



# **Università Politecnica delle Marche**

---

Corso in Ingegneria Informatica e dell'Automazione

Relazione del corso di Ingegneria del Software e Tecniche per  
l'Informatica Distribuita

Progetto: IoT  
**FOR EMERGENCY MANAGEMENT**

**Studenti:**

Federico Simonetti

Simone Accattoli

Niccolò Marini

# SOMMARIO

<b>1.1 - Motivazioni</b>	<b>3</b>
<b>1.2 - Presentazione del Problema</b>	<b>3</b>
<b>1.3 - Obiettivi</b>	<b>4</b>
<b>2.1 - Beacon</b>	<b>5</b>
<b>2.2 - Android</b>	<b>5</b>
2.2.1 - Ciclo di vita di un'Activity	6
<b>2.3 - Servizi Rest</b>	<b>7</b>
<b>2.4 - Glassfish</b>	<b>8</b>
<b>3.1 - Analisi dei requisiti con Tropos</b>	<b>9</b>
<b>3.2 - Storie Utente e package</b>	<b>9</b>
<b>3.3 - Project Scooping</b>	<b>10</b>
<b>3.4 - User Story</b>	<b>12</b>
<b>3.5 - Analisi CRC</b>	<b>13</b>
<b>3.6 - Architettura</b>	<b>14</b>
<b>4.1 - Classe WelcomeActivity</b>	<b>17</b>
<b>4.2 - Implementazione Server Rest</b>	<b>18</b>
<b>4.3 - Beacon</b>	<b>19</b>
4.3.1 - Posizionamento dei Beacon	19
4.3.2 - Funzione Scan	20
4.3.3 - Macchina a Stati	21
4.3.3.1 - Librerie Android	21
4.3.3.2 - Sincronizzazione	22
4.3.3.3 - BeaconScanner	23
4.3.3.4 - BeaconConnection	25
4.3.4 - Interruzione della connessione	27
<b>5.1 - Problemi legati al Server Rest</b>	<b>29</b>
<b>5.2 - Server emergenze</b>	<b>30</b>
<b>5.3 - Soluzione lato Requester</b>	<b>30</b>
<b>6.1 - Tipologie di utenti</b>	<b>32</b>
<b>6.2 - Iscrizione</b>	<b>33</b>
<b>6.3 - Login</b>	<b>34</b>

<b>6.4 - Funzionalità</b>	<b>34</b>
<b>Appendice</b>	<b>38</b>
<b>A - MANUALE DI FUNZIONAMENTO</b>	<b>38</b>
A.1- Installazione dei tools e importazione del progetto	38
A.2 - Configurazione ed avvio del server	41
A.3 - Avvio database	45
A.3 - Configurazione applicazione	46
A.4 - Manuale di utilizzo e navigazione dell'applicazione	48
A.5 -Gestione delle emergenze	52

# 1 - Introduzione

La sicurezza è un campo molto ampio e sensibile che riguarda ogni individuo all'interno di una comunità. Essa non ha una definizione ben precisa ma si potrebbe definire come la consapevolezza che una certa situazione di pericolo, generata da un qualsiasi evento, non arrechi danni alle persone presenti. I campi che necessitano di sicurezza sono numerosi, considerando che ogni aspetto della vita moderna ha delle implicazioni riguardanti la sicurezza. Nel tempo, vista la grande mole di eventi che possono generare situazioni di pericolo, sono state sviluppate molte tecniche per la gestione di un'emergenza. L'uso di sistemi informatici come strumento di prevenzione, per le emergenze, è molto diffuso sia per minacce legate al settore informatico stesso che ad altri possibili pericoli cui una persona può incontrare quotidianamente. Lo scenario in cui ci si pone nel seguente progetto è utilizzare un sistema informatico per la sicurezza legata a persone poste in un edificio pubblico.

## 1.1 - Motivazioni

Il progetto nasce dall'esigenza di un sistema che aiuti gli utenti in caso di emergenza. Questo sistema si inserisce nell'ambito della gestione delle emergenze all'interno di edifici pubblici, in particolare vuole fornire un aiuto in fase di evacuazione e soccorso mediante una piattaforma di sensori.

Tale sistema nasce con l'idea di fornire informazioni utili ai soccorritori, come ad esempio l'identificazione di una persona ancora intrappolata nella struttura durante una situazione di pericolo.

## 1.2 - Presentazione del Problema

L'edificio pubblico preso in considerazione come scenario è la struttura di Ingegneria dell'Università Politecnica delle Marche. Ovviamente questa scelta è solamente legata allo sviluppo e progettazione del software in una prima fase, poiché in seguito esso deve essere capace di lavorare con molteplici strutture pubbliche. All'interno dell'edificio, gli utenti possono interagire con degli strumenti, chiamati *sensortag*, attraverso l'applicazione. Con questo sistema è possibile osservare la planimetria dell'edificio per orientarsi e ricevere notifiche qualora si presenti un'emergenza. Per aver accesso alla piattaforma, l'utente deve necessariamente utilizzare un dispositivo mobile, con installata una versione di Android superiore alla 4.3. Le emergenze prese in considerazione per ora sono tre:

- Incendio;
- Terremoto;
- Fuga di gas.

Per poter monitorare l'edificio e rilevare eventuali anomalie si fa uso di una rete di sensori, disposti lungo tutta la struttura. Qualora si presentasse un'emergenza, gli utenti vengono guidati attraverso l'interfaccia verso l'uscita di emergenza più vicina, rispetto alla loro posizione.

## 1.3 - Obiettivi

Il progetto si pone come obiettivo la progettazione e lo sviluppo di un software secondo un'architettura orientata ai servizi. In particolare:

- Progettazione e sviluppo di un'applicazione Android, che permetta di interfacciare gli utenti con il sistema. Tale componente rappresenta il lato cosiddetto requester (all'interno dell'architettura orientata ai servizi).
- Sviluppo di un server Rest, per memorizzare e gestire i dati rilevati dai vari sensori, con la possibilità di notificare, agli utenti connessi, la presenza di emergenze.

In particolare i requisiti non funzionali richiesti dai vari stakeholder sono:

1. La mobile app deve essere sviluppata per piattaforma Android. I dispositivi mobili inviano i dati al server centrale e ricevono notifiche e assistenza dal server centrale. I dati ricevuti dai sensori e dal server devono essere visualizzati su una mappa.
2. La mobile app deve essere in grado di funzionare anche quando non è connessa al server, in particolare si dovrà poter
3. I sensori messi a disposizione sono dei beacon. In particolare sono i SensorTag della Texas Instruments CC2650STK. Essi possono collegarsi con eventuali utenti e inviare ad essi determinati dati come la posizione e alcune grandezze fisiche ambientali (temperatura, accelerometro,...). Questi dati poi saranno inviati al server centrale.
4. I messaggi scambiati devono essere nel formato JSON.
5. I dati personali devono essere trattati in accordo al Codice della Privacy
6. Il server adotta un'architettura di tipo orientata alle risorse (servizi REST).
7. Il server utilizza un Database per memorizzare e gestire i dati ricevuti dagli utenti.

## 2 - Stato dell'arte

### 2.1 - Beacon

Il *sensortag* (chiamato anche *beacon*) è uno strumento capace di reperire informazioni ambientali e di trasmettere una piccola quantità di dati, nell'ordine dei bytes, sfruttando la tecnologia wireless. Per lo sviluppo del progetto si è lavorato con un dispositivo prodotto dalla Texas Instruments, in particolare ci si è serviti del modello CC2650STK. Questo componente contiene al suo interno all'incirca una decina di sensori, utilizzati per misurare alcune informazioni d'interesse. Riguardo gli scopi del progetto, ci si è serviti solamente del sensore di temperatura, di quello di umidità, del barometro e dell'accelerometro. Una volta che vengono campionati tutti i dati, il *sensortag* ha la possibilità di inviarli, sfruttando la tecnologia del *Bluetooth Low Energy*. Questa è stata concepita per migliorare alcune caratteristiche della specifica Bluetooth, come ad esempio la possibilità di inviare i dati con un minore consumo di energia, caratteristica molto importante, alla luce del fatto che trova molta applicazione in dispositivi simili al beacon. Tutti gli scambi di informazioni si basano sull'utilizzo di *UUID*, un identificativo che permette di veicolare informazione, in maniera univoca, senza bisogno di un'autorità centrale, come nel caso del *sensortag*, dove sono presenti più sensori. Il *sensortag* invia ripetutamente in broadcast un segnale, in modo da essere individuato da opportune applicazioni, ma dentro questi bytes è presente solo un suo identificativo. Per poter leggere informazioni legate ai sensori, è necessario effettuare una procedura, descritta nell'apposita sezione, per collegarsi al beacon. Avendo come obiettivo, fra gli altri, quello di minimizzare il consumo di energia, il raggio di azione del beacon risulta piuttosto limitato: è possibile rintracciarlo lavorando in un intorno di circa 3-4 metri rispetto alla sua posizione. Questo particolare dispositivo, inoltre, ha implementato al suo interno una funzionalità che gli permette di lavorare come un iBeacon. Questo è un sistema di posizionamento, che basa il suo funzionamento su uno strumento del tutto simile al *sensortag*, sviluppato da Apple. Grazie a ciò, è possibile sfruttare le API progettate dall'azienda californiana, offrendo allo sviluppatore una gran quantità di librerie totalmente riutilizzabili.

### 2.2 - Android

Il sistema su cui deve essere eseguita l'applicazione a lato requester è Android. Tale scelta è stata per i seguenti motivi:

- Android è il sistema per telefonia mobile più usato al mondo.

- La piattaforma dispone di una grande documentazione, reperibile sulla rete, oltre ad essere dotata di molti framework che aiutano nello sviluppo di applicazioni.
- Esistono delle API Android progettate per gestire la comunicazione con i *sensortag*.

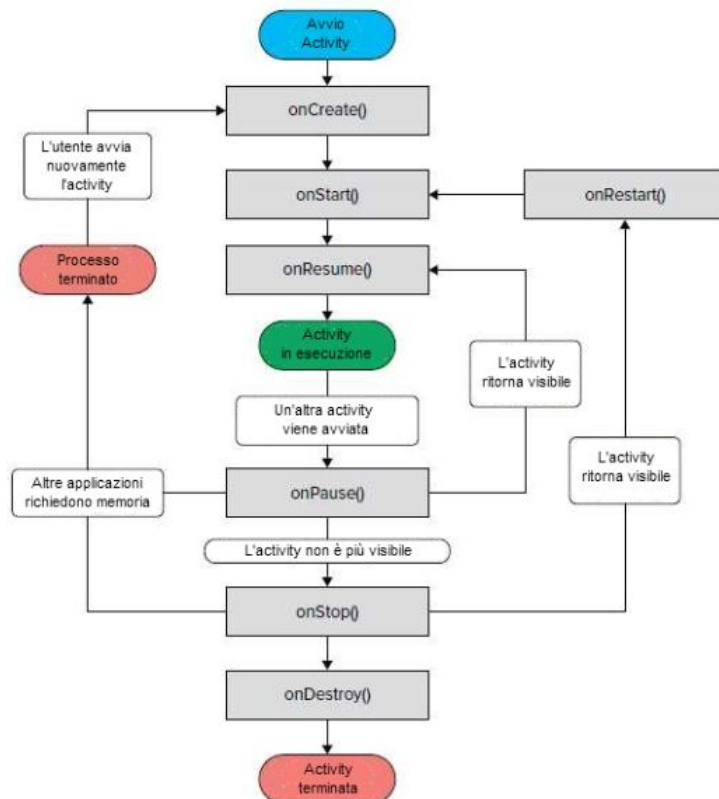
Per sviluppare l'applicazione, si è deciso di lavorare con Android Studio, un ambiente di sviluppo realizzato da Google per scrivere codice.

In questa applicazione si è fatto largo uso di alcuni strumenti molto importanti:

- **Broadcast Receiver:** questo componente è stato concepito per scambiare messaggi fra sezioni differenti di un'applicazione. Esso sta in ascolto di eventuali richieste di comunicazione (come un qualsiasi listener). Per evitare che un receiver legga messaggi indirizzati verso un altro, bisogna impostare dei filtri, in modo che ognuno interpreti come suoi quelli con un determinato intent. Quando viene ricevuto un messaggio, è possibile programmare la reazione dell'oggetto che contiene il receiver.
- **Shared Preferences:** tramite questo oggetto è possibile salvare alcuni dati in maniera permanente all'interno di un'applicazione Android. Quello che viene effettivamente memorizzato è una coppia (chiave, valore). In questa maniera, a partire dalla stringa utilizzata per il salvataggio, è possibile recuperare il corrispondente valore.
- **Activity.** L'activity rappresenta il contenuto di una finestra. Al suo interno sono contenuti gli elementi dell'interfaccia grafica dell'applicazione, con la possibilità di interfacciarsi e comunicare con l'utente. All'interno di un'applicazione possono essere definite diverse Activity, ognuna con una particolare funzione, in base agli obiettivi dell'utente. E'importante non confondere l'activity con l'interfaccia grafica: la prima, infatti, al suo interno contiene gli elementi e le operazioni legate alla seconda. Considerando che normalmente, all'interno di un'applicazione esistono più activity, esiste la possibilità di muoversi fra queste, secondo il modello di navigazione.

## 2.2.1 - Ciclo di vita di un'Activity

Il ciclo di vita di un'Activity rappresenta lo stato in cui essa può trovarsi, dal momento in cui viene avviata, fino al momento in cui viene messa in background o eliminata. La Figura sottostante mostra il ciclo di vita.



Una caratteristica molto interessante del ciclo di vita di un'activity sta nella simmetria delle chiamate dei metodi: per ogni di essi è presente una chiamata di apertura e la duale di chiusura.

Nel dettaglio c'è la onCreate che viene richiamata solo una volta dal momento in cui viene creata l'activity, analogamente ad onDestroy(), richiamata solo al momento della sua distruzione. Lo stesso discorso è applicabile alla coppia onStart()/onStop(), invocate rispettivamente nel momento in cui diventa visibile l'activity e quando non lo è più. Durante tutto il ciclo di vita di un'attività è possibile che questi vengano chiamati più volte. Infine, onResume() ed onPause(), partono quando l'activity passa in uno stato di foreground, il primo, o di background, il secondo.

## 2.3 - Servizi Rest

Le architetture software distribuite si possono dividere in:

- Architetture orientate ai componenti.
- Architetture orientate ai servizi.

L'obiettivo, su cui è basata quest'ultima, consiste nell'integrare diverse applicazioni e/o componenti all'interno di un unico software, offrendo il tutto agli utenti sotto forma di servizio. Senza voler entrare troppo nel dettaglio, un software che lavora con i servizi può sfruttare fondamentalmente due tipologie di stili architettonici: SOAP o RESTFUL. Il progetto qui presentato è stato sviluppato appoggiandosi a quest'ultima tipologia. L'architettura Restful, pur non rappresentando uno standard di riferimento,



bensi una filosofia di progettazione, è fortemente influenzata dall'utilizzo di alcune tecnologie standard, quali:

- Http, come protocollo di trasporto e invio di messaggi.
- URL, per l'identificazione delle risorse.
- XML,HTML,JSON per la rappresentazione delle risorse. Va comunque sottolineato come in realtà RestFul sia compatibile con molti altri standard di riferimento.
- text/xml, text/html, image/jpeg, etc. Usati per definire i tipi di risorse.

L'architettura di un servizio Rest può essere divisa nei seguenti livelli:

1. A livello più alto si trova la logica dell'applicazione, ovvero il livello in cui è indicato come i vari servizi debbano poter interagire tra loro. A questo livello è possibile trovare l'orchestrazione dei vari servizi, che secondo questa filosofia viene indicata tramite BPMN 2.0 (un linguaggio d'orchestrazione o di workflow).
2. Nel secondo livello si trovano i messaggi. Come già spiegato precedentemente, Rest è compatibile con molti standard per la rappresentazioni dei messaggi. Queste diverse tipologie di messaggio rappresentano il suo corpo. Il protocollo http si preoccupa quindi di eseguire l'incapsulamento e definire l'interfaccia per tali tipi di messaggi.
3. Infine, nello strato più basso, si trova il protocollo di comunicazione utilizzato per il trasporto effettivo dei messaggi. RestFul, anche in questo caso, si appoggia sul protocollo http.

## 2.4 - Glassfish

Glassfish è un application server, ovvero uno strumento software, completamente open source (disponibile sul marketplace di Eclipse) sviluppato per lavorare con JAVA EE. Questo strumento che permette, fra le altre cose, la realizzazione di un server http, fornisce un'infrastruttura che ne supporti lo sviluppo per ogni funzionalità. Considerando il fatto che i servizi web, che siano di tipo SOAP o RESTFUL, si appoggiano sul protocollo http per il trasporto dati, questo tool consente di lavorare con entrambi. Naturalmente, viste le distinzioni presenti nelle due tipologie di servizio, Glassfish da solo non è sufficiente per lavorare con entrambi. Per ovviare a questo problema, esso si appoggia a sua volta su due framework, chiamati Metro e Jersey, rispettivamente per SOAP e RESTFUL. All'interno del seguente progetto si è lavorato con il secondo per la costruzione dell'intero progetto. L'utilizzo di questo software è strettamente legato al codice con cui si lavora: per permettere a Glassfish di interagire con le risorse, è necessario specificare all'application server quali esse siano e attraverso quali metodi è possibile accedervi. Tutto ciò viene realizzato tramite l'utilizzo di annotazioni, direttamente inserite nel codice Java.

# 3 - Analisi e progettazione

Come strumento per la progettazione del Software è stato scelto un approccio ibrido

## 3.1 - Analisi dei requisiti con Tropos

Durante l'analisi dei requisiti, sono stati realizzati due diagrammi i\*, in modo da formulare una definizione ad alto livello del problema, su cui appoggiarsi e da cui partire per lo sviluppo dell'intero progetto. In questa fase sono emersi tre attori: *utente*, *beacon* e *server*. Per avere un'idea più chiara si è riflettuto sul comportamento dei tre e, infine, questi sono stati modellati come due ruoli, l'utente ed i beacon, e un agente. Questa scelta è stata dettata dalla considerazione che i primi due non possano essere identificati con uno specifico individuo, ma bensì possono essere interpretati da più soggetti. Discorso opposto, invece, per quanto riguarda invece il server, che viene identificato con un agente, anche per via della sua unicità all'interno del progetto. Il primo diagramma prodotto è stato quello denominato *Strategic Dependency Model*. La maggior parte delle dipendenze sono legate all'utente, data la sua centralità all'interno del progetto. Fra tutti gli elementi, va sottolineata la presenza di un *softgoal*, la cui presenza è dettata da esigenze emerse durante l'intervista agli stakeholder. Una volta viste le interazioni fra gli attori, si è provveduto all'analisi del *boundary*, la parte del diagramma di competenza di un attore, per ognuno di essi, ottenendo infine l'altro prodotto dell'analisi, lo *Strategic Rational Model*.

## 3.2 - Storie Utente e package

Dopo questa fase di analisi dei requisiti, si sono iniziate a sviluppare le storie utente, tenendo comunque conto dei diagrammi i\*. Da questa fase è emersa la necessità di realizzare i seguenti package:

- Beacon: al suo interno si è pensato di inserire tutte classi legate alla gestione dei sensortag, in ogni possibile aspetto, come ad esempio la ricerca del sensore e la seguente connessione ad esso.
- Communication: questo package è stato ideato con lo scopo di coordinare tutti gli aspetti legati alla comunicazione con il server, sia per inviare dati che per ricevere notifiche. Al suo interno è possibile incontrare l'implementazione di tutte le richieste http (get, post, put, delete).
- Maps: questo package è stato creato per fornire gli strumenti utili alla visualizzazione ed alla gestione delle mappe di una struttura.
- SharedStorage: l'applicazione è stata concepita secondo un'architettura Model-View-Controller. Affinché ciò possa essere realizzato, risulta necessario

predisporre una zona dell'applicazione che lavori in memoria condivisa. Per tale fine, è stato concepito questo package, il quale contiene tutti i metodi per l'aggiornamento e l'accesso ai dati.

- User: per quanto riguarda la gestione dell'utente, si è pensato di creare un apposito package, in modo da poterne gestire due aspetti di questa entità: l'utente visto come individuo che utilizza l'applicazione e l'utente visto come soggetto che si registra presso la piattaforma.
- Utility: considerando che a priori, nelle prime fasi di progettazione almeno, non è possibile conoscere a priori ogni singolo aspetto riguardo l'implementazione del codice, questo package è stato creato per contenere alcune classi di interesse generale per l'applicazione.

## 3.3 - Project Scooping

### **Problema/Opportunità (perché fare questo progetto)**

All'interno di un edificio pubblico a volte non è sufficiente l'utilizzo di mappe cartacee, vista la grandezza della struttura. Nel caso in cui si presenti un'emergenza, considerando la quantità di soggetti presenti, diventa anche difficoltoso coordinare i movimenti della grande mole di individui. Con la creazione di questo sistema è possibile non solo gestire in maniera automatizzata ogni possibile emergenza, ma consente inoltre di rispondere alle particolari esigenze degli utenti, riferite alla ricerca di un luogo, anche in condizioni normali.

### **Project Goal**

Lo scopo di questo progetto è realizzare un sistema software che possa gestire le emergenze (incendi, terremoti, ecc..) all'interno di edifici pubblici. Tutto ciò verrà realizzato grazie alla comunicazione con alcuni dispositivi (chiamati beacon), disposti su tutta l'area presa in analisi. Questi apparecchi producono dei dati che poi vengono analizzati da un server che indica ad ogni utente collegato il piano di fuga migliore (cioè con la minima presenza di rischi).

### **Obiettivi (specifici, misurabili, assegnabili, realistici, relativi al tempo), durata e costi**

Gli obiettivi da conseguire all'interno del progetto sono i seguenti: realizzare e gestire la comunicazione tra l'applicativo ed il server, per poter inviare e ricevere i necessari al funzionamento dell'applicazione (ad esempio . Un altro obiettivo da ottenere consiste nel far comunicare il dispositivo con il beacon, per poter estrarre i dati dai sensori, conoscendo così in che stato si trovi l'ambiente (se in situazione di pace o di

emergenza). Inoltre, quando il dispositivo si connette con l'apparecchio, il sistema deve registrare la posizione in cui si trova.

Per poter identificare i dispositivi specifici e poter affrontare in maniera più efficiente possibile ogni emergenza (in vista dell'intervento dei soccorsi), va gestito l'insieme degli utenti che decidono di utilizzare l'applicazione, sia nel caso in cui si fossero registrati alla piattaforma, sia nel caso in cui utilizzino l'applicazione senza aver effettuato l'accesso. Per questo va realizzata un'interfaccia grafica comune, considerando come l'unica differenza sia quella riferita all'identificazione del dispositivo. Gli utenti che decidono di registrarsi presso l'applicativo infatti devono effettuare una procedura simile a quella che viene compilata quando ci si iscrive ad un servizio online.

Per il corretto funzionamento dell'applicazione è necessario implementare il sistema delle notifiche, che devono essere messe in evidenza dall'applicazione, per eventuali comunicazioni con l'utente.

L'applicazione può lavorare in due modalità, in condizioni normali e in caso di emergenza, ognuna delle quali ha delle funzionalità specifiche. In entrambi i casi, bisogna fare in modo che vengano visualizzate le mappe riguardanti il percorso che l'utente deve seguire per giungere all'obiettivo selezionato. La distinzione fra le due situazioni è dovuta al fatto che nel primo caso il percorso è determinato in base al luogo dove l'utente desidera recarsi, mentre nel secondo vengono indicate le vie di fuga per uscire dall'edificio, in maniera predefinita.

### **Criteri di successo (risultati misurabili)**

Il progetto potrà ritenersi concluso nel momento in cui saranno realizzate tre funzionalità: l'applicazione darà la possibilità all'utente di ricercare aule e posti sensibili in condizioni di funzionamento "normali" (cioè in assenza di emergenze), sarà capace localizzare, in tempo reale, ogni singolo utente connesso al sistema e, in caso di emergenza, indicare il percorso di fuga disposto su una mappa.

### **Assunzioni rischi e ostacoli**

Si suppone che il beacon sia sempre connesso e che non presentino malfunzionamenti. Un'altra assunzione fatta è che tutte le persone all'interno dell'edificio utilizzino l'applicazione e che quindi si possano ricavare tutte le loro posizioni, in tempo reale. Il rischio maggiore legato all'applicazione è la tempestività con cui viene consegnata agli utenti la segnalazione dell'emergenza agli utenti: se l'applicazione non funzionasse correttamente, gli utenti, invece di dirigersi verso un punto sicuro, potrebbero, nel panico generale, andare verso una direzione pericolosa. L'ostacolo principale affrontato riguarda però la copertura dell'intero edificio, considerando il ridotto raggio d'azione del beacon, che potrebbe generare zone d'ombra.

## 3.4 - User Story

In fase di analisi dei requisiti sono state definite le seguente User Story:

User Story Client:

1. Gestione utente
  - 1.1. Scelta dei criteri per l'identificazione dell'utente.
  - 1.2. Scelta dei criteri per distinguere fra un utente registrato o meno.
  - 1.3. Disegno della form iscrizione.
  - 1.4. Scelta dei campi per la registrazione.
  - 1.5. Disegno della form login.
  - 1.6. Gestione profilo.
2. Comunicazione beacon
  - 2.1. Scelta dei criteri per l'identificazione dispositivo.
  - 2.2. Scelta dei sensori da cui leggere i dati dal dispositivo.
  - 2.3. Scelta delle librerie software da utilizzare.
  - 2.4. Scelta dei passi da seguire per trovare un dispositivo.
  - 2.5. Scelta dei passi da seguire per leggere i dati dei sensori.
  - 2.6. Comunicazione dispositivo
3. Gestione mappa
  - 3.1. Modalità di caricamento delle mappe in memoria
  - 3.2. Modalità per il disegno della mappa
  - 3.3. Disegno dell'interfaccia grafica per ricercare le mappe
  - 3.4. Creazione documenti contenenti i nodi della mappa
4. Emergenze
  - 4.1. Visualizzazione mappa
  - 4.2. Scelta del metodo per la notifica dell'emergenza.
  - 4.3. Scelta del comportamento dell'applicazione durante l'emergenza.
  - 4.4. Scelta degli elementi visivi da mostrare sullo schermo.
  - 4.5. Condizione di terminazione dell'emergenza.
  - 4.6. Metodi per la gestione delle notifiche push.
5. Comunicazione server
  - 5.1. Decisione riguardo quali dati da inviare al server
  - 5.2. Decisione riguardo il metodo di invio dei dati
  - 5.3. Modalità comunicazione con il server
  - 5.4. Invio dati dei sensori
  - 5.5. Ricezione segnalazioni

User Story Server:

6. Server
  - 6.1. Scelta delle tabelle per realizzare il database
  - 6.2. Creazione del database
  - 6.3. Scelta criterio per la segnalazione dell'emergenza
  - 6.4. Scelta criteri per gestire gli utenti

## 3.5 - Analisi CRC

Nell'analisi dei requisiti successivamente alla definizione di tutti i diagrammi, storie utente e task card (visibili all'interno della cartella documentazione):

Utente

Responsability:

- ricerca di un luogo
- registrazione presso la piattaforma
- accesso alla piattaforma
- visualizzazione della mappa
- ricezione dei dati dal beacon
- invio dei dati al server
- ricezione delle notifiche

Collaboration:

-  
server  
server  
-  
beacon  
server  
server

Beacon:

Responsability:

- invio dei dati

Collaboration:

utente

Server:

Responsability:

- ricezione delle informazioni
- invio delle notifiche di emergenza
- elaborazione dei dati ricevuti
- gestione degli utenti
- gestione del percorso di fuga
- aggiornamento della mappa durante l'emergenza

Collaboration:

utente  
utente  
-  
-  
-  
-

## 3.6 - Architettura

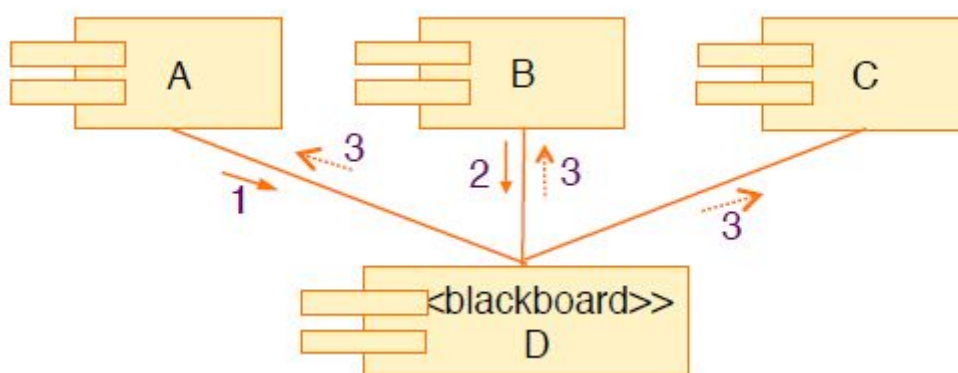
La scelta dell'architettura è stata molto complessa dato che il sistema presentava il problema della condivisione di dati tra classi differenti, garantendone la coerenza.

Per sopperire a questo inizialmente si è discusso del come risolvere tale quesito e la soluzione intrapresa inizialmente, probabilmente anche la più ovvia, si può riassumere nei seguenti passi:

- creare una struttura dati dinamica;
- definire metodi setters e getters per la gestione dei dati;
- implementare delle classi che utilizzano tali metodi.

Tale approccio è risultato però troppo semplicistico, in quanto è emersa l'impossibilità di gestire le chiamate asincrone. Per questa ragione è stato pensato un sistema alternativo, che adottasse un approccio orientato all'aggiornamento delle informazioni da condividere. Il tutto è stato realizzato attraverso il modulo "*Shared Storage*". Con il seguente esempio si illustra il funzionamento del sistema: si supponga che una classe A abbia bisogno di lavorare con delle informazioni sempre aggiornate, mentre una certa classe B, indipendentemente dalla prima, modifichi un dato di interesse comune. Quest'ultima potrebbe direttamente lavorare con la copia del dato di A, ma tale scelta progettuale risulterebbe essere poco efficiente.

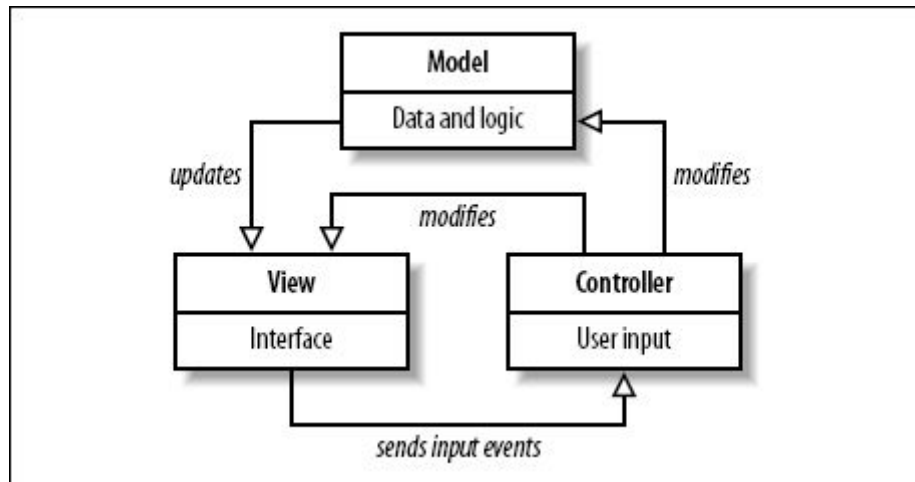
Se si realizzasse una struttura in cui, è possibile far registrare un oggetto come *subscribers*, o sottoscrittore, si ottiene la possibilità di effettuare modifiche o recuperare l'ultima versione delle informazioni. L'intero meccanismo è del tutto automatico, garantisce la consistenza delle informazioni e prende spunto dallo stile architettonico chiamato *Blackboard*.



Principio di funzionamento:

1. Tutti i moduli che vogliono lavorare con la struttura dati si registrano (subscriber).
2. Quando un modulo, in questo caso B, lavora con il dato, inserisce la nuova copia all'interno della blackboard.
3. La struttura notifica a tutti i subscriber la nuova copia del dato.

Ecco che risolto il problema la scelta dello stile architettonico dell'applicazione si basa sulla suddivisione in diversi moduli. Tale modello prende il nome di *Model-View-Controller*, in cui si pone una netta distinzione tra logica dell'applicazione (Controller), Interfaccia utente (View) e Dati (Model). La motivazione dietro l'utilizzo di tale schema è dettato dalla ricerca di una soluzione efficiente, tra diversi componenti. Il Model dell'applicazione viene definito, quindi, dalla struttura *Shared Storage*.



Per realizzare tutto questo sono state definite delle classi per la gestione delle singole informazioni e un'interfaccia, la quale viene implementata (realizzata) da ogni singolo subscriber che si registra.

Entrando nel dettaglio abbiamo le seguenti classi:

- Data
- SharedStorage
- DataListener
- Notification
- UserPosition

Lo schema è visibile all'interno del diagramma delle classi, chiamato Shared Storage, presente nell'apposita cartella documentazione.

La prima classe analizzata è *Data*: questa è concepita come un riferimento statico, utilizzato per accedere alle strutture dati *Notification* e *UserPosition*, attraverso due metodi getter. Il nocciolo del problema, come esposto sopra, riguarda la gestione delle modifiche alla struttura dati, derivanti dalle chiamate asincrone. A questo punto entra in gioco la classe *SharedStorage*, progettata perché sia astratta, che funge da modello per le strutture *blackboard* implementate.

Essa si occupa della gestione di una serie di *listeners*, una lista di sottoscrittori, registrati presso la struttura dati specifica. Più specificatamente, ogni oggetto che si registra come subscriber nella struttura *UserPosition*, lo fa esclusivamente per conoscere in tempo reale la posizione attuale dell'utente; nel caso di *Notification*, invece, per ricevere una notifica riguardo un'eventuale emergenza segnalata dal



server. La sottoscrizione avviene tramite il metodo `addListener()` della struttura, il quale banalmente aggiunge l'oggetto alla struttura dati dinamica.

Infine, considerando il fatto che a questo punto sono stati progettati i listeners ed i rispettivi comportamenti in fase di aggiornamento, è stato realizzato il metodo `updateInformation()`, che lavora comportandosi come un *trigger*, poiché richiama tutti i metodi `retrive()` dei sottoscrittori. In questo modo, ognuno di essi recupera una copia aggiornata delle informazioni condivise.

Le classi *UserPosition* e *Notification* hanno il compito di implementare tale classe, preoccupandosi di scrivere i relativi metodi modificatori. Ovviamente sfruttando l'ereditarietà quest'ultimi anche richiamano il metodo `updateInformation()` della classe padre, attivando quindi la procedura di aggiornamento.

Nel dettaglio, seguendo il diagramma delle sequenze presente nell'appendice, viene spiegata la sequenza il funzionamento della struttura per entrambe le procedure:

Lo schema è visibile all'interno del diagramma delle sequenze presente nell'apposita cartella documentazione.

- *Notification*, tale struttura è stata predisposta per gestire la ricezione delle notifiche legate alle emergenze. Quando il server invia una notifica, viene avviata la procedura di emergenza interna all'applicazione, che mostra la mappa riferita alla posizione dell'utente, con eventuali pericoli lungo il percorso. Il sistema di ricezione è descritto dalla classe *HttpReceiveThread* che si occupa di gestire la comunicazione con il server per ricevere messaggi da questo, in maniera asincrona. Al termine di quest'ultima operazione, viene infine chiamata la funzione `update()`, che aggiorna la struttura dati. Va sottolineato il fatto che ogni sottoscrittore, tra cui la classe che gestisce la mappa, successivamente alla chiamata, sfrutta la funzionalità `retrive()` per dotarsi di una copia aggiornata del dato.
- *Userposition*, in questa classe, invece, si utilizza uno strumento per la gestione della posizione dell'utente. Nel momento in cui questo si muove all'interno della struttura ed intercetta un beacon, si verificano due eventi: l'invio della posizione corrente al server ed il conseguente l'aggiornamento della struttura dati. Seguendo il diagramma delle sequenze di tale procedura, è possibile notare che quando la classe, registrata come sottoscrittore, *BeaconScanner* incontra un beacon, viene subito richiamato il metodo `update()`, opportunamente implementato da tale classe, con il compito di modificare il valore della variabile contenente il beacon corrente, che a livello applicativo si può ritenere rappresentare la posizione dell'utente. Di conseguenza, dopo aver modificato il dato, parte la procedura di aggiornamento, in cui tutti i sottoscrittori ottengono la nuova posizione. La classe gestore della mappa è stata concepita in modo tale da modificare la posizione dell'utente in tempo reale, senza interferire con il resto del sistema.

## 4 - Communication

Un punto centrale, per l'intero funzionamento dell'applicazione, riguarda la necessità di scambiare messaggi con un server, il quale memorizza permanentemente le informazioni e si occupa di gestire tutto ciò che concerne le emergenze. Nel dettaglio, il modello progettato per la comunicazione è stato sviluppato concentrandosi sulla realizzazione di due sezioni:

- lato client-to-server, che si occupa di gestire ogni aspetto riguardo l'invio di messaggi (e le conseguenti risposte) dall'applicazione verso il server Rest;
- lato server-to-client, che riguarda tutta la gestione delle emergenze, comunicate dal server verso il client in maniera asincrona.

Ovviamente, per consentire tutto ciò, è stata realizzata un'infrastruttura che monitori lo stato della connessione, all'interno di ogni comunicazione. Entrando più nel dettaglio, questi controlli sono stati inseriti all'interno di una classe centrale, chiamata *WelcomeActivity*, implementata esclusivamente per poter impostare alcuni parametri all'avvio dell'applicazione. I controlli non lavorano solamente in questa, bensì sono presenti anche all'interno di ogni procedura di connessione al server, nelle diverse chiamate effettuate dalle diverse classi.

### 4.1 - Classe *WelcomeActivity*

Tale classe dà la possibilità, al primo avvio dell'app, di:

- definire l'indirizzo IP della macchina server a cui collegarsi;
- accedere all'applicazione in modalità *offline*. Questa modalità non è però sempre accessibile: affinché ciò possa accadere, devono essere stati scaricati precedentemente i file contenente la lista dei sensori sparsi nell'edificio e la lista dei punti dove l'utente può dirigersi;
- scegliere l'edificio in cui l'applicazione deve lavorare, in modo da poter richiedere al server solamente i dati riferiti ad esso.

Nel caso in cui venga spuntata la seconda opzione, l'intera applicazione funziona in una modalità chiamata *offline*, la quale impedisce ogni possibile accesso alle funzionalità dove è richiesta l'interazione tra applicazione ed il server, come ad esempio il login, o l'iscrizione. In questa maniera non risulta possibile inviare dati relativi alla posizione o quelli estratti dai sensori, andando a creare una sorta di falla nella rete di utenti, con alcuni problemi, descritti in seguito, che possono emergere. Per evitare che un utente, non utilizzando spesso l'applicazione si ritrovi a lavorare con documenti non aggiornati, è stato implementato un algoritmo di versioning della lista di nodi e stanze di un edificio. Questa situazione risulta comunque fuori dall'ordinario, ma in caso si presenti un'emergenza, avere un documento non aggiornato può essere molto rischiosa per l'utente. Nel caso in cui i gestori del

server decidano di modificare i dati di una stanza o di un beacon, hanno anche l'obbligo di modificare la versione corrente di mappe e stanze, modificando l'opportuna voce nel database. Come conseguenza di questo controllo, all'avvio dell'applicazione, se la versione risultasse obsoleta, partirebbe immediatamente la sequenza di scaricamento delle nuove mappe, in maniera del tutto trasparente per l'utente.

## 4.2 - Implementazione Server Rest

Terminata la descrizione della sezione introduttiva, riguardante il controllo e la definizione dell'indirizzo del server, si può entrare più nel merito, riguardo la comunicazione verso il server.

Quest'ultimo adotta un'architettura a servizi di tipo Rest, orientata quindi alle risorse. L'intera comunicazione si basa sui quattro metodi HTTP (Post,Get,Put,Delete) per interagire con le risorse accessibili tramite un URI (Uniform Resource Identifier).

Ognuno di questi metodi è stato realizzato sfruttando le librerie Android, al fine di utilizzare il protocollo. Ovviamente la comunicazione deve essere asincrona rispetto al resto dell'applicazione, perciò ogni chiamata è stata gestita grazie all'ausilio della classe AsyncTask, che permette di costruire un processo che lavora in background.

La comunicazione richiede, come da specifiche di progetto, l'utilizzo di documenti in formato JSON. Per lavorare con tale estensione, tutti i messaggi sfruttano le funzionalità di due classi che costruiscono e decodificano stringhe della struttura richiesta.

Nel dettaglio i metodi implementati sono:

- GET, utilizzato per la raccolta delle informazioni, il messaggio viene preso e decodificato dal formato JSON;
- POST, permette di creare delle nuove risorse prendendo in input le informazioni contenute nel corpo della richiesta;
- DELETE, dà la possibilità di cancellare dei dati nel server;
- PUT, metodo idempotente sfruttato per modificare un certo oggetto, cambiandone il contenuto.

In ultima analisi, è necessario analizzare la ricezione dei messaggi inviati dal server per notificare un'emergenza. In particolar modo è stata implementata una classe che permetta di ricevere messaggi nella porta 8888 del dispositivo su cui è installata l'app. Ogni volta che arriva una notifica, il contenuto viene estratto e inserito all'interno della struttura dati condivisa *Notification*. Se il payload del messaggio fosse vuoto, allora l'applicazione dovrebbe interpretare il messaggio come la comunicazione che lo stato di emergenza sia rientrato, riportando l'applicazione in condizione di normalità. In caso contrario, il sistema deve avvertire l'utente della presenza di uno stato di pericolo, in modo da permettergli di indirizzarsi verso le uscite di emergenza, evitando possibili pericoli disposti sulla mappa.

## 4.3 - Beacon

Uno degli elementi fondamentali, emersi dall'analisi dei requisiti, riguarda senza dubbio la gestione dei sensori *beacon*, posizionati all'interno della struttura. Attraverso questi, infatti, è possibile estrarre alcune informazioni legate all'ambiente in cui si trovano ad operare, tramite l'uso dei sensori di cui dispongono (come ad esempio la temperatura o la pressione), in modo da poter rilevare eventuali problemi in una zona dell'edificio. Qualora queste si presentino, può essere un ottimo strumento per indirizzare velocemente gli utenti verso le vie di fuga, oltre che per consentire alle autorità un pronto intervento, fornendo informazioni nell'eventualità in cui ci siano persone in situazioni di pericolo. Considerando comunque il fatto che la presenza di un'emergenza rappresenti una situazione extra ordinaria, i *beacon* possono anche rappresentare un ottimo strumento per consentire all'utente di muoversi dentro una struttura, tramite la realizzazione un'infrastruttura che gli permetta di identificare la propria posizione su di una mappa con quella del sensore ad esso più vicino.

### 4.3.1 - Posizionamento dei Beacon

Prima di addentrarsi nella spiegazione delle idee scaturite in fase progettazione per questa sezione dell'applicazione, bisogna soffermarsi sul metodo utilizzato per associare un sensortag ad una posizione sulla mappa. Inizialmente si è pensato di modificare il nome del dispositivo beacon, considerando la presenza del modulo Bluetooth, inserendo tutte le informazioni relative alla posizione (piano e coordinate x,y) dentro questa stringa. L'applicazione, a questo punto, una volta rilevato un sensortag nelle sue vicinanze, avrebbe semplicemente dovuto effettuare il parsing di questa stringa, ottenendo ogni informazione. Questa soluzione avrebbe inoltre garantito la possibilità di rendere ogni sostituzione di un nodo, nella rete dei beacon, completamente trasparente all'utente, oltre che velocizzare molte procedure. Per poter mettere in pratica ciò, sarebbe stato sufficiente far lavorare il dispositivo in modalità *iBeacon*, ma questo aspetto è risultato un problema, come spiegato in seguito. Per poter superare questa difficoltà, si è deciso di salvare all'interno di un documento CSV, presente sul server, tutti i sensortag presenti all'interno di un edificio, ognuno di questi è identificato da una struttura composta da: *BluetoothAddress*, piano, coordinata x, coordinata y. Questa soluzione ha permesso di superare il problema, ma al suo interno comunque mantiene delle incognite, in quanto rende l'applicazione più difficilmente estendibile, in quanto ogni modifica ad un nodo a questo punto risulta non più trasparente per l'utente. Alla luce di ciò, si è implementato un sistema di versioning, ovvero di controllo della versione salvata sul dispositivo, in modo che ad ogni accensione dell'applicazione, questa venga

controllata e, qualora non siano presenti i dati aggiornati, si occupa di richiederli al server.

### 4.3.2 - Funzione Scan

La logica legata al funzionamento dei *sensortag* risulta piuttosto semplice: l'applicazione ricerca periodicamente uno di questi nei pressi della sua posizione (a potenza del segnale è rilevabile in un intorno di circa tre-quattro metri) e, una volta trovato può leggere i dati misurati dai numerosi sensori presenti al suo interno. Trascorso un certo periodo di attesa, l'applicazione ricomincia la ricerca di un *sensortag*. L'azione svolta nella ricerca di beacon viene detta *scan*. Alla luce del fatto che l'app debba lavorare in maniera differente in base al contesto in cui si trova l'utente, è emersa la necessità di modellare tre diverse modalità di funzionamento.

- La prima viene definita *normale* (già nominata in precedenza) ed indica la condizione di default in cui lavora: l'app ricerca un sensore nelle sue vicinanze e qualora lo trovi si connette ad esso per leggerne i dati ambientali, per poi inviarli ad un server che provveda ad elaborarli. In questa tipologia di funzionamento gli intervalli di tempo che intercorrono fra due ricerche consecutive non sono molto ravvicinati: si considera il fatto che quando un utente si trova dentro una struttura pubblica raramente si trova a spostarsi, assumendo quindi per la maggior parte del tempo una posizione fissa. Partendo da questa assunzione si è deciso di far trascorrere un periodo di tempo nell'ordine della decina di minuti fra due *scan* consecutivi.
- La seconda modalità di funzionamento viene definita *searching*, in quanto viene utilizzata nel momento in cui l'utente vuole spostarsi all'interno della struttura, visualizzando una mappa, con l'apposita funzionalità fornita dall'applicazione. Dal momento in cui si sta lavorando in questa maniera, risulta altamente probabile che l'utente si stia muovendo, perciò la scelta dei tempi segue un approccio diverso. L'intervallo di tempo che intercorre fra due *scan* consecutivi viene ridotto drasticamente, divenendo all'incirca di circa cinque secondi. Inoltre in questa modalità di funzionamento l'applicazione non si collega al sensore, non estraendo leggendo alcun dato ambientale dai sensori, per fare in modo che la procedura risulti più veloce.
- Infine la terza ed ultima modalità di funzionamento, detta *emergency*, viene attivata automaticamente nel momento in cui all'applicazione viene notificata la presenza di un'emergenza. Questa modalità è simile alla precedente, considerando che neppure questa si collega al sensore per leggerne i dati, ma i tempi di attesa fra due *scan* sono ancora più ristretti, per poter fronteggiare tempestivamente un'evacuazione. Per poter gestire queste diverse configurazioni, si è ideata una classe, chiamata Setup, contenente tutte queste informazioni.

L'applicazione è concepita in modo che questi elementi sopra descritti siano parametrici: se l'applicazione lavora in modalità online, connettendosi quindi ad un server, allora i dati legati ai periodi vengono scaricati ogni volta, altrimenti, se si lavora in modalità offline, comunque sono presenti dei valori di default a cui l'app fa riferimento.

### 4.3.3 - Macchina a Stati

Per poter gestire in maniera ottimale queste situazioni, si è reso necessario uno studio del funzionamento dei *beacon*, il quale ha fornito delle indicazioni su come interfacciarsi ad essi: sono stati individuati e progettati due cicli fondamentali, ovvero il *ciclo di scan* ed il *ciclo di connessione*. Il primo si occupa di ricercare i beacon, mentre il secondo gestisce il collegamento ad essi per la lettura dei dati. Per poter gestire entrambe le situazioni si è pensato di costruire una macchina a stati finiti, nello specifico con stati a proposizioni atomiche. Si consideri che lo stato 1 (in cui ci si collega al beacon per leggerne i dati ambientali) contiene al suo interno un'altra macchina a stati e quindi per questo è definito stato composito. Questa viene invece rappresentata come una macchina a transizioni con variabili. Le macchine a stati implementate descrivono i due cicli sopra presentati: all'interno del *ciclo di scan*, uno stato è rappresentato dal ciclo di connessione. A fronte di questi oggetti matematici utilizzati per modellare la problematica, si è deciso di realizzare una classe astratta, chiamata *StateMachine*, contenente i metodi necessari per eseguire le operazioni di ogni stato e, in base alla valutazione di alcune variabili, andare a valutare il prossimo stato in cui debba lavorare la macchina. Le due classi che si occupano di gestire lo scanner e la connessione, rispettivamente *BeaconScanner* e *BeaconConnection*, sono figlie di questa classe astratta. Prima di addentrarsi nella spiegazione tecnica di ciò che è stato realizzato, vanno menzionate le tecnologie software utilizzate.

#### 4.3.3.1 - Librerie Android

I *sensortag*, come già espresso precedentemente, comunicano con i dispositivi sui quali lavora l'applicazione tramite la tecnologia Bluetooth Low Energy (BLE). Seguendo una delle specifiche di progettazione, ovvero la realizzazione di un applicativo Android, una prima scelta è ricaduta sull'uso delle librerie da utilizzare per gestire tale comunicazione. Difatti, poiché questa tecnologia si sta diffondendo molto velocemente sul mercato, molti utenti hanno offerto la propria soluzione riferita a questa problematica, in maniera *opensource*. In un primo momento, anche alla luce del fatto che la Texas Instrument, l'azienda produttrice dei sensori, ha realizzato i dispositivi implementando al loro interno anche il comportamento in modalità iBeacon, si è pensato di appoggiarsi ad apposite librerie sviluppate dalla Apple, contenenti già molte delle funzionalità ideali per lo scopo di questo progetto. In un secondo momento, emersa l'impossibilità di poter lavorare in modalità iBeacon, se non modificato il firmware dei *sensortag*, si è quindi optato per l'utilizzo delle librerie

standard di Android, nello specifico tutte quelle utili per la comunicazione BLE. Va sottolineato come la comunicazione con i dispositivi tramite Bluetooth Low Energy sia afflitta da una conflittualità legata alle librerie fornite da Android stessa: esistono infatti due opportunità per poter lavorare, con la prima che risulta essere però deprecata. In questa variante non esiste un oggetto specifico per il BLE, bensì si fa riferimento ad un oggetto chiamato *BluetoothAdapter*, il quale ha la possibilità di implementare un'interfaccia, chiamata *LeScanCallback*, la quale al suo interno contiene tutte le callback necessarie per lo scan. Nel corso degli anni Google, l'azienda che sviluppa le API (l'insieme delle procedure disponibili per l'utente) per scrivere codice in Android, ha però deciso di deprecare questa funzionalità da una certa versione in poi, sostituendo l'interfaccia con un vero e proprio oggetto chiamato *LeScanner*, anch'esso legato al *BluetoothAdapter*. Non avendo una precisa indicazione sulla versione Android in cui vada utilizzata l'applicazione, si è deciso di coprire più dispositivi possibile, considerando comunque che tutti i dispositivi con versioni precedenti la 4.3 non dispongono della comunicazione BLE, implementando all'interno del progetto entrambe le modalità di *scan*. Al momento dell'esecuzione, in base alla versione corrente Android, se ne utilizza uno dei due, in maniera totalmente trasparente per l'utente, che comunque non nota la differenza fra i due. Per quanto riguarda invece la connessione fra il beacon e il dispositivo, Android mette a disposizione un'interfaccia, questa volta uniforme per tutte le librerie che supportano la comunicazione BLE, chiamata *BluetoothGattCallback*. Questa al suo interno implementa i metodi per coordinare al meglio ogni fase della connessione, dalla connessione vera e propria, fino alla lettura dei dati.

#### 4.3.3.2 - Sincronizzazione

Un elemento cruciale per la gestione degli strumenti messi a disposizione dalle librerie, riguardante tutte le interfacce presenti, si identifica nella sincronizzazione dei metodi e delle callback. Considerando il fatto che l'implementazione di questa interfaccia risulta necessaria, ma che comunque ha bisogno di una struttura costruita attorno per poter essere gestita in maniera ottimale, si è deciso di realizzare un'apposita classe, *GattLeService*, all'interno del quale oltre all'implementazione dell'interfaccia sono stati inseriti metodi accessori per poter gestire meglio ogni singola chiamata ad una callback, oltre che per lavorare con dati inviati sotto forma di byte. Queste callback sono state scritte dagli sviluppatori Android, in modo tale che, alla chiamata di alcuni metodi delle librerie, essi siano automaticamente invocati. Considerando che i *sensortag* scambiano informazione tramite Bluetooth, è stato necessario dotarsi di strumenti che permettessero di interpretare i byte letti, in modo che fossero comprensibili anche ad alto livello. Tornando nello specifico a tutti gli elementi implementati nelle interfacce, sia per la connessione che per lo scanner, va posta l'attenzione su un altro elemento delicato: tutti questi sono stati concepiti come thread, ovvero come processi a sé stanti. Per evitare di finire in condizioni in cui la

mancanza di sincronizzazione portasse a gravi errori logici e di programmazione, è risultato necessario definire un sistema per poter scongiurare questa eventualità. Partendo dal presupposto che una funzionalità offerta da questi metodi rappresentasse un singolo stato della macchina e che, logicamente, due stati non potessero andare in esecuzione contemporaneamente, si sono subito presentate due possibili soluzioni in ambito progettuale. La prima fondata sull'idea che ognuno di questi metodi, una volta terminato, invocasse lui stesso il metodo successivo, in modo da superare eventuali problemi di sincronizzazione. Per quanto questa fosse di più semplice realizzazione, fin dalla sua descrizione ad alto livello è emerso come questa soluzione non solo non evitasse i problemi di sincronizzazione, ma che comunque fosse troppo poco dinamica per altre esigenze progettuali, come ad esempio per la gestione dell'interruzione forzata dello scanner. Nonostante dalla sua avesse la semplicità concettuale, si è optato per la creazione di un sistema di scambio di messaggi fra i metodi: si è pensato di fare in modo che ognuno di questi, nel momento della terminazione, comunicasse tramite un messaggio di ACKNOWLEDGE, la fine della propria esecuzione. Sono stati quindi realizzati dei `broadcastReceiver`, oggetti Android per permettere lo scambio di messaggi tra parti diverse del programma. In questa maniera, oltre a comunicare la terminazione di un metodo per poi gestire il conseguente cambio di stato, è anche possibile andare a gestire situazioni più delicate, come la terminazione anticipata di una connessione o l'interruzione dello scan, che verranno presentate in seguito. Per farsi un'idea di come è stato gestito questo scambio di messaggi per la gestione delle due macchine, si faccia riferimento all'apposita immagine presente nell'appendice.

#### 4.3.3.3 - BeaconScanner

La prima macchina, contenuta nella classe *BeaconScanner*, a stati presentata è quella del *ciclo di scan*. Lo scopo di questa si identifica nel rintracciare un *sensortag* a cui collegarsi, all'interno del raggio d'azione di questi. Come è possibile vedere dalla figura (nell'appendice) sono presenti quattro diversi stati. Qui di seguito viene descritta la funzione di ognuno di essi:

- Stato 0: l'unico stato iniziale possibile. Al suo interno viene lanciato il metodo `discoverBLEDevices()`, il quale ha la funzione di inizializzare e lanciare il thread che si occupa della ricerca vera e propria dei dispositivi nelle vicinanze. In questa fase vengono utilizzate alcune callback fornite dalle librerie di Android (di cui si è già parlato in precedenza), per poter iniziare e fermare lo scan. Considerando la forte diffusione dei dispositivi Bluetooth nella quotidianità di ogni utente, è stata aggiunta una *maschera*, con l'indirizzo UUID comune per tutti i *sensortag*, in modo da evitare che nello scan rientrino anche dispositivi che nulla hanno a che fare con gli scopi dell'app. La macchina a stati passa relativamente poco tempo in questo stato iniziale, con piccole distinzioni legate alle modalità di funzionamento dell'applicazione, rispetto a quanto ne passa negli altri. È stato



notato che già in un intervallo di due-tre secondi, l'applicazione riesce a rintracciare tutti i sensori nelle sue prossimità, per questa ragione ci si è orientati verso un periodo corto (compreso fra i tre ed i cinque secondi). L'obiettivo di questa fase risulta, come già detto, rintracciare tutti i beacon presenti nel raggio d'azione dell'utente ed eventualmente collegarsi al più vicino, o comunque legare la propria posizione a questo. Per fare ciò è necessaria un'analisi globale di tutti i sensortag incontrati. Si è quindi realizzato un oggetto chiamato `LeDeviceListAdapter`, all'interno del quale viene salvato ogni beacon incontrato insieme al suo RSSI (segnale di potenza). Qui, una volta terminato lo scan vero e proprio, prima di passare alla fase successiva, vengono elaborate tutte le informazioni ricevute sulla potenza dei sensori individuati ed infine viene restituito un riferimento al beacon più vicino, in modo che l'utente si agganci a quello ed eventualmente ne legga i dati.

- Stato 1: in questo stato è contenuta una macchina a stati per la gestione del collegamento al sensortag più vicino. Verrà descritta più nello specifico in seguito. Come detto in precedenza, non sempre terminata la ricerca, si leggono i dati ambientali. Affinché ciò avvenga vanno rispettate alcune condizioni: l'applicazione deve lavorare in maniera *normal* e dalla fase di scan deve esser stato rintracciato almeno un beacon. Qualora una di queste non sia stata rispettata, si passa direttamente allo stato 2. Come si è già specificato in precedenza si è preferito eliminare la lettura dei dati dalle altre modalità di funzionamento.
- Stato 2: terminato la ricerca e l'eventuale lettura dei dati, bisogna attendere un periodo di tempo, prima di ricominciare con il ciclo di scan. Il periodo d'attesa varia in funzione della modalità di funzionamento dell'applicazione, passando dai dieci minuti della modalità *normal*, ai più ravvicinati cinque e tre secondi, rispettivamente per le modalità di funzionamento *searching* ed *emergency*. La scelta di inserire un periodo di attesa è stata fatta per non sovraccaricare di lavoro l'applicazione, alla luce del fatto che in periodi di tempo brevi l'utente non cambi sensibilmente la propria posizione, ed evitando quindi all'applicazione di leggere gli stessi dati. Alla base di questa considerazione sta il fatto che per la maggior parte del tempo l'utente non si trovi in condizioni d'emergenza o stia cercando una stanza, bensì sia fermo in un punto. Ciò non toglie comunque che, variando l'attesa in funzione della modalità, le altre due situazioni possano essere gestite in maniera ottimale. Qualora non si interrompa il ciclo, terminata l'attesa ricomincia una nuova ricerca (si torna quindi nello stato 0).
- Stato 3: lo stato finale del ciclo, all'interno del quale viene solamente comunicato alla classe *MainApplication* che la macchina ha terminato di lavorare, in modo che possa gestire il cambio di scanner o la chiusura dell'applicazione. Questo stato viene raggiunto solamente quando l'utente esegue determinate azioni (come ad esempio ricercare un elemento sulla mappa, poiché bisogna passare

dalla modalità normal, di default, alla modalità searching) oppure il server ha comunicato un'emergenza.

#### 4.3.3.4 - BeaconConnection

Come già sottolineato in precedenza, lo stato 1 contiene a sua volta una macchina a stati che, a differenza della prima, oltre a contenere molti più stati (come è possibile vedere nella figura in appendice), è una macchina definita a transizioni con variabili. Per tutta la gestione di questa macchina si è realizzato, come già detto in precedenza, una classe apposita, figlia di *StateMachine*, chiamata *BeaconConnection*. Il numero maggiori di stati presenti è identificabile nel maggior numero di azioni necessarie per la connessione al sensore. Difatti la connessione si articola nella lettura dei *servizi* offerti, intesi come i sensori che sono disponibili all'interno del beacon, l'accessione del sensore desiderato e la lettura e l'invio dei dati quando disponibili. Per quanto riguarda i servizi vanno fatte alcuni chiarimenti. Per come è stato realizzato il beacon, ogni sensore corrisponde ad un sensore, che è raggiungibile tramite un indirizzo UUID. Questo indirizzo però, nonostante identifichi univocamente un servizio, da solo non è sufficiente per eseguire tutte le operazioni di cui si parlerà nel seguito. Ognuno di questi servizi si potrebbe rappresentare come una terna di indirizzi UUID, che differiscono per una cifra: il primo per identificare il servizio vero e proprio, il secondo per la lettura della caratteristica riferita ai dati, il terzo per la lettura della caratteristica riferita alla configurazione del sensore. Con caratteristica si intende un insieme di byte: nel caso dei dati indica il valore letto dal sensore, mentre nel caso della configurazione è importante in quanto è possibile modificarne una parte per attivare alcune funzionalità del sensore, come la sua accensione o spegnimento, oppure il fatto che invii i suoi dati. Alla luce della presenza di questa terna, si è deciso di realizzare una nuova classe, chiama *BeaconService*, all'interno del quale descrivere perfettamente la situazione e salvare i servizi di interesse. Si può facilmente notare una somiglianza le coppie di stati (2,3) e (5,6), sia a livello di comportamento che a livello di metodi, vengono usati gli stessi, a cui però sono passati parametri diversi. L'unica differenza sta nell'ordine in cui si richiamano, che nel secondo caso è l'opposto rispetto al primo, per preservare la simmetria tra i due. Qui di seguito viene descritta la funzione di ogni stato:

- Stato 0: in questo stato viene effettuata la connessione al sensore (identificato come un oggetto *BluetoothDevice* nel codice), appoggiandosi su di un metodo di *GattLeService*, chiamato *execute*. Questo, oltre ad inizializzare alcuni elementi d'interesse per quella classe, contiene anche il comando che permette di iniziare la connessione tramite l'interfaccia, ovvero il metodo *connectGatt*. Tramite questo si può dire che ufficialmente iniziare la connessione e subito viene invocato il primo metodo dell'interfaccia: *onConnectionChange*, che come suggerisce il nome, notifica il cambio di stato della connessione, che passa ad attivo. Avendo il

dovere di implementare i metodi, all'interno di questo è stato inserito anche la lettura dei *servizi*, identificati da un indirizzo UUID, forniti dal *sensortag*. Questa azione implica l'invocazione di un altro metodo dell'interfaccia, ovvero *onServicesDiscovered*, il quale si occupa di trovare tutti i servizi disponibili offerti dal *sensortag*.

- Stato 1: dopo aver salvato i servizi offerti in un'apposita struttura dati (*BeaconService*) in modo tale che poi sia più facile avere accesso ai loro indirizzi UUID, viene utilizzato un metodo, che scorre ciclicamente tutti i servizi d'interesse per l'applicazione (sensore di temperatura, sensore di luminosità, barometro ed accelerometro), fra quelli trovati nello stato precedente.
- Stato 2: una volta determinato il sensore da cui si vogliono leggere i dati, questo va acceso. Per questo scopo, è stato creato un metodo, denominato *changeStateSensor*, il quale si occupa di inviare una particolare maschera di due byte al sensore, in modo tale che questo venga acceso. Anche in questo caso risulta necessario appoggiarsi ai metodi dell'interfaccia presente in *GattLeService*, tramite *onCharacteristicWrite*. Questa infatti viene invocata nel momento in cui si scrive la maschera ed al suo interno viene inviato il messaggio di *ACKNOWLEDGE* che comunica la fine dello stato.
- Stato 3: nel momento in cui il sensore viene acceso, non basta per leggere le sue misurazioni. Risulta infatti necessaria un'ulteriore fase, molto simile alla precedente, nella quale viene comunicato al *sensortag* che deve permettere l'invio di notifiche di un sensore (in questo caso quello attivato nello stato precedente) nel momento in cui cambiano i dati contenuti nella sua caratteristica (l'insieme di byte che indicano il valore di un sensore). Anche in questo caso viene invocato un metodo, *changeNotificationState*, il quale scrive una maschera di byte in una certa struttura del *sensortag*, chiama descrittore. Ci si appoggia anche in questo caso su di un metodo dell'interfaccia, ovvero *onDescriptorWrite*, nella quale viene anche inviato il messaggio di *ACKNOWLEDGE*.
- Stato 4: in questo momento, dopo che nei precedenti stati si è impostato il *sensortag* per misurare ed inviare dati, si leggono effettivamente i dati del sensore prescelto. Per prima cosa vengono inizializzate le strutture dati dove immagazzinare i dati. Per cercare di compensare eventuali situazioni straordinarie, dove il sensore potrebbe fornire dati non veritieri, si è deciso di effettuare più misurazioni dello stesso sensore. Il numero di misurazioni è infatti un attributo della classe *GattLeService*, in modo da mantenere il sistema più parametrico possibile. Anche per questo motivo la scelta in questo frangente si è rivelata delicata. Da un lato l'impostazione di tutta l'applicazione fortemente parametrica e dall'altra la non omogeneità dimensionale dei dati (si pensi ad esempio che la temperatura viene rappresentata tramite uno scalare, mentre l'accelerometro è identificato da un vettore di tre elementi). Per mantenere il sistema più dinamico si è deciso quindi di realizzare un array (in quanto il numero di servizi letti è noto a priori, quattro) di *ArrayList*, a loro volta contenente valori di

tipo Double. Grazie alla dinamicità dell'ArrayList è possibile utilizzare la stessa struttura dati per immagazzinare dati scalari e vettori. Dopo aver impostato la struttura dati, si setta al valore true un booleano, che indica che i dati vengono letti. All'interno della definizione dello stato non è presente nessun altro metodo. Poiché nella situazione descritta il sensore scelto è acceso e le notifiche anche lo sono, i dati vengono misurati ed inviati all'applicazione. Per fare ciò si sfrutta un altro metodo presente nell'interfaccia: *onCharacteristicChange*. Come già spiegato in precedenza, il *sensortag* interpreta il cambiamento di alcuni byte come il segnale che è stato letto un dato. Al suo interno sono state gestite tutte le esigenze legate alla lettura dei dati, ed infine l'invio del messaggio di ACKNOWLEDGE.

- Stato 5: questo stato rappresenta il duale dello stato 3. Qui vanno infatti spente le notifiche che il *sensortag* invia all'applicazione quando viene letto un dato. Ci si comporta esattamente come prima, utilizzando lo stesso metodo (*changeNotificationState*) che permette di scrivere una maschera di byte (logicamente in questo caso con valori diversi rispetto allo stato 3).
- Stato 6: questo stato rappresenta il duale dello stato 2. Mentre prima il sensore è stato acceso, in questo caso viene spento. Si utilizza lo stesso metodo utilizzato per l'accensione (*changeStateSensor*), al quale sono però passati parametri diversi per ottenere il comportamento opposto. Questo metodo permette di scrivere una maschera di byte (con valori però diversi rispetto al caso precedente).
- Stato 7: in questo stato, qualora siano stati letti i dati dai sensori, si calcola la media matematica per tutte le misurazioni che fanno riferimento allo stesso sensore.
- Stato 8: all'interno di questi stato, l'ultimo, la connessione è quasi terminata, ma prima che ciò accada vanno effettuate le ultime azioni: i dati letti vengono impacchettati in file JSON ed inviati al server; viene cancellato il *broadcastReceiver* creato per ricevere i messaggi, in quanto ormai non è più necessario ed infine si invia un messaggio di termina allo scanner, il quale può riprendere l'esecuzione della sua macchina a stati.

#### 4.3.4 - Interruzione della connessione

Un altro elemento su cui si è ragionato molto riguarda l'interruzione anticipata dello scan o della connessione. Può capitare infatti che questi vengano interrotti prima della loro naturale conclusione. Ad esempio lo scan, che in condizioni di default è nella configurazione Normal, può dover cambiare per passare ad un'altra delle due modalità di funzionamento, come già spiegato sopra. D'altra parte, il cambio di scan può portare anche alla terminazione anticipata della connessione. L'approccio a questa problematica è stato affrontato per evitare che, data la presenza delle callback e dei loro thread in queste macchine a stati, la non corretta chiusura di una

delle due portasse alla presenza contemporanea di due scan o di due connessioni, con la conseguente sovrapposizione degli eventi e di tutto ciò che ne deriva. Per seguire questo approccio, fin dalla creazione delle macchine a stati e delle possibili transizioni di stato, si è tenuto conto di un'eventuale interruzione asincrona di scan e/o connessione. Come si può vedere nei sequence diagram in appendice, si è lavorato in modo che la chiusura non fosse brutale, bensì questa viene dettata dal semplice cambio di valore di alcune variabili di controllo. In questo modo, quando si va a valutare lo stato futuro che deve essere assunto dalle macchine a stati, queste vadano naturalmente verso la loro conclusione, in maniera trasparente per l'utente. Molto importante risulta, in questa fase, lo spegnimento di tutti i broadcastReceiver, per evitare che continuino ad arrivare i messaggi, e soprattutto la chiusura della connessione con l'apposito metodo delle librerie Android. Solo a questo punto, dopo aver sistemato questi dettagli, è possibile andare a cambiare l'istanza contenente l'oggetto scanner.

## 5 - Gestione delle emergenze

La gestione delle emergenze rappresenta l'elemento centrale di tutto il progetto. Fin dall'analisi dei requisiti e delle specifiche è stato presentato come la parte più delicata, per poter garantire la sicurezza degli utenti e gestire in maniera coordinata ed efficiente i soccorsi. Nello specifico, per gestione dell'emergenza, si intende la procedura per affrontare una situazione di pericolo, comunicandolo all'utente ed indicandogli la via di fuga più breve per mettersi in sicurezza. Dai primi approcci mossi verso il problema sono emersi parecchi dubbi. Apparentemente si possono rintracciare in questa fase progettuale molte componenti già presentate, in quanto usate in altre parti del progetto: basti pensare solamente a tutta la mole di oggetti deputati alla comunicazione con il server, alla gestione dello scanner dei beacon o a tutti quelli che sono figli di *SharedStorage*.

### 5.1 - Problemi legati al Server Rest

Risulta evidente come la gestione delle emergenze vada affrontata considerando un aspetto fondamentale: questa fase progettuale richiede il contributo di componenti sia lato server e lato client. La prima si è presentata subito come un'incognita, nonostante a prima vista si possa pensare che non si discosti molto dal modello di comunicazione già proposto, ovvero l'architettura a servizi di tipo REST. Il problema deriva dal fatto che questa è concepita in modo tale che l'interazione fra le componenti dell'architettura sia di tipo *pull-based*. Questo significa quindi che il server ha l'unico compito di rispondere alle richieste del client, senza la possibilità di inviare autonomamente messaggi ai client. Questo modello funziona in maniera ottimale per tutte le altre tipologie di interazioni, ma in questo preciso ambito rischia di essere inefficiente. Si è pensato, in un primo momento, di rimanere fedeli al modello, cercando soluzioni che ben si conciliassero con l'architettura REST. Ad esempio, una prima idea ragionava sulla possibilità di far inviare periodicamente dall'applicazione una richiesta sullo stato della struttura al server, in modo tale da avere una comunicazione tempestiva. Al momento di formulare ipotesi sul periodo di tempo di attesa fra due richieste consecutive, ipotizzando tempi molto vicini fra loro, ci si è resi conto di come questa soluzione, oltre a rappresentare un'evidente forzatura, nascondesse un'incognita molto rischiosa per tutto il sistema. Inviare periodicamente una richiesta, avrebbe potuto generare un overhead non indifferente, in quanto la maggior parte dei messaggi scambiati tra client e server sarebbero stati richieste di questo tipo. Alla luce del fatto che un'emergenza, come già detto in precedenza, rappresenta una situazione extra ordinaria, si sarebbe rischiato sovraccaricare il server di lavoro inutile.

## 5.2 - Server emergenze

A questo punto, si è deciso di rendere più complessa la struttura del server, affiancando un secondo server a quello REST, con l'unico scopo di inviare notifiche di emergenze agli utenti tramite documenti JSON, con opportuna struttura (il messaggio contiene un codice che identifichi la stanza, un codice che individui il piano, uno che indichi che tipo di emergenza è presente ed infine uno per identificare univocamente l'emergenza). Volendo evitare che queste due componenti fossero totalmente slegate fra loro, si è utilizzato il database presente nel server come una sorta di memoria condivisa: entrambi hanno infatti la possibilità di accedere a questo, perciò è possibile far comunicare queste due sezioni. Non avendo l'onere di progettare un sistema che rilevi automaticamente la presenza di una situazione pericolosa, si è supposto che sia presente un'autorità, la quale dopo aver interpretato i dati ambientali dei sensori, comunichi agli utenti il pericolo. Si è realizzata quindi un'applicazione lato server, invia la notifica a tutti gli utenti online in quel momento. Da questo aspetto è possibile notare come aver creato quella zona di memoria condivisa fosse un espediente utile ai fini pratici. L'applicazione difatti scorre la lista degli utenti che hanno inviato in precedenza la loro posizione, i quali vengono considerati online. Alla luce del bisogno di tempestività, per evitare che la rete nel momento del pericolo sia intasata da richieste, avendo la possibilità di inviare a determinati utenti il messaggio di segnalazione del pericolo, sono stati pensati e realizzati anche meccanismi per tenere il più possibile aggiornata questa base di dati: ad esempio, nel momento in cui l'utente spegne l'applicazione (senza forzare la chiusura) viene inviato un messaggio al server, contenente una richiesta di tipo delete, per fare in modo che venga cancellato dall'apposita tabella.

## 5.3 - Soluzione lato Requester

Per quanto riguarda la progettazione lato client, anche in questo caso, ciò realizzato per la ricezione delle notifiche mostra alcune similitudini con alcuni elementi progettati per la comunicazione con il server, ma comunque si possono notare delle differenze. Si consideri il fatto che per effettuare le richieste al server, la gestione di ciò ricevuto come risposta è insito nei metodi, senza bisogno di elementi supplementari. In questa situazione invece, non essendoci nessuna richiesta, bensì la ricezione di una notifica in maniera asincrona rispetto al funzionamento dell'applicazione, si è dovuto costruire un tipo di oggetto particolare: *GetReceiver*. Questo, tralasciando gli elementi per effettuare la connessione vera e propria, similari a quanto già discusso, contiene al suo interno un *thread*, deputato a controllare esclusivamente la ricezione o meno di una notifica. Tutta questa struttura comunque si appoggia molto su un'altra soluzione presentata in *SharedStorage*:

l'istanza della classe viene aggiunta all'*arrayList* Notification. In questa maniera, come già spiegato, ogni qualvolta viene scorso il vettore, vengono richiamate gli *update* e le *retrieve* di tutti gli elementi registrati. La gestione dell'emergenza, a livello client, è contenuta effettivamente all'interno dell'*update* della classe *GetReceiver*. Una volta che il *thread* identifica la ricezione di una notifica, come prima cosa questa subisce un'operazione di *parsing*, per poterla interpretare. La notifica ricevuta può comunicare l'inizio o il termine di una situazione di emergenza. Nel primo caso il messaggio contiene quattro valori, descritti in precedenza, mentre nel secondo è un messaggio vuoto. Nel caso in cui il server comunichi una situazione di pericolo, come prima cosa viene impostato un apposito flag, denominato *emergency* e contenuto nella classe *MainApplication* al valore *true*. Nel effettuare questa operazione viene anche sospeso lo *scan*, con le modalità già raccontate nel paragrafo riferito allo scanner. In questa maniera, una volta terminata tutta questa procedura, è possibile inizializzare un nuovo scanner, impostato in modalità emergenza e contemporaneamente cambiare l'*activity*, in modo che si visualizzi direttamente la schermata contenente le mappe. Questa viene creata, impostando come obiettivo da raggiungere le vie di fuga e indicando anche, con determinati simboli negli appositi punti, eventuali pericoli, come ad esempio le fiamme. Tutta questa sequenza non è interrompibile, che dal momento della ricezione della notifica, perde il controllo dell'applicazione: una volta aperta la schermata delle mappe, non è possibile navigare liberamente per le diverse schermate, in modo che l'utente prenda sul serio il pericolo e si muova velocemente verso le vie di fuga. Tutto ciò però si basa sul presupposto però che l'utente tenga costantemente l'applicazione accesa in *foreground*, ovvero aperta sullo schermo. Questa situazione risulta piuttosto inverosimile, considerando che un utente potrebbe trovarsi ad utilizzare il suo smartphone nelle più disparate maniere. Per ovviare a questo problema, si è implementato un sistema di notifiche *push*, in modo che un utente riceva una segnalazione del pericolo. Cliccando sulla notifica, viene aperta l'applicazione, direttamente sulla schermata delle mappe, nella situazione descritta in precedenza. Va sottolineato, comunque, che tutta questa struttura si basa sull'ipotesi che l'utente abbia il proprio dispositivo collegato alla rete, con l'applicazione sempre accesa. Per quanto invece riguarda la ricezione di una notifica inviata per comunicare il termine di un'emergenza, il funzionamento è descrivibile in maniera del tutto simile al caso precedente: viene interrotto il ciclo di *scan*; ne viene inizializzato uno nuovo, con la modalità di funzionamento di default ed infine viene mostrata la schermata in cui ci si trovava in precedenza.



## 6 - Gestione utente

Un'altra componente progettuale per cui vale la pena spendere qualche parola riguarda la gestione dell'utente. Questa entità identifica un soggetto che utilizza l'applicazione sul proprio dispositivo. Egli rappresenta l'elemento cardine di tutto il progetto: l'applicazione, come già specificato, ha come scopo quello di coordinare, in maniera centralizzata, il comportamento delle persone all'interno di una struttura. Questo aspetto progettuale, come molti altri, risente molto dell'architettura client/server adottata. Alla luce di ciò, per avere una visione più completa della problematica, si devono valutare due livelli per quanto riguarda la gestione dell'entità utente: lato client e lato server. Prima di analizzarle in dettaglio, va preventivamente sottolineata una distinzione per quanto riguarda le due modalità di funzionamento dell'applicazione. Qualora un individuo decidesse di lavorare in modalità offline, la componente legata al server si verrebbe a perdere, rendendo l'applicazione un sistema stand alone. La distinzione è stata espressamente richiesta dagli stakeholder, considerando che comunque l'applicativo può rappresentare un ottimo strumento per potersi muovere dentro una struttura sconosciuta. In questa modalità infatti, viene eliminata ogni tipo di comunicazione, ma comunque è permessa all'utente solo la possibilità di utilizzare le mappe per orientarsi dentro una struttura. Va comunque detto che, qualora molte persone decidessero di far lavorare l'applicazione in questo modo, si verrebbero a creare varie falle nel reticolo dei nodi (i sensori), con il rischio concreto che un pericolo non venga segnalato tempestivamente. Si raccomanda perciò, se possibile, di far lavorare l'applicazione in modalità online.

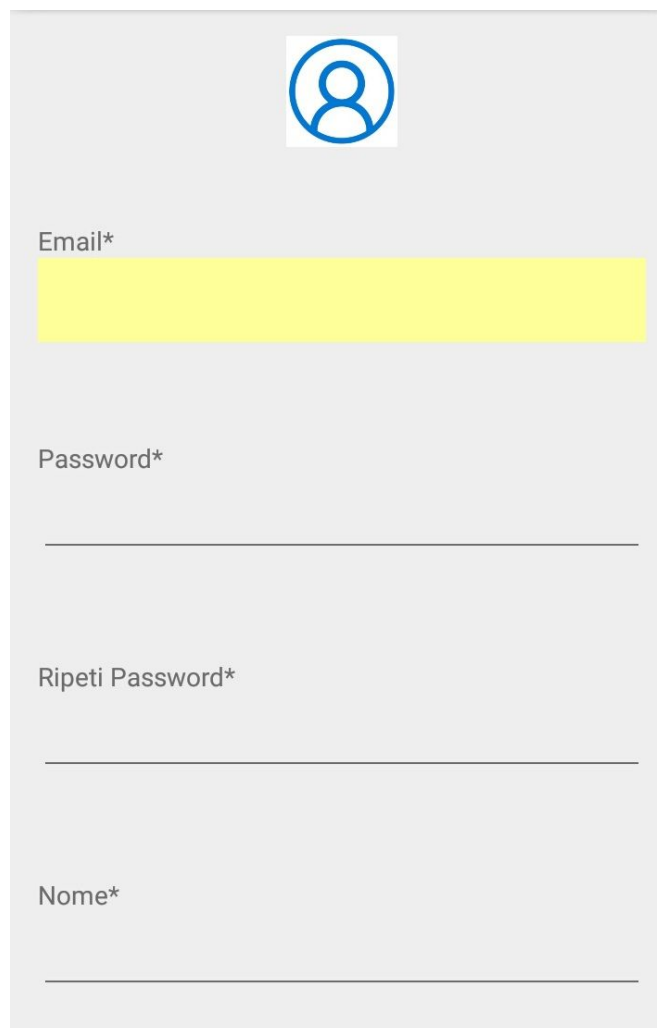
### 6.1 - Tipologie di utenti

Tornando alla gestione dell'utente, la prima componente descritta è quella lato client. Presupponendo che l'applicazione sia stata avviata in modalità online, inizialmente l'utente viene visto come un utente sconosciuto chiamato utente *guest*. Questa situazione si presenta per il fatto che all'utente è data la possibilità di iscriversi ad una piattaforma, gestita nel server. Iscriversi non comporta particolari privilegi a livello di funzionamento dell'applicazione: le funzionalità rimangono immutate sia per l'utente autenticato che per il *guest*. Ciò apparentemente può sembrare un errore logico, poiché si potrebbe essere portati a credere che questa distinzione, non aggiungendo alcun privilegio a chi si autentichi, rappresenti un inutile fonte di ridondanza. La discriminazione fra queste due tipologie di utenti, però, assume un'importanza non indifferente nel momento in cui si presenti un'emergenza ed utente rimanga bloccato. In questo caso infatti i soccorsi disporrebbero di informazioni aggiuntive rispetto al caso in cui si intervenga muovendosi al buio.

L'utente guest, senza considerare l'uso delle mappe che rappresenta una funzionalità comune per entrambi i casi, ha di fronte due opportunità: può decidere di iscriversi alla piattaforma o, qualora già disponga di un profilo, effettuare il login.

## 6.2 - Iscrizione

Nel primo caso viene presentata una vista con dei campi, dove è possibile inserire i propri dati. Di questi campi solamente cinque risultano obbligatori (nome, cognome, email, password e codice fiscale): il mancato inserimento di un elemento al loro interno comporta il fallimento della procedura. Sono stati comunque implementati dei controlli (contenuti nella classe *FormControl*) sulle stringhe inserite nei vari campi, per evitare che i dati non fossero corretti.

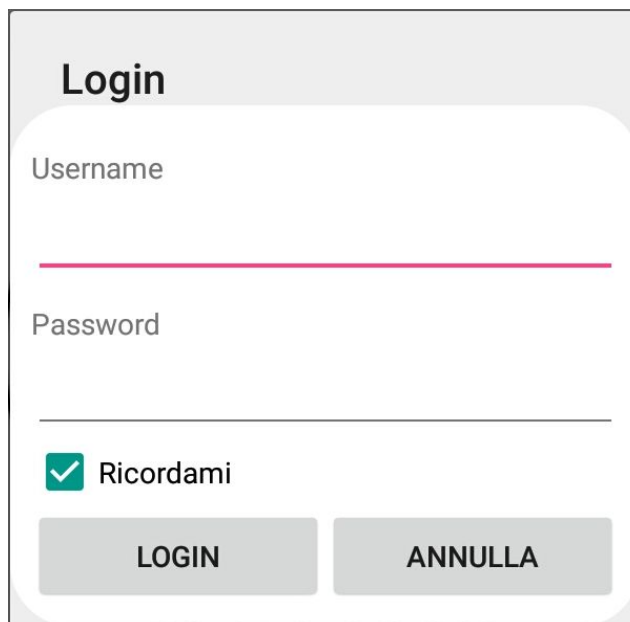


The image shows a user registration form on a light gray background. At the top center is a blue circular icon containing a white silhouette of a person. Below the icon, the form contains four labeled input fields, each with an asterisk indicating it is required. The first field is labeled 'Email\*' and has a yellow background. The second field is labeled 'Password\*'. The third field is labeled 'Ripeti Password\*'. The fourth field is labeled 'Nome\*'. Each field is represented by a horizontal line.

Nel momento in cui l'utente decide di completare la procedura di iscrizione, i campi vengono controllati e viene segnalato un eventuale errore.

## 6.3 - Login

Qualora un individuo disponga già di un proprio account, ha la possibilità di effettuare il login, inserendo il proprio indirizzo email e la password.

A login form with a light gray header containing the title "Login". Below the header is a white rounded rectangle containing the form fields. There are two text input fields: the first is labeled "Username" and the second is labeled "Password". Below the password field is a checkbox with a green checkmark icon and the text "Ricordami". At the bottom of the form are two buttons: "LOGIN" and "ANNULLA".

Questi dati vengono inviati al server, il quale dopo aver effettuato i debiti controlli, invia il responso (positivo o negativo che sia) dell'operazione all'applicazione, tramite un messaggio all'applicazione. Durante questa procedura, è possibile comunque rendere permanente l'accesso, selezionando l'apposito segno di spunta. In questa maniera, ad ogni nuova accensione non risulta necessario effettuare il login. Per implementare questa funzionalità, si è sfruttato un oggetto inserito all'interno delle *API* di Android: *SharedPreferences*. Tramite questo è infatti possibile salvare alcuni dati in maniera permanente all'interno del dispositivo. Il meccanismo si basa sul fatto che, nel momento in cui l'utente effettua con successo l'accesso alla piattaforma, vengono scritti i dati dentro le *preferences*. Ad ogni nuova accensione dell'applicazione, viene controllato, dentro di queste, la presenza dell'informazione riguardante l'utente e, qualora venga trovata, tutti gli elementi che dipendono da essa vengono opportunamente modificati (ad esempio le voci presenti nel menu).

## 6.4 - Funzionalità

Supponendo che il login sia stato effettuato, l'utente ha la possibilità di effettuare il logout o di modificare i propri dati, tramite una vista del tutto simile a quella usata per l'iscrizione (viene usata infatti la stessa *activity*). L'utente ha la possibilità di decidere

una di queste funzionalità tramite un apposito menù, contenuto nella home dell'applicazione, dove sono presenti (per entrambi i casi) tre voci disponibili: mappe, per muoversi all'interno di una struttura, login/logout, iscrizione/modifica informazioni (rispettivamente in modalità guest e in modalità autenticazione). Ognuna di queste ultime due coppie di operazioni, per poter funzionare, necessita di comunicare con il server, tramite lo scambio di messaggi. A questo punto, è spontaneo domandarsi in che maniera l'utente venga individuato, in riferimento alla componente server del sistema. Ebbene, all'interno del server l'utente viene considerato secondo due diverse accezioni: l'utente visto come un account e l'utente visto come individuo presente in quel preciso momento in una struttura. Come già descritto in precedenza, in questo lato del sistema, sono presenti alcuni database, per poter gestire in maniera ottimale tutta la mole di informazioni che gravitano intorno all'intera applicazione. Fra questi, due vengono deputati alle due funzioni descritte: in uno vengono salvati i dati riguardanti gli utenti iscritti, in modo da poter gestire la piattaforma, mentre nel secondo, più importante rispetto all'altro, sono presenti i dati riguardanti la situazione attuale di una struttura. Questo si può configurare come un'istantanea del sistema, nella quale l'utente, dopo aver inviato la propria posizione, viene registrato utilizzando il suo indirizzo IP (in modo anche da potergli inviare eventuali notifiche di pericolo), la sua posizione, tramite il beacon a quale è legato ed infine due stringhe che lo identifichino (nome e cognome, qualora il messaggio di aggiornamento arrivi da un utente, due stringhe "guest" altrimenti). Per fare in modo che questo database non contenga più *tuple* riferite allo stesso utente, sono stati implementati meccanismi che permettono di aggiornarlo costantemente, ad esempio l'invio di un messaggio per cancellare la tupla nel momento in cui l'applicazione viene spenta. Ogni volta che un utente invia un messaggio per segnalare la propria posizione, il database viene aggiornato con l'opportuna tupla.

## 7 - Metriche

A margine della progettazione, sono state calcolate metriche, un insieme di indicatori, per verificare la bontà delle scelte progettuali. Per questo scopo è stato utilizzato un plugin di Android Studio, chiamato MetricsReloaded. Fra le tante disponibili, si è scelto di soffermarsi solo su alcuni valori:

- WMC, indica la somma pesata dei metodi implementati in una classe.
- RFC, indica il numero di metodi appartenenti ad una classe e la relativa
- V(G), indica la complessità ciclomatica, ovvero, ipotizzando di modellizzare l'evoluzione di un programma secondo un grafo, il numero di cammini linearmente indipendenti presenti in questo.
- LCOM, indica il grado di coesione e diversità tra i metodi di una classe.
- CBO, indica il numero di relazione di dipendenza in uscita dalla classe.
- DIT, indica, alla luce della presenza di relazioni di ereditarietà modellizzate tramite un albero, la massima profondità di questo.
- NOC, indica, alla luce della presenza di relazioni di ereditarietà, il numero di sottoclassi dirette, legate alla classe padre.
- I, indica l'instabilità del package.
- A, indica il numero di classi astratte o interfaccia contenute in un package, rispetto a tutte le classi che contiene.

Method metrics	Class metrics	Interface metrics	Package metrics	Module metrics
package ▲	A	I	v(G)avg	v(G)tot
application	0,00	0,67	1,75	49
application.beacon	0,00	0,65	2,38	209
application.comunication	0,00	0,43	1,50	21
application.comunication.http	0,00	0,56	6,82	75
application.comunication.message	0,00	0,00	7,33	22
application.maps.components	0,00	0,07	1,07	32
application.maps.grid	0,00	0,33	3,50	35
application.sharedstorage	0,20	0,20	1,08	14
application.user	0,00	0,45	2,00	62
application.utility	0,50	0,00	1,92	23
application.validation	1,00	0,00	2,00	16
com.core.progettoingegneriadelsoftware	0,00	0,80	3,70	237
<b>Total</b>				<b>795</b>

Method metrics	Class metrics	Interface metrics	Package metrics	Module metrics	Project metrics			
class								
	CBO	DIT	LCOM	NOC	OCavg	RFC	WMC	
application.beacon.BeaconConnection	8	2	1	0	4,00	63	48	
application.beacon.BeaconScanner	14	2	3	0	2,04	75	55	
application.beacon.BeaconService	3	1	4	0	1,09	17	12	
application.beacon.GattLeService	3	1	5	0	2,08	54	54	
application.beacon.LeDeviceListAdapter	2	1	1	0	1,43	27	10	
application.beacon.Setup	2	1	4	0	2,33	10	14	
application.comunication.http.DeleteRequest	2	2	1	0	6,00	25	6	
application.comunication.http.GetReceiver	3	2	1	0	1,67	9	5	
application.comunication.http.GetRequest	2	2	1	0	4,00	25	4	
application.comunication.http.HttpReceiverThread	9	2	2	0	4,00	41	16	
application.comunication.http.PostRequest	2	2	1	0	6,00	32	6	
application.comunication.http.PutRequest	2	2	1	0	6,00	32	6	
application.comunication.message.MessageBuilder	4	1	1	0	7,00	10	7	
application.comunication.message.MessageParser	2	1	1	0	2,50	14	5	
application.comunication.ServerCommunication	13	1	2	0	1,14	30	16	
application.MainApplication	28	1	4	0	1,57	77	44	
application.maps.components.Floor	6	1	6	0	1,18	21	13	
application.maps.components.Node	3	1	2	0	1,00	4	4	
application.maps.components.Notify	5	1	4	0	1,00	10	10	
application.maps.components.Room	4	1	4	0	1,00	5	5	
application.maps.grid.TouchImageView	2	4	2	0	2,56	27	23	
application.maps.grid.TouchImageView.ScaleListener	1	2	2	0	2,50	7	5	
application.sharedstorage.Data	7	1	2	0	1,00	4	2	
application.sharedstorage.DataListener						2		
application.sharedstorage.Notification	6	2	1	0	1,00	5	3	
application.sharedstorage.SharedData	6	1	1	2	1,33	6	4	
application.sharedstorage.UserPositions	6	2	2	0	1,00	6	5	
application.user.UserHandler	8	1	4	0	2,00	57	36	
application.user.UserProfile	3	1	11	0	1,00	13	13	
application.utility.CSVHandler	3	1	2	0	2,00	30	14	
application.utility.StateMachine	2	1	3	2	1,00	5	5	
application.validation.FormControl	2	1	8	0	2,00	10	16	
com.core.progettoingegneriadelsoftware.ActivityMaps	10	10	2	0	1,80	60	27	
com.core.progettoingegneriadelsoftware.ExampleInstrumentedTest	0	1	1	0	1,00	4	1	
com.core.progettoingegneriadelsoftware.ExampleUnitTest	0	1	1	0	1,00	2	1	
com.core.progettoingegneriadelsoftware.FullScreenMap	14	10	5	0	2,88	87	46	
com.core.progettoingegneriadelsoftware.Home	12	10	5	0	2,69	87	43	
com.core.progettoingegneriadelsoftware.InformationsHandler	9	10	6	0	2,67	73	72	
com.core.progettoingegneriadelsoftware.WelcomeActivity	8	10	3	0	2,92	61	38	
<b>Total</b>							<b>694</b>	
Average	5,68	2,55	2,89	0,11	2,05	28,90	18,26	

# Appendice

## A - MANUALE DI FUNZIONAMENTO

Tutto il materiale necessario è contenuto all'interno della cartella, in particolare ci saranno le seguenti programmi:

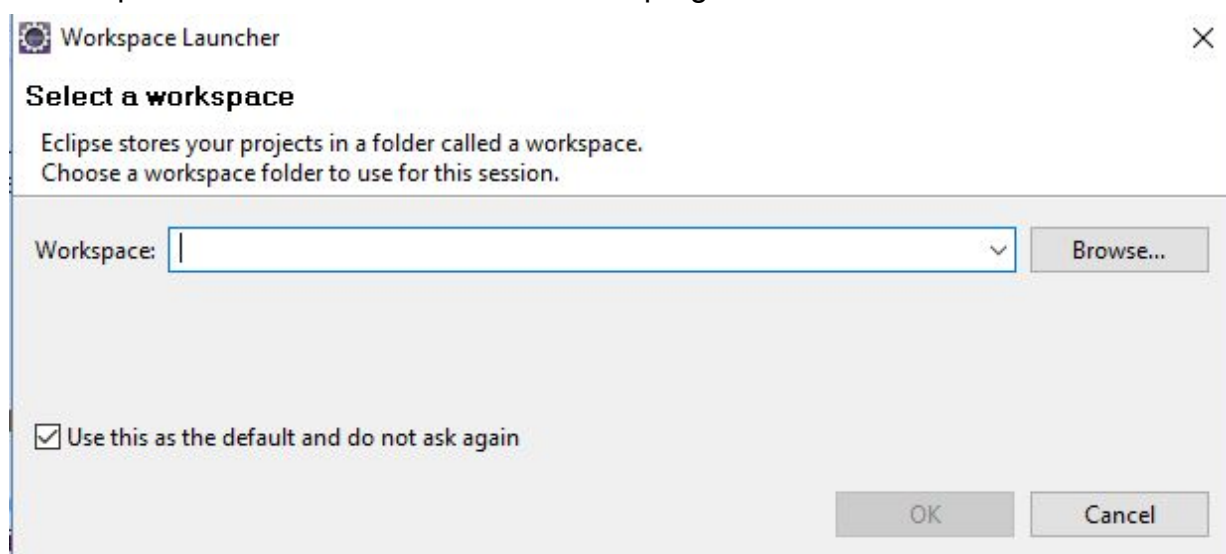
- Eclipse per la gestione del server;
- file .apk dell'applicazione da installare sul telefono;
- file batch Database per la gestione del database;
- eseguibile Java per la generazione delle notifiche;
- JDK da installare.

I primi quattro elementi sono inseriti all'interno della cartella *"ESEGUIBILI"*, mentre l'ultimo si trova dentro la cartella *"JDK"*. Prima di partire ricordarsi di scaricare tutto il contenuto in locale.

### A.1- Installazione dei tools e importazione del progetto

Innanzitutto si deve far partire il server, quindi il primo passo è aprire il programma Eclipse cliccando due volte sulla seguente icona:

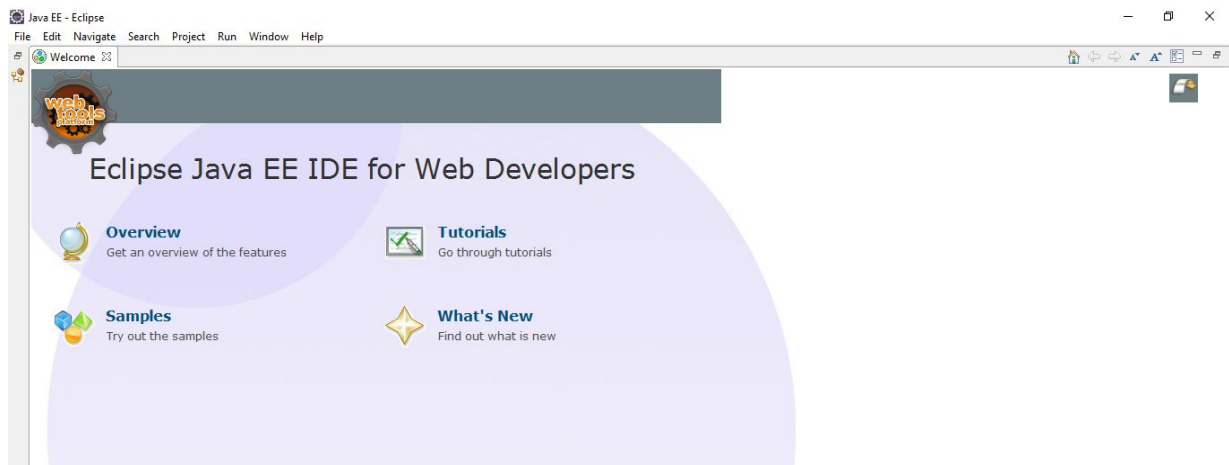
All'apertura potrebbe comparire la seguente schermata che chiede in quale cartella del computer dovranno essere memorizzati i progetti.



Una volta definita la cartella, grazie anche alla funzionalità Sfoglia, cliccare su OK.




Il programma è ora aperto, chiudere ora la tap welcome.



Il passo successivo è installare il pacchetto GitHub che serve per scaricare il progetto. Cliccare quindi su Help -> Eclipse Marketplace -> Digitare sul campo di ricerca “GitHub” e scaricare il prodotto seguente:

**EGit - Git Integration for Eclipse 4.6.0**




EGit is the Git integration for Eclipse. Git is a distributed versioning system, which means every developer has a full copy of all history of every revision of... [more info](#)

by Eclipse.org, EPL  
[egit](#) [jgit](#) [git](#) [dvcs](#) [scm](#) [vcs](#) [github](#)

★ 555    ➔ Installs: **476K** (1.589 last month)    [Update](#)    [Uninstall](#)

Ora ripetere la procedura digitando, questa volta, la stringa “Glassfish” e scaricare:

**GlassFish Tools**



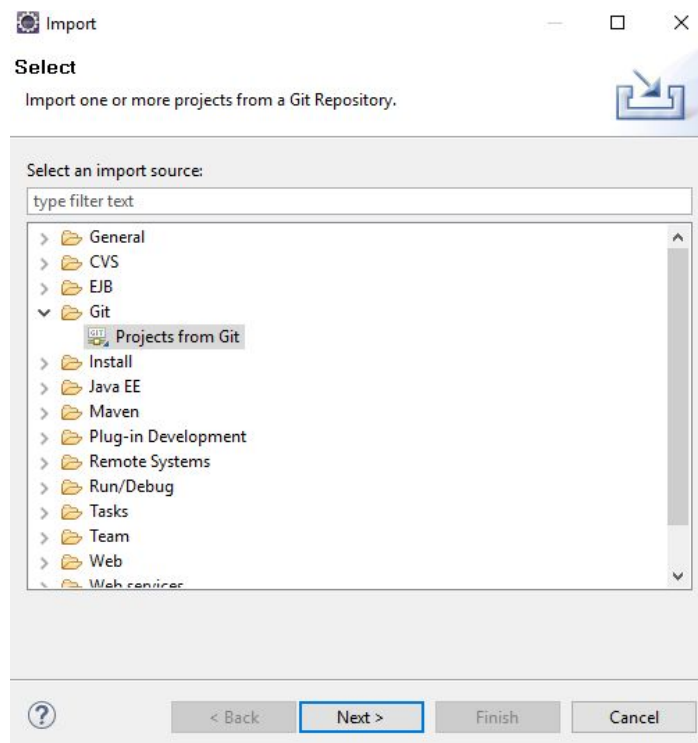
Tools for developing applications for GlassFish. Integrates with Eclipse Web Tools. Supports GlassFish 4 and 3.1. These features are also part of... [more info](#)

by Oracle, EPL

★ 15    ➔ Installs: **87,1K** (2.365 last month)    [Update](#)    [Uninstall](#)



Raccolto tutto il materiale necessario è possibile importare il progetto. Quindi seguire

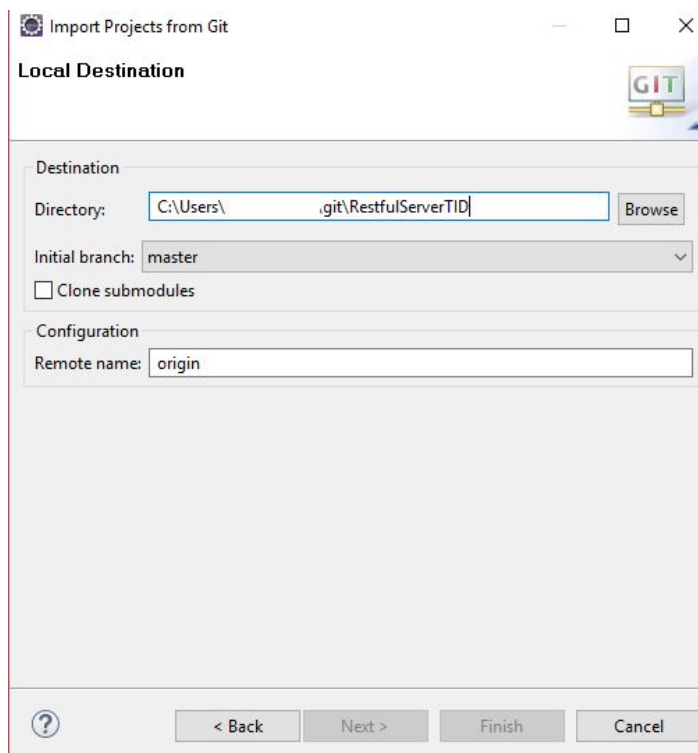


i seguenti passi: Cliccare su File -> Import -> Git -> Projects From Git

Ora selezionare la voce “Clone URI”, proseguire di nuovo su Next. Nella finestra che viene aperta si deve solamente incollare il seguente indirizzo

***<https://github.com/Soapfedan/RestfulServerTID.git>***

Proseguire avanti fino alla suddetta finestra in cui viene richiesto di inserire una cartella locale su cui scaricare l'intero server.



L'ultima cosa da fare è caricare il tutto, quindi proseguire sempre in avanti fino all'ultima Tab in cui ci è richiesto di premere il tasto Finish. Al termine della procedura il nostro progetto è stato correttamente installato sulla nostra macchina.

## A.2 - Configurazione ed avvio del server

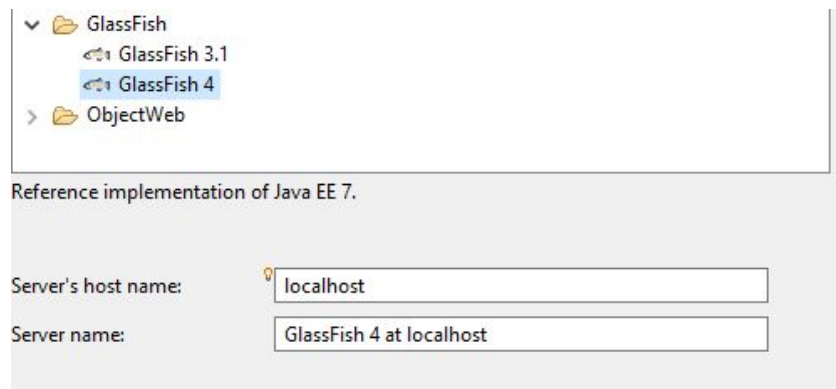
La configurazione del server parte dall'installazione del Java Development Kit (JDK) contenuto all'interno della cartella. Scegliere la versione adatta al sistema operativo della macchina, quindi una tra la 64bit (consigliata) o la 32bit (se non dovesse funzionare la prima scegliere questa versione che è universalmente compatibile). Seguire tutti i passi, cliccando sempre su avanti, fino alla fine.

La fase seguente riguarda la creazione del server. Quindi bisogna definire quale applicazione dovrà svolgere tale compito.

Per prima cosa si deve aggiungere la vista "Server" con cui si riesce a gestire il tutto. Per attivarla si deve cliccare su "Window" -> Show View -> Cliccare su **"Servers"**, che aprirà la seguente Tab:



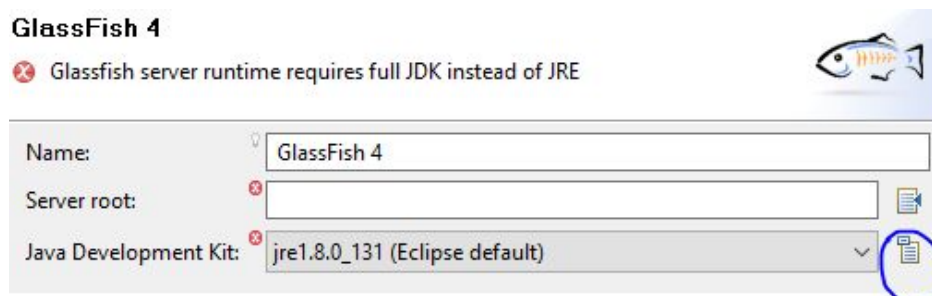
Cliccare quindi sulla scritta *"No servers are available. Click this link to create a new server."* Selezionare quindi la voce **"Glassfish 4.0"**



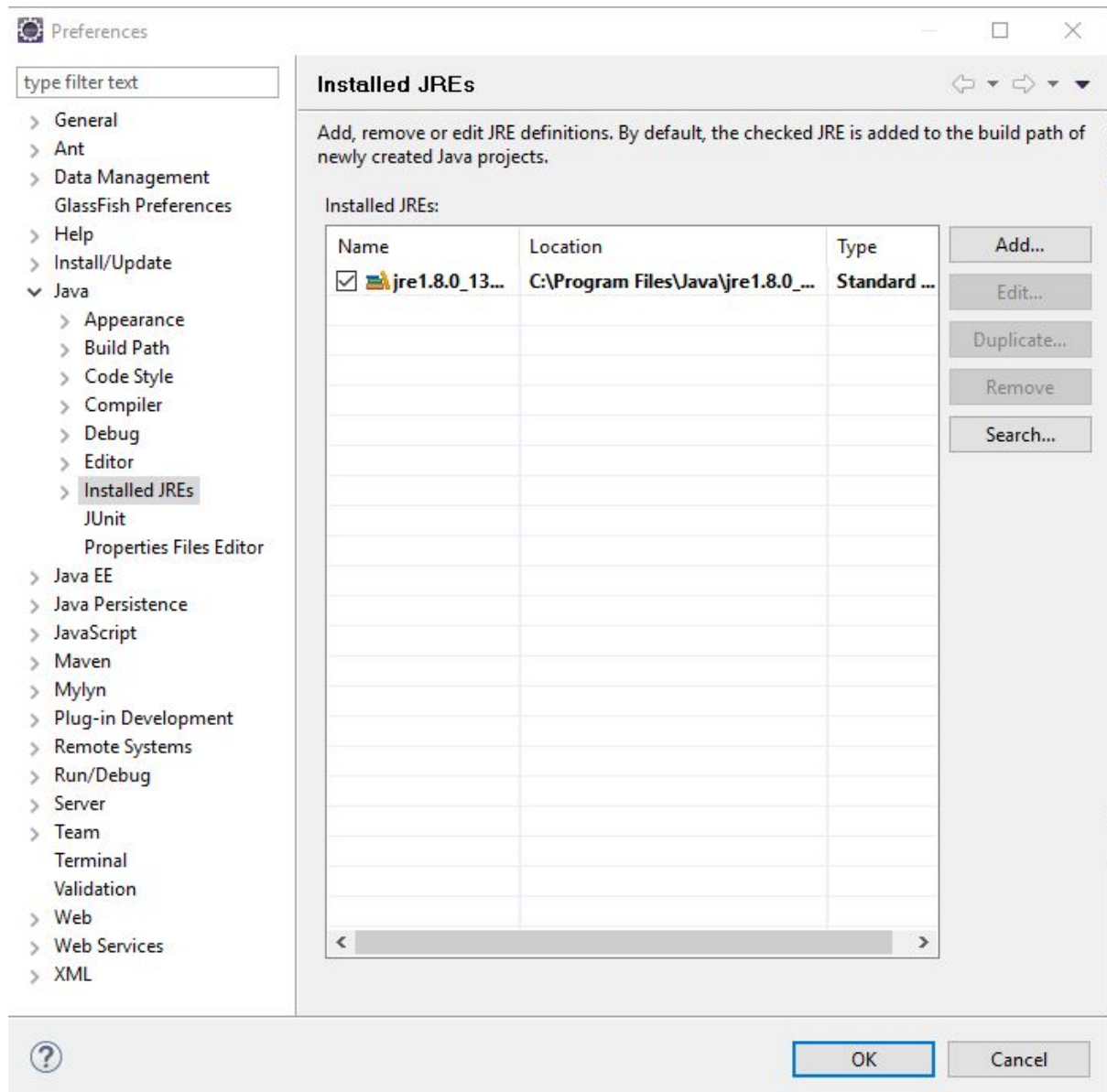
Proseguire su Next.

Bisogna aggiungere il pacchetto appena installato. Quindi cliccare sul pulsante Add -> Selezionare **"Standard VM"** e premere Avanti.

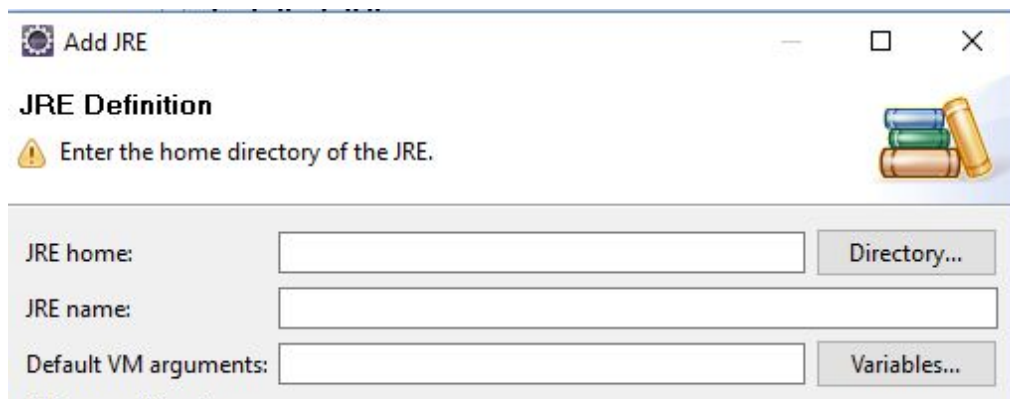
Ci si ritrova nella seguente schermata:



La prima cosa da fare è inserire la JDK installata all'inizio quindi cliccare sul pulsante cerchiato di blu che apre la seguente finestra:



Il passo seguente è aggiungerla tra gli elementi di ambiente di Eclipse, cliccare quindi su Add, selezionare la voce "Standart VM", proseguire ed ecco che appare la seguente scheda:



Qui bisogna indicare il percorso della libreria appena installata. Selezionare quindi la directory cliccando sull'omonimo pulsante, seguendo il percorso:

1. Questo-PC/Risorse del computer;
2. Disco Locale C:
3. Programmi (Selezionare questa voce se nell'installazione della JDK è stato scelto il file a 64bit, altrimenti selezionare la voce Programmi(x86));
4. Cartella Java
5. Selezionare la voce **"jdk1.8.0\_xx"**
6. Ora selezionata la cartella cliccare su OK

Terminata la procedura, il programma ha caricato tutto le librerie necessarie:

Ora cliccare sul pulsante finish,

Name	Location	Type
<input checked="" type="checkbox"/> jdk1.8.0_60 ...	C:\Program Files\Java\jdk1.8.0...	Standard ...
<input type="checkbox"/> jre1.8.0_131	C:\Program Files\Java\jre1.8.0_131	Standard VM

Spuntare la nuova voce e cliccare su OK.

Ora si ritorna sulla schermata di creazione del server quindi selezionare dal menù a tendina la JDK.

Nella seconda sezione si deve definire il percorso in cui è presente Glassfish, che si trova all'interno del pacchetto scaricato. Cercare quindi dentro la cartella **"Strumenti"** -> **"glassfish4"** -> **"glassfish"**.

Al termine ci sarà il seguente risultato:

Andare avanti, dove bisogna definire il path del dominio, cliccare quindi sul pulsante indicato di blu e spuntare la casella cerchiata di rosso:

**GlassFish 4**

Domain path must be specified

Domain path:

Admin name:

Admin password:

Debug port:

☒ Preserve sessions across redeployment

☐ Use JAR archives for deployment

Il percorso è comune al precedente, difatti si deve indicare la cartella già presente chiamata "DomainTID" che ha il seguente path:

*"Strumenti\glassfish4\glassfish\domains\DomainTid".*

Compiute entrambe le azioni proseguire avanti:

**New Server**

**Add and Remove**

Modify the resources that are configured on the server

Move resources to the right to configure them on the server

Available:		Configured:
RestfulServerTID	<div>Add &gt;</div> <div>&lt; Remove</div> <div>Add All &gt;&gt;</div> <div>&lt;&lt; Remove All</div>	

Selezionare quindi la voce "RestfulServerTID" e cliccare su "Add >", quindi si ottiene:

**Configured:**

RestfulServerTID

Infine basta terminare la procedura su "Finish".

L'ultima cosa è ora avviare il server cioè ritornare sulla tab "Servers" e avviarlo selezionando la voce a cliccando sul pulsante indicato.



La procedura richiede alcune decine di secondi al termine il server è pronto a partire.

### A.3 - Avvio database

L'ultima fase è avviare il database, quindi entrare all'interno della cartella Eseguibili, fare doppio click sul programma "Database"

Ora aperta la schermata seguente digitare il comando "*start-database*"

```
C:\> asadmin - collegamento
Use "exit" to exit and "help" for online help.
asadmin> start-database
```

Al termine della procedura, chiudere la finestra.

Il server ora è pronto a lavorare ed interagire con l'applicazione.

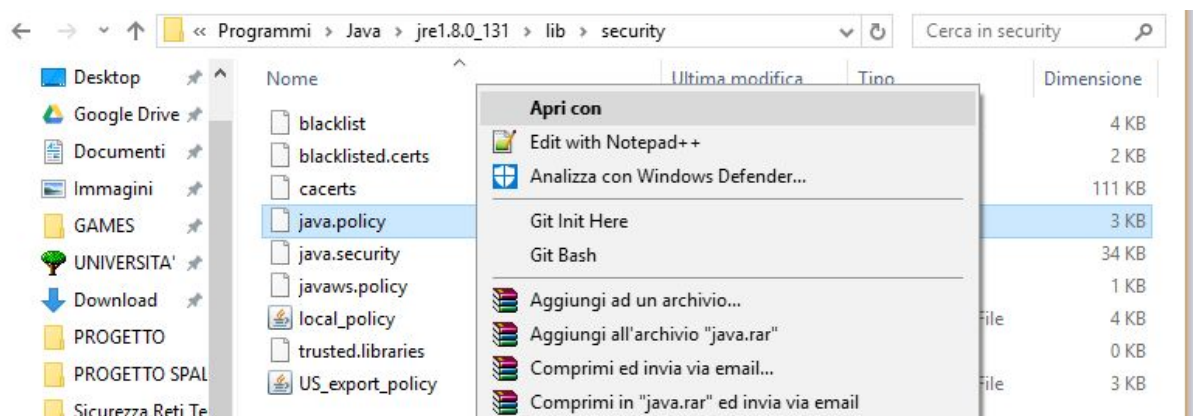
#### NB.

Se la procedura di avvio non dovesse terminare con successo, è possibile che ci sia un problema con le Policy di Java. Quindi seguire la seguente procedura descritta anche nel link seguente:

<http://novicejava1.blogspot.it/2014/06/unable-to-startup-derby-database.html>

Andare su:

- Questo-PC(o Risorse del Computer);
- Disco locale C:
- Programmi (x86 se è stata installata la versione x32);
- Java ---> jre1.8.0\_XX (scegliere l'ultima versione) ---> lib ---> security



Cliccare su "Apri con", poi selezionare "Blocco Note" e inserire dentro il paragrafo Grant la stringa ***permission java.net.SocketPermission "localhost:1525", "listen"***. Salvare il file chiuderlo e riprovare la procedura di avvio del Database.



## A.3 - Configurazione applicazione

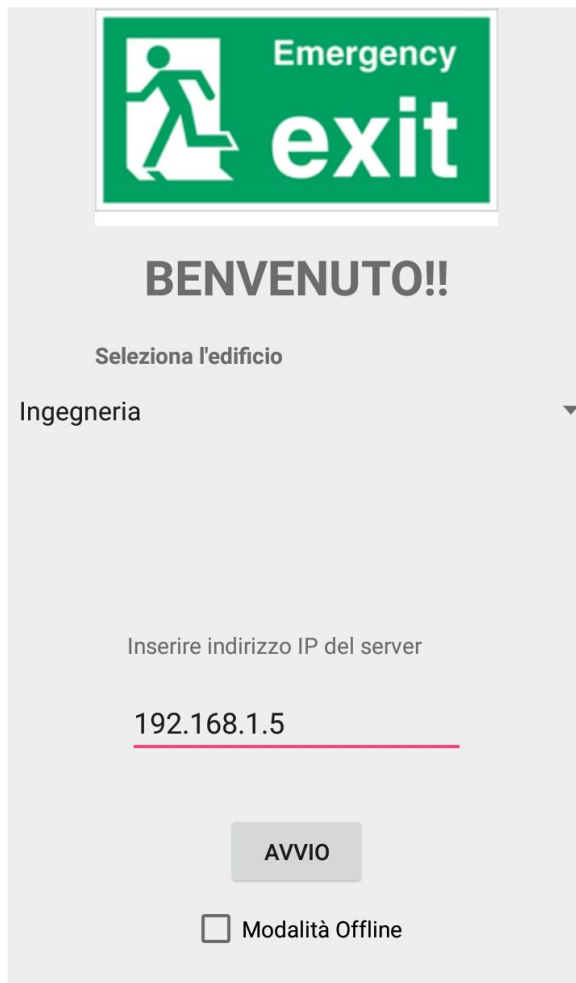
Una volta installato e configurato il server su una macchina, è possibile lavorare con l'applicazione. Per prima cosa questa va installata: nella cartella contenente gli elementi utili per l'installazione del sistema, è presente un file con estensione APK, chiamato "ProgettoIngegneriaDelSoftware.apk". Questo file va salvato all'interno del dispositivo mobile sul quale si vuole lavorare, in una cartella arbitraria, a discrezione dell'utente. A questo punto il file va eseguito. Il sistema operativo Android potrebbe generare, molto probabilmente, un avviso di pericolo: ciò è dovuto al fatto che la fonte viene ritenuta dal sistema un fonte non sicura, non essendo stata ottenuta tramite il marketplace. Per procedere all'installazione, è necessario settare nelle impostazioni del dispositivo, la "possibilità di installare software da fonti non sicure". Questa opzione varia in base ad ogni modello del telefono, ma generalmente è presente all'interno della sezione *Sicurezza*, nel menù *Impostazioni*, presentato graficamente con un segno di spunta. Una volta che attivata questa funzionalità, è possibile installare l'applicazione eseguendo il file APK. Una volta portata a termine l'installazione, alla prima apertura dell'applicazione è necessario configurare alcuni elementi software. Nella schermata che compare, è necessario inserire l'indirizzo ip della macchina su cui lavora il server. Qualora l'utente non conosca l'indirizzo ip della macchina, è sufficiente aprire il terminale shell e richiederlo tramite il comando. Per aprire il terminale bisogna digitare "Prompt di comandi" sulla barra di ricerca di Windows. Una volta avviata la shell di Windows, bisogna digitare il comando "arp -a" e premere invio.

```
C:\Users>arp -a

Interfaccia 192.168.1.4 -- 0x7
  Indirizzo Internet  Indirizzo fisico  Tipo
  192.168.1.1         bc-75-74-8b-9e-72  dinamico
  192.168.1.255       ff-ff-ff-ff-ff-ff  statico
  224.0.0.22          01-00-5e-00-00-16  statico
  224.0.0.251         01-00-5e-00-00-fb  statico
  224.0.0.252         01-00-5e-00-00-fc  statico
  239.255.255.250     01-00-5e-7f-ff-fa  statico
  255.255.255.255     ff-ff-ff-ff-ff-ff  statico

Interfaccia: 192.168.76.1 --- 0x10
  Indirizzo Internet  Indirizzo fisico  Tipo
  192.168.76.255       ff-ff-ff-ff-ff-ff  statico
  224.0.0.22          01-00-5e-00-00-16  statico
  224.0.0.251         01-00-5e-00-00-fb  statico
  224.0.0.252         01-00-5e-00-00-fc  statico
  239.255.255.250     01-00-5e-7f-ff-fa  statico
```

L'elemento dentro il riquadro indica l'indirizzo ip della macchina server. Tornando sull'applicazione, ora che si conosce l'indirizzo del server, bisogna inserirlo nella casella di testo e premere il bottone presente nella finestra.



The screenshot shows the main interface of the 'Emergency exit' application. At the top is a green header with a white icon of a person running and the text 'Emergency exit'. Below this, the word 'BENVENUTO!!' is displayed in large, bold, black letters. Underneath is a label 'Seleziona l'edificio' followed by a dropdown menu currently showing 'Ingegneria'. Further down is a text input field with the placeholder 'Inserire indirizzo IP del server' and the value '192.168.1.5' entered. Below the input field is a grey button labeled 'AVVIO'. At the bottom, there is a checkbox labeled 'Modalità Offline' which is currently unchecked.

Se la procedura è stata svolta correttamente, l'applicazione passa alla schermata di home. Se l'utente ha invece commesso un errore, questo viene evidenziato da apposite scritte. Va comunque sottolineato, anche in sede di revisione, questa finestra è stata creata esclusivamente per una fase di testing, considerando il fatto che il server non abbia un indirizzo fisso in rete. In questa prima accensione, nel momento in cui viene stabilita la connessione al server, vengono anche scaricati i documenti contenenti le stanze di un edificio e i sensori sparsi. Qualora si voglia modificare l'indirizzo IP del server, è possibile seguire una delle due opzioni: la prima consiste nell'andare a cancellare i dati dell'applicazione, andando a cercare l'opportuna voce all'interno del Menù Impostazioni, sezione Applicazioni, del sistema operativo Android, oppure è sufficiente accendere l'applicazione in un momento in cui non sia possibile collegarsi al server, il cui indirizzo è memorizzato nell'applicazione. L'applicazione, infatti, è stata concepita considerando che ogni tipo di comunicazione con il server, vada effettuata solo dopo che si è verificata la disponibilità di quest'ultimo.



## A.4 - Manuale di utilizzo e navigazione dell'applicazione

Una volta terminata questa fase di configurazione, supponendo che tutto sia andato a buon fine, è possibile utilizzare l'applicazione. Qualora sul dispositivo non sia stato attivato preventivamente il Bluetooth, l'applicazione fornisce all'utente l'opzione per attivarlo, semplicemente cliccando su di un bottone, sotto l'opportuna richiesta. Il Bluetooth è un elemento fondamentale per quanto riguarda il funzionamento dell'applicazione. In fase di sviluppo e testing del codice, è emersa una problematica insita nelle ultime versioni sviluppate da Android, a partire da Marshmallow: per poter ricercare un sensore è necessario attivare il sistema di localizzazione del dispositivo, dove lavora l'applicazione. Anche in questo caso, l'applicazione, riconoscendo la versione di Android su cui sta girando, richiede all'utente l'attivazione della funzionalità, tramite un'opportuna finestra di dialogo. Nel momento in cui l'utente ha attivato anche queste opzioni, l'applicazione risulta utilizzabile. La prima schermata che compare è la Home, quella principale. Qui l'utente vede il marchio dell'università, mentre sulla sinistra è presente aprire un menù, che all'apertura mostra tre voci. Come già sottolineato in precedenza, essendo la maggior parte delle funzionalità legate alla comunicazione con il server, qualora l'utente decidesse di lavorare in modalità offline, alcune di queste potrebbero risultare non disponibili (come ad esempio la possibilità di autenticarsi presso la piattaforma da parte dell'utente, oppure la possibilità di registrarsi presso di questa). Inoltre, avendo impostato il progetto in modo che l'utente possa lavorare sia come *guest* che come *individuo autenticato*, le voci nel menù risultano diverse in base a questa condizione. Procedendo per ordine, partendo dal presupposto che l'unica delle tre opzioni che rimane immutata, in entrambi i casi, risulta essere quella per l'accesso alle mappe, le restanti componenti del menu possono essere riassunte in questa maniera:

- Utente guest: l'utente guest non gode di nessun privilegio, non essendosi ancora autenticato presso la piattaforma. Nel caso si voglia procedere in questa direzione, nel menù è possibile selezionare l'opzione *login*. Cliccando su questa compare un dialog, all'interno del quale è possibile inserire l'indirizzo email e password del proprio account. In ogni caso, qualunque fosse l'esito, l'utente comunque rimane nella pagina home. Le differenze sostanziali, oltre ad alcuni elementi dell'interfaccia grafica, sono quantificabili soprattutto a livello del server. Se invece un utente ancora non possiede nessun account, cliccando sulla seconda voce è possibile effettuare la procedura di registrazione. A questo punto si passa su un'altra finestra, quella contenente la form per la registrazione, dove è possibile inserire i propri dati personali, come già trattato nel paragrafo sulla descrizione dell'utente. Una volta inseriti tutti i dati, o almeno i cinque obbligatori, l'utente può cliccare sul bottone di invio, per processare questa richiesta che, solo dopo essere stata

valutata positivamente dai controlli dell'applicazione, viene inviata al server, il quale risponde con comunicando all'utente l'esito della procedura. A questo punto l'utente viene reindirizzato sulla pagina home dell'applicazione. Qualora qualche dato non fosse stato inserito in maniera corretta, l'applicazione indica all'utente il tipo di errore commesso e a quale voce corrisponda, evidenziandolo con un colore opportuno. L'utente quindi rimane ancora in questa finestra, fino a che non riesce a portare a termine la procedura con successo oppure decide di tornare indietro, senza essersi iscritto.

- Utente autenticato: l'utente che effettua l'accesso alla piattaforma visualizza un contenuto del menù con voci che, come è possibile intuire, duali rispetto al caso precedente. La prima voce dà la possibilità di effettuare la procedura di logout, tornando alla condizione di utente guest. Un click su questa voce mantiene comunque l'utente sulla pagina home. L'altra voce presente nel menù, *Modifica dati personali*, invece, permette di modificare i propri dati personali. Come per il caso della registrazione, l'applicazione mostra la stessa finestra descritta nel caso duale. In questo momento però, le textview dove inserire i propri dati contengono una stringa, corrispondente al valore inserito dall'utente in fase di registrazione per quel campo. Una volta che l'utente ha effettuato le modifiche desiderate, si può procedere con l'invio dei dati al server, cliccando sul bottone in fondo alla pagina, secondo le stesse dinamiche già viste in precedenza. Nel caso in cui un utente sia autenticato, nel menù è presente un'ultima voce, che gli permette di passare in un'altra finestra, dove può leggere i propri dati personali.

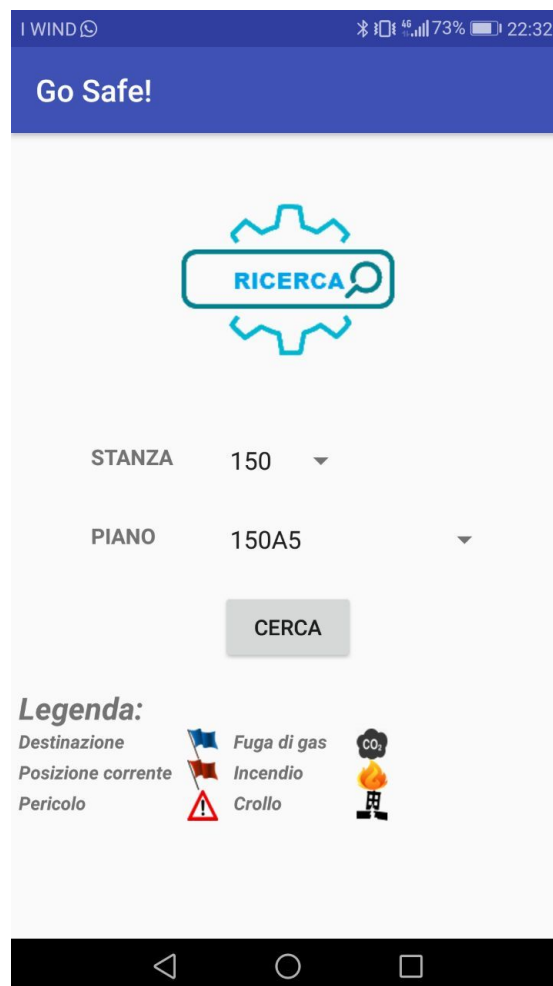
Come già descritto in precedenza, l'unico elemento comune alle due casistiche sopra descritte riguarda la possibilità di accedere alle mappe. Cliccando sulla voce presente nel menù, l'applicazione cambia finestra, visualizzandone una nuova, all'interno della quale è possibile scegliere la stanza verso cui ci si vuole dirigere. L'interfaccia in questa pagina è composta da due menù a tendina, dove è possibile selezionare rispettivamente il piano e la stanza dove ci si vuole dirigere. Come già descritto in precedenza, il contenuto del secondo menu dipende da ciò che è stato selezionato nel primo, in modo da mantenere la coerenza. Nel momento in cui l'utente ha deciso il luogo in cui dirigersi, deve cliccare l'apposito bottone, che gli permette di passare alla visualizzazione della mappa. Dal momento in cui si clicca, non è detto che immediatamente l'applicazione cambi finestra. Ciò non è dovuto alla complessità dell'operazione in sé, bensì dal fatto che l'applicazione prima di poter passare alla *modalità ricerca*, deve prima interrompere il precedente scanner. Se questo, per un caso fortuito si trovasse nello stato in cui l'applicazione si sta collegando al beacon, nello specifico mentre sta leggendo i dati, potrebbero passare anche cinque secondi prima che si visualizzi la mappa. Per via della natura di scanner e connessione, concepite come macchine a stati, si è voluto evitare che il passaggio alla visualizzazione della mappa fosse brutale, perciò si è deciso di fare in

modo che la macchina a stati fosse portata ad evolvere in maniera naturale, indirizzandola però verso il suo stato finale.

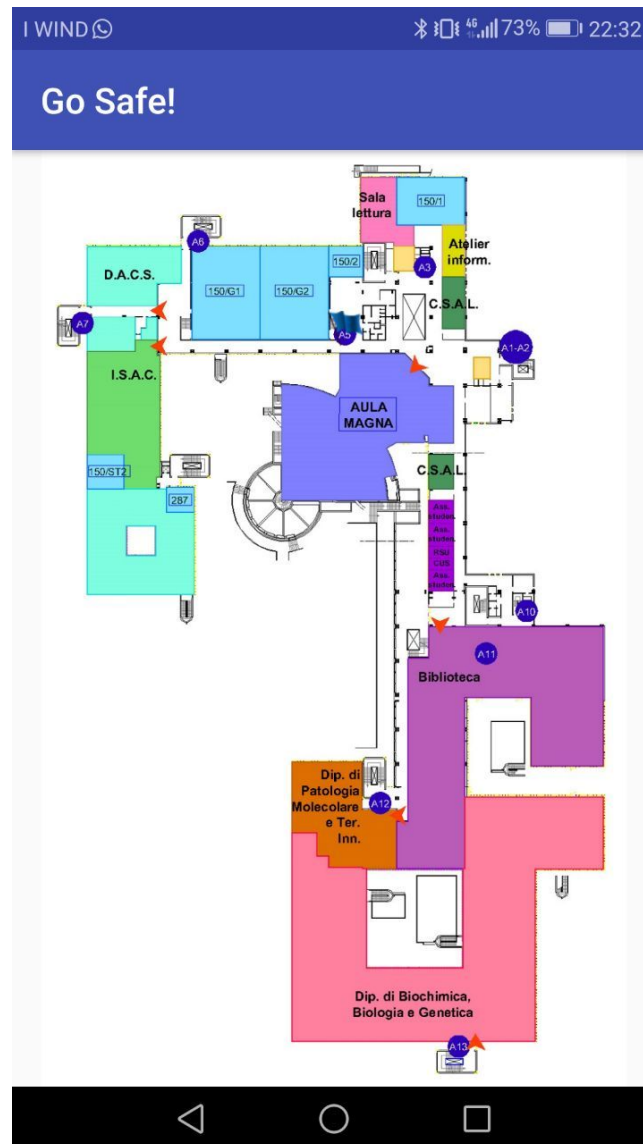
Terminata questa sorta di transitorio, in cui l'applicazione imposta alcuni parametri e riinizializza un nuovo ciclo di scanner per questa modalità, si passa alla finestra in cui si visualizza la mappa. All'interno della mappa l'utente può trovare differenti elementi che rappresentano la posizione stimata dei vari elementi indicati nella legenda. L'utente, qualora avesse già incontrato un sensortag, può vedere la propria posizione (indicata da una bandiera rossa) e il suo obiettivo (indicato da una bandiera blu). Qualora il suo obiettivo si trovi su di un altro piano, allora l'utente viene indirizzato verso le scale.

Qualora invece il server notifica un'emergenza all'applicazione, all'interno della mappa vengono visualizzati delle particolari icone (vedi legenda sull'interfaccia della ricerca) per segnalare la posizione dell'eventuale incidente. Oltre a questa segnalazione viene anche riportata la posizione dell'uscita di emergenza di quel particolare piano. La notifica dell'emergenza sarà segnalata attraverso una notifica push.

Quindi per poter interagire con le mappe basta selezionare la voce "mappe" del menu della schermata di home a quel punto l'utente si troverà di fronte la seguente schermata:

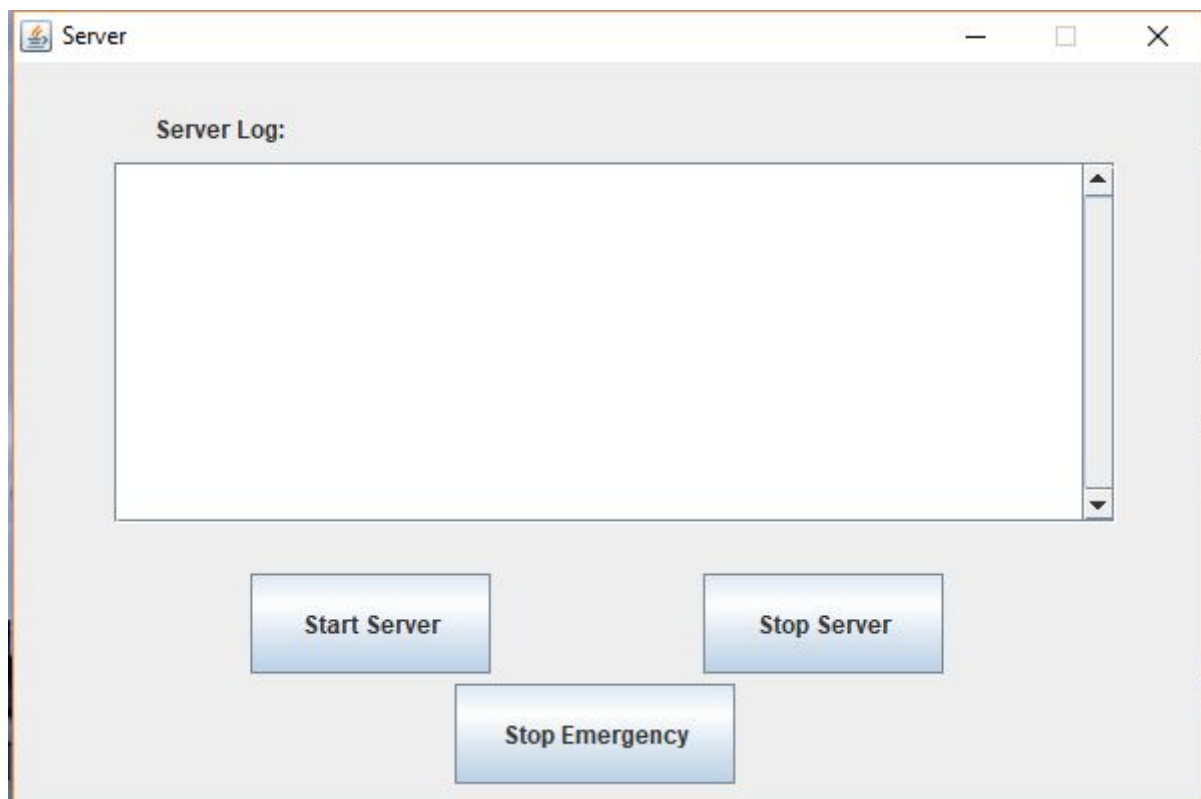


A quel punto selezionato il piano e l'edificio per poter visualizzare la mappa vera e proprio, come si può vedere vedere nella seguente schermata



## A.5 -Gestione delle emergenze

L'ultima parte riguarda il gestore delle emergenze, che si trova nella cartella Eseguibili sotto il nome di NotificationHandler, infatti è stato realizzato un piccolo programma che comunica a tutti gli utenti connessi in quel momento i pericoli che possono incontrare lungo il percorso. Prima di avviare questo applicativo assicurarsi che il database sia stato aperto con successo. L'interfaccia si presenta in questa maniera:



La finestra contiene tre pulsanti e un'area di testo in cui compaiono alcuni messaggi relativi all'elaborazione delle operazioni. Entrando nel dettaglio dei singoli elementi:

- **Start Server**, invia le notifiche contenenti l'avviso della presenza di un'emergenza, a tutti gli utenti connessi, identificati tramite indirizzo IP. Nell'area di testo viene comunicato l'esito della procedura, sia nel caso che l'utente abbia ricevuto la notifica, sia nel caso il server non sia riuscito a contattarlo.
- **Stop Server**, interrompe l'invio di richieste a tutti gli utenti. Una volta cliccato si blocca quindi il processo di comunicazione.
- **Stop Emergency**, invia un messaggio a tutti gli utenti connessi che l'emergenza è terminata e non ci sono più pericoli.