update = req.body;
13   next();
14 }

```

Figura 3.22: Controller dei dati in ingresso per le operazioni degli utenti

```

1  async function postContactController(req, res, next) {
2    res.contactInfo = {
3      deviceDetector: req.body.deviceDetector,
4      deviceDetected: req.body.deviceDetected,
5      distance: req.body.distance,
6      contactDate: req.body.contactDate,
7    };
8    next();
9  }
10 async function patchContactController(req, res, next) {
11   res.id = req.query.id;
12   res.update = req.body;
13   next();
14 }

```

Figura 3.23: Controller dei dati in ingresso per le operazioni dei contatti

3.2.8 Service

I moduli javascript **Service** comprendono la logica del business relative alle operazioni CRUD, essi si occupano di ricevere i dati, elaborati dai controller, ed eseguire le operazioni per l'elaborazione degli stessi, tra cui le operazioni nel database. I Service, oltre ad eseguire le operazioni in modo asincrono, comprendono una gestione degli **errori** relativi alla **comunicazione** con il database.

I moduli Service rispecchiano le operazioni CRUD del database, e per ogni operazione

implementa la politica di gestione del database.

In particolare, le politiche per la gestione degli **utenti** implementate:

1. **POST**: Relativamente all'operazione di creazione, se l'utente già esiste, non viene creato un nuovo utente uguale.
2. **PATCH**: Per l'operazione di aggiornamento di un'informazione, se un determinato utente specificato non esiste nella collezione, nessun utente viene aggiornato.
3. **DELETE**: Analogamente all'operazione di aggiornamento, nella funzione di eliminazione di un record, se l'utente specificato non esiste nella collezione, nessun utente viene eliminato.

Le politiche per la gestione dei **contatti** implementate:

1. **POST**: Per ciò che concerne l'operazione di creazione, se gli utenti coinvolti nella rilevazione non esistono nella collezione, essi vengono creati nella collezione degli utenti.
2. **PATCH**: Relativamente all'operazione di aggiornamento di un'informazione, se un determinato contatto specificato non esiste nella collezione, nessun contatto viene aggiornato.
3. **DELETE**: Analogamente all'operazione di aggiornamento, nella funzione di eliminazione di un record, se il contatto specificato non esiste nella collezione, nessun contatto viene eliminato.

Esempio di funzione POST nel modulo Service dei Contatti nella Figura 3.24.



```
1  async function postContactService(contactInfo) {
2    if (!(await userExists(contactInfo.deviceDetector)))
3      postUserService(userInfoConverter(contactInfo.deviceDetector));
4
5    if (!(await userExists(contactInfo.deviceDetected)))
6      postUserService(userInfoConverter(contactInfo.deviceDetected));
7
8    let newContact;
9    try {
10      newContact = await postContactRepository(contactInfo);
11    } catch (err) {
12      console.error(err);
13      return { message: err.message };
14    }
15    return newContact;
16 }
```

Figura 3.24: Business logic relativa all'operazione POST dei contatti

3.2.9 Repository

I moduli Repository si riferiscono direttamente agli **oggetti** per la **comunicazione** con il database tramite mongoose. Le funzioni relative al Repository utilizzano mongoose per comunicazione con il database, le azioni nel database si eseguono attraverso l'oggetto Schema, e le sue funzioni associate. Esempio di operazione POST relativa agli Utenti nella Figura 3.25: si crea un nuovo oggetto dalla classe Schema 'User' con le informazioni in input nella funzione; in modo asincrono, attraverso la funzione *save()* l'oggetto viene salvato nel database.



```

1  async function postUserRepository(userInfo) {
2    const user = new User({
3      _id: userInfo._id,
4      registerDate: userInfo.registerDate,
5    });
6    return await user.save();
7  }

```

Figura 3.25: Operazione POST nel database attraverso il repository.

3.2.10 JWT

Il progetto implementa l'utilizzo dei **JSON Web Token**, uno standard per creare una firma che collega due parti: **client** e **server**. I JSON Web Token sono dei token che vengono utilizzati dal client per dimostrare l'**autenticazione** al Server, il quale provvede a **verificarli**. I token sono delle stringhe di chiavi private che vengono generate dal Server.

Il progetto include un sistema di controllo dei token per gli end-point che lo necessitano tramite il sistema di gestione dei token **Auth0**¹⁵. In questo modo, le API del Server possono essere protette da un sistema di autenticazione. Nella Figura 3.26 è rappresentata la funzione helper per la validazione dei token.

3.2.11 Routes di visualizzazione

Le routes di visualizzazione utilizzano il medesimo oggetto *router* utilizzato per gli end-point API, con la funzione associata *get()*. Queste routes, utilizzano una funzione di rendering nell'oggetto *res*: la funzione *render()*. La funzione di rendering esegue il render e la visualizzazione della pagina impostata come parametro, all'interno della cartella *views* e passa alla pagina i parametri che indichiamo. Ad esempio, nella figura 3.27, il parametro passato alla pagina è riferito al titolo della stessa.

¹⁵Sistema esterno di autenticazione HTTP. Vedi <https://auth0.com/>

```
● ● ●

1 const jwToken = jwt({
2   secret: jwksRsa.expressJwtSecret({
3     cache: true,
4     rateLimit: true,
5     jwksRequestsPerMinute: 5,
6     jwksUri: `https://contacttracer.eu.auth0.com/.well-known/jwks.json`,
7   }),
8
9   // Validate the audience and the issuer.
10  audience: "https://contacttracer-api/",
11  issuer: `https://contacttracer.eu.auth0.com/`,
12  algorithms: ["RS256"],
13 });


```

Figura 3.26: Funzione di validazione token.

```
● ● ●

1 router.get("/", function (req, res, next) {
2   res.render("index", {
3     title: "Contact Tracer",
4   });
5 });
6
7 router.get("/mycontacts", function (req, res, next) {
8   res.render("user/mycontacts", {
9     title: "Contact Tracer",
10  });
11 });


```

Figura 3.27: Routes per la visualizzazione delle pagine web.

3.2.12 Visualizzazione dei dati

La visualizzazione dei dati viene implementata attraverso una pagina web di riferimento, che provvede, per l'utente collegato, a **recuperare** i propri dati sui contatti avvenuti, e visualizzarli in una tabella numerata con gli attributi del contatto. Un esempio di visualizzazione è mostrato nella Figura 3.28, dove però la posizione è stata disabilitata dalle rilevazioni nell'App di riferimento.

HOME				
#	ID	Data	Distanza	Posizione
1	f48ffd8bfe743f831583efd0fc6df2dca77aco9d860367288b1f8c1775a5e71	2021-10-15T21:39:12.000Z	0.0707945784384138m	Non rilevata
2	60278e8563f9f11fdab74ec8abcf85ded648b56ba3e4e1634bc15fe35504ec49	2021-10-15T21:40:03.000Z	0.31622776601683794m	Non rilevata
3	f48ffd8bfe743f831583efd0fc6df2dca77aco9d860367288b1f8c1775a5e71	2021-10-15T21:40:03.000Z	0.06309573444801933m	Non rilevata
4	b3c72dcc4eb4a9a1d80e4263c8e70d45d3426f41c9be69e8ec0fbbaa87a096e	2021-10-15T21:40:08.000Z	0.06309573444801933m	Non rilevata
5	4871dd8060e8999087e8f51b28ea8b7c327fd09c0475ea003456a0a464fdec84	2021-10-15T21:40:11.000Z	0.01778279410038923m	Non rilevata
6	4871dd8060e8999087e8f51b28ea8b7c327fd09c0475ea003456a0a464fdec84	2021-10-15T21:40:11.000Z	0.1m	Non rilevata
7	4871dd8060e8999087e8f51b28ea8b7c327fd09c0475ea003456a0a464fdec84	2021-10-15T21:40:12.000Z	0.0199526231496888m	Non rilevata
8	4871dd8060e8999087e8f51b28ea8b7c327fd09c0475ea003456a0a464fdec84	2021-10-15T21:40:13.000Z	0.01412537544622754m	Non rilevata
9	4871dd8060e8999087e8f51b28ea8b7c327fd09c0475ea003456a0a464fdec84	2021-10-15T21:40:21.000Z	0.015848931924611134m	Non rilevata

Figura 3.28: Esempio di pagina di visualizzazione dei contatti di un determinato utente.

Ajax

Ajax (*Asynchronous JavaScript and XML*) è una tecnica di sviluppo che combina javascript e l'oggetto *XMLHttpRequest* implementato nel browser. XMLHttpRequest è un oggetto che contiene un **set di API** per l'interazione con il server. Ajax permette l'aggiornamento *asincrono* delle pagine web, recuperando i dati da un Web Server: infatti, Ajax fa uso di oggetti JSON per la comunicazione con le API del Server che si è implementato, in questo modo, si ricevono dati in forma JSON che possono essere elaborati e aggiunti alla pagina web.

Nella funzione riportata in Figura 3.29, la pagina web, esegue una richiesta nell'end-point dichiarato nell'url tramite javascript. La funzione, invoca poi una funzione di callback - quando la richiesta viene soddisfatta con successo - che contiene l'oggetto **data**, dove sono presenti i dati che il Server ha restituito rispetto alla chiamata ajax.

Nella tabella già presente nella pagina web, con id '*contactsTable*' vengono create le righe rispetto ai dati del database.

jQuery

Le funzioni ajax sono implementate tramite l'utilizzo di **jQuery**, una libreria standard di javascript sviluppata per la semplificazione dell'implementazione di applicazioni web, dei dati dinamici in esso, degli eventi e della modifica degli elementi in HTML.



```

1 $.ajax({
2     url: "/contacts/",
3     data: "",
4     type: "GET",
5     dataType: "json",
6     contentType: "application/json; charset=utf-8",
7     success: function (data) {
8         var trHTML = "";
9         var count = 1;
10        $.each(data, function (key, value) {
11            trHTML +=
12                "<tr><th scope='row'>" +
13                count +
14                "</th><td>" +
15                value.deviceDetected +
16                "</td><td>" +
17                value.contactDate +
18                "</td><td>" +
19                value.distance +
20                " m" +
21                "</td><td>" +
22                value.position +
23                "</td></tr>";
24            count++;
25        });
26        $("#contactsTable").append(trHTML);
27    },
28    error: function (XMLHttpRequest, textStatus, errorThrown) {},
29 });
30 });

```

Figura 3.29: Funzione AJAX della pagina web di visualizzazione dati.

3.2.13 User Interface

La descrizione di come i dati vengono rappresentati nell’interfaccia utente è trattata perciò nei paragrafi precedenti, motivando l’utilizzo degli strumenti necessari al compito.

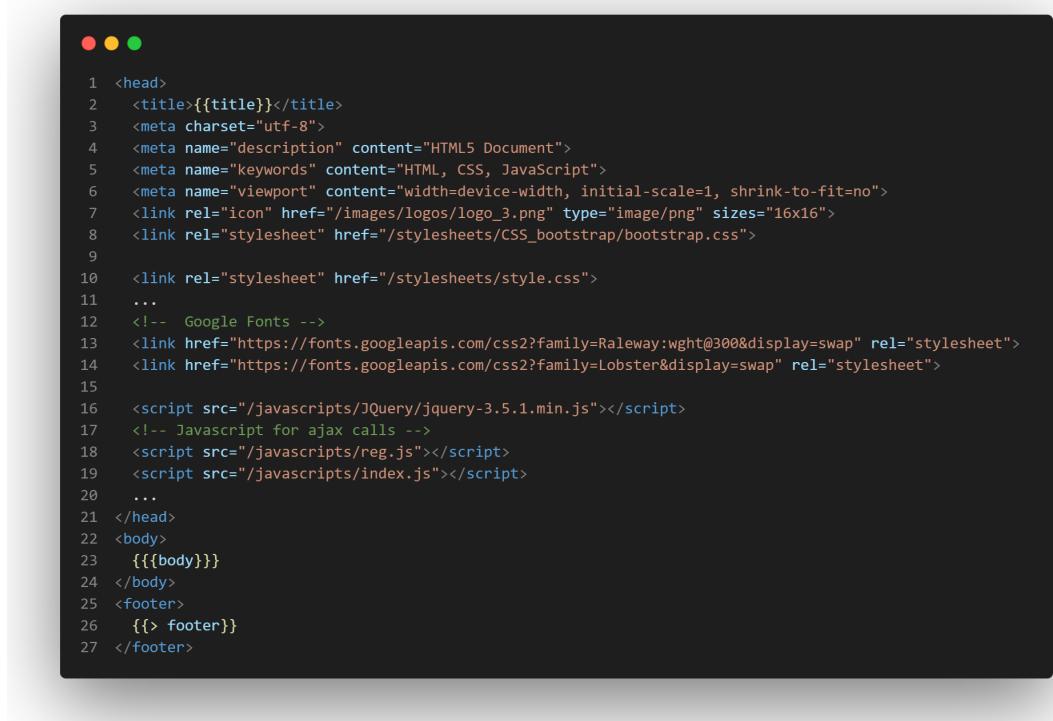
L’interfaccia grafica che l’utente utilizza per visualizzare i dati e per dare input al Server è realizzata mediante la formattazione in HTML e CSS.

Handlebars permette la creazione di **layout** HTML, documenti che corrispondono alla struttura di tutte le pagine web a cui viene applicato. In questo caso, un foglio layout viene applicato a tutte le pagine. Il layout utilizzato contiene:

- La parte **header** delle pagine web, con *meta tag*, le relazioni ai documenti esterni quali *fogli di stile*, *icona*, *fonts*, e i moduli javascript per lo *scripting*.
- La parte **body**, con il riferimento di handlebars al body, il quale è variabile per ogni pagina web.

- La parte **footer**, con il riferimento di handlebars al footer, il quale è un componente *partial* che cambia in base ai dati che riceve.

Nella prossima Figura, la 3.30, si mostra una parte di layout utilizzato per il progetto.



```

1 <head>
2   <title>{{title}}</title>
3   <meta charset="utf-8">
4   <meta name="description" content="HTML5 Document">
5   <meta name="keywords" content="HTML, CSS, JavaScript">
6   <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
7   <link rel="icon" href="/images/logos/logo_3.png" type="image/png" sizes="16x16">
8   <link rel="stylesheet" href="/stylesheets/CSS_bootstrap/bootstrap.css">
9
10  <link rel="stylesheet" href="/stylesheets/style.css">
11 ...
12 <!-- Google Fonts -->
13 <link href="https://fonts.googleapis.com/css2?family=Raleway:wght@300&display=swap" rel="stylesheet">
14 <link href="https://fonts.googleapis.com/css2?family=Lobster&display=swap" rel="stylesheet">
15
16 <script src="/javascripts/JQuery/jquery-3.5.1.min.js"></script>
17 <!-- Javascript for ajax calls -->
18 <script src="/javascripts/reg.js"></script>
19 <script src="/javascripts/index.js"></script>
20 ...
21 </head>
22 <body>
23   {{body}}
24 </body>
25 <footer>
26   {{> footer}}
27 </footer>

```

Figura 3.30: Layout HTML utilizzato per Handlebars.

Handlebars

Handlebars utilizza il formato *'.hbs'* per contenere il codice HTML e il proprio linguaggio di scripting. Il linguaggio di Handlebars, semplice e di poche funzioni, è creato per favorire il templating, perciò implementa le funzioni per il collegamento tra le componenti della pagina web e per la logica di visualizzazione delle pagine tramite i dati che riceve dal Server. Ad esempio, tramite Handlebars, si può decidere di visualizzare un componente in una pagina web rispetto ad un altro basandosi sul tipo di accesso che ha l'utente, o ad esempio se esso ha eseguito il login nel Server.

Bootstrap

Bootstrap viene implementato importando le librerie CSS e JS (Javascript) all'interno del progetto. Bootstrap ha un proprio work-flow e un modo di organizzare le pagine web, infatti usa il sistema di griglia per definire il posizionamento delle componenti. Questo fa sì che il layout sia responsive e si adatti a qualunque tipo di grandezza dello schermo.

Per la parte responsive, Bootstrap utilizza delle *media-query*, che sono particolari costrutti che definiscono la grandezza dello schermo per la quale certi stili CSS vengono applicati.

Javascript

La parte di scripting relativa al front-end¹⁶ viene implementata attraverso moduli javascript, che coprono la connessione al database, la validazione dei dati e le animazioni della pagina web. La connessione al database, come descritta nei paragrafi precedenti, viene implementata attraverso l'uso di jQuery e Ajax.

Le animazioni nelle pagine web vengono implementate con jQuery. jQuery fornisce gli strumenti per lo sviluppo di applicazioni web in modo da semplificare la gestione del DOM¹⁷ (*Document Object Model*). Nello specifico, nella pagina web viene implementata un'animazione che riguarda il comportamento della barra di navigazione, la quale cambia il colore di sfondo quando si naviga verso il basso nella pagina.



```

1 $(document).ready(function () {
2   $(window).scroll(function () {
3     if ($("#nav").offset().top > 100) {
4       $("#nav").addClass("navbar-shrink");
5     } else {
6       $("#nav").removeClass("navbar-shrink");
7     }
8   });
9
10  $(".navbar-nav>li>a").on("click", function () {
11    $(".navbar-collapse").collapse("hide");
12  });
13 });

```

Figura 3.31: Codice jQuery per l'animazione della Navigation Bar.

3.2.14 Gestione degli errori

La gestione degli errori nel Server si divide in due importanti concetti: la gestione degli errori relativi ai **metodi HTTP** e la gestione degli errori di programmazione.

La gestione degli errori HTTP viene delegata a Express, esso intercetta gli errori nelle chiamate HTTP e restituisce una pagina di errore in base al tipo di errore che intercetta. In Figura 3.32 vi è l'implementazione delle funzioni che Express userà per la gestione degli errori HTTP.

Per quanto riguarda gli errori di programmazione, si usano i costrutti di programmazione forniti da javascript. I costrutti sono: il blocco **try-catch** ed il **throw**. Il primo blocco, viene usato letteralmente per eseguire una porzione di codice e 'catturare' le possibili eccezioni che si potrebbero verificare in certe operazioni: in questo modo si riesce a gestire tutte le eccezioni e risolvere, per restituire un risultato corretto.

Il secondo strumento, **throw**, è una funzione che permette l'esecuzione di un'eccezione

¹⁶Interfaccia grafica per l'utente

¹⁷Forma di rappresentazione dei documenti strutturati come modello orientato agli oggetti.



```

1 _app.use(function (req, res, next) {
2   next(createError(404));
3 });
4
5 /** Error handler */
6 _app.use(function (err, req, res, next) {
7   // set locals, only providing error in development
8   res.locals.message = err.message;
9   res.locals.error = req.app.get("env") === "development" ? err : {};
10
11  // render the error page
12  res.status(err.status || 500);
13  res.render("error", { layout: false });
14 });

```

Figura 3.32: Gestione degli errori HTTP.

che viene sollevata nel metodo che ha chiamato la funzione in cui vi è il *throw*. Con il *throw* si riesce a prevenire le eccezioni e il blocco del sistema.

3.2.15 Autenticazione

L'applicativo supporta e implementa un'autenticazione per eseguire le operazioni nel Server. Il sistema di autenticazione è il precedentemente descritto **Auth0**.

Auth0 è un servizio di autenticazione che fornisce una soluzione di autenticazione e autorizzazione per le applicazioni web, esso provvede a un'autenticazione sicura e permette la creazione delle API in modo autorizzato.

Auth0, come visto nei paragrafi precedenti relativi ai JSON Web Tokens, viene impostato come servizio per l'autenticazione e il Server fa riferimento ad esso per la validazione dei token per l'accesso alle funzioni CRUD dell'applicazione.

L'impostazione di Auth0 prevede la registrazione al sito web <https://auth0.com/>, e successivamente la creazione delle API che si interfacciano con l'applicazione creata.

Conclusioni e Sviluppi Futuri

La realizzazione di questo progetto di Tesi di Laurea mi ha formato ulteriormente sulle nuove tecnologie per lo sviluppo di applicazioni per dispositivi mobili, web, server e database. Il progetto copre diversi ambiti dell'Informatica, infatti, i corsi seguiti e sostenuti nel corso degli studi sono stati di notevole importanza e valore per l'acquisizione delle conoscenze necessarie per lo sviluppo dello stesso.

Nel progetto vengono trattati i seguenti temi affrontati all'**Università degli Studi di Camerino**: *Ingegneria del software, Programmazione ad oggetti, Pattern vari (quali MVC, MVVM, Observer, Factory), Strutture dati, Basi di dati, Internet reti e sicurezza, Networking, Sviluppo di applicazioni web, Sviluppo Server, Diritto (per quanto riguarda la parte relativa alla privacy).*

In conclusione, il progetto si è rivelato essere una sfida personale oltre che accademica e perciò mi ritengo soddisfatto del lavoro che è stato svolto all'interno dell'Università di Camerino e della formazione professionale che ho ricevuto dai professori e dai corsi accademici.

Il **codice sorgente** dell'intero progetto verrà fornito su richiesta.

I miei più cari ringraziamenti vanno in particolar modo al professor **Rosario Culmone** per avermi seguito e guidato con le sue indicazioni, suggerimenti e per il supporto che ha dimostrato nel percorso della Tesi e della Laurea come Relatore.

Bibliografia

- [Boj18] Valentin Bojinov. *RESTful web API design with Node.js 10: learn to create robust RESTful web services with Node.js, MongoDB, and Express.js.* 3^a ed. Packt Publishing, 2018.
- [Cen] Embedded Centric. *Introduction to Bluetooth Low Energy 5.* URL: <https://embeddeditcentric.com/introduction-to-bluetooth-low-energy-bluetooth-5/>.
- [Dev] DevDocs. *Express.js Documentation.* URL: <https://devdocs.io/express/>.
- [Fou] OpenJS Foundation. *Node.js Documentation.* URL: <https://nodejs.org/it/docs/>.
- [Goo] Google. *Android Studio Documentation.* URL: <https://developer.android.com/docs>.
- [Her20] David Herron. *Node.js Web Development: Server-side web development made easy with Node 14 using practical examples.* 5^a ed. Birmingham, UK: Packt Publishing, 2020.
- [Jet] JetBrains. *Kotlin Documentation.* URL: <https://kotlinlang.org/docs/home.html>.
- [Mar18] Robert Cecil. Martin. *Clean code.* 1^a ed. Apogeo, 2018.
- [Mon] Inc. MongoDB. *MongoDB Documentation.* URL: <https://docs.mongodb.com/>.