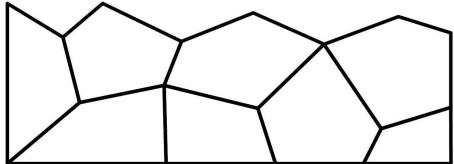




LIFE-SAVING

FURNITURE SYSTEM



Safe-Device Locator UI

Lorenzo Tanganelli

Luca Patarca

Nico Trionfetti

24 giugno 2021

Indice

1	Introduzione	2
2	Prima iterazione	4
2.1	Requisiti	4
2.2	Modelli	5
2.3	Progettazione e Implementazione	6
2.4	Validazione e rilascio	12
3	Seconda iterazione	14
3.1	Requisiti	14
3.2	Progettazione e Implementazione	14
3.3	Validazione e rilascio	16
4	Terza iterazione	18
4.1	Requisiti	18
4.2	Modelli	18
4.3	Progettazione e Implementazione	18
4.4	Validazione e rilascio	23
5	Quarta iterazione	24
5.1	Requisiti	24
5.2	Modelli	24
5.3	Progettazione e Implementazione	24
5.4	Validazione e rilascio	27
6	Quinta iterazione	29
6.1	Requisiti	29
6.2	Modelli	29
6.3	Progettazione e implementazione	29
6.4	Validazione e rilascio	34
7	Deployment	38
7.1	Configurazione	38
7.1.1	Frontend	38
7.1.2	Backend	39
7.2	Compilazione	39
7.3	Requisiti	39

Capitolo 1

Introduzione

Il progetto di ricerca industriale *S.A.F.E Project*[1] ha come obiettivo la realizzazione di sistemi di arredo innovativi capaci di trasformarsi in sistemi intelligenti di protezione passiva delle persone in caso di crollo dell'edificio causato da un terremoto.

Questi sistemi di arredo smart saranno dotati di sensoristica "salva-vita" capace di pre-allertare in caso di terremoto, di rilevare e localizzare la presenza di vita dopo un crollo, di monitorare le condizioni ambientali sotto le macerie e di elaborare e trasmettere informazioni utili a chi deve portare soccorso.

Il ciclo di vita dei sensori si divide in tre scenari operativi:

- i. **Tempo di pace:** monitoraggio per il pre-allertamento (es. misure accelerometriche)
- ii. **Durante l'evento:** invio dei dati per il rilevamento dei danni (es. misure accelerometriche, inclinometriche e di spostamento) e attivazione di logiche di intervento in seguito al riconoscimento dell'evento.
- iii. **Dopo l'evento:** invio dei dati per la localizzazione delle vittime e monitoraggio ambientale al fine di guidare gli operatori nel triage di soccorso.

L'invio di dati tra i sensori ed il mondo esterno avviene utilizzando la tecnologia LoRa.

LoRa consente trasmissioni a lungo raggio e a basso consumo energetico arrivando oltre 10 km nelle zone rurali e 3–5 km in zone fortemente urbanizzate.[2]

Facendo riferimento al modello ISO/OSI la tecnologia è presente in due strati:

- **LoRa:** Il livello fisico LoRa è proprietario della Semtech e non se ne conoscono i dettagli implementativi. LoRa utilizza una modulazione a spettro espanso proprietaria, derivata dalla modulazione Chirp Spread Spectrum (CSS). Inoltre utilizza la codifica Forward Error Correction (FEC) come meccanismo di rilevazione e successiva correzione degli errori contro le interferenze.
- **LoRaWAN:** LoRaWAN è un protocollo del livello Media Access Control (MAC) che lavora a livello di rete per la gestione delle comunicazioni tra gateway Low Power Wide Area Network (LPWAN) e dispositivi end-node come protocollo di routing.

Lo scenario operativo post evento si divide in tre attività:

- i. **Campionatura:** mediante l'utilizzo di un drone dotato di tecnologia che supporta il protocollo LoRaWAN viene campionata l'area coperta dalle macerie. Durante la fase di volo vengono memorizzati i dati ricevuti dai sensori e la potenza del segnale.
- ii. **Analisi dati:** sfruttando opportuni algoritmi di localizzazione vengono analizzati i dati memorizzati dal drone così da determinare dei centroidi in cui si suppone si trovi il disperso.
- iii. **Guidare soccorritori:** i soccorritori, dotati di opportuni tablet, visualizzeranno una mappa con la heatmap e i centroidi risultanti dall'attività di analisi dati, così da potersi orientare per individuare i dispersi.

Il nostro progetto, all'interno di S.A.F.E., ha l'obiettivo di creare un applicativo per tablet android utile nell'ultima attività post evento.

Trovandosi in uno stato d'emergenza, l'applicazione punta ad avere un interfaccia grafica semplice e funzionale così da agevolare il lavoro degli operatori. Inoltre, dovrà essere in grado di funzionare senza connessione internet in quanto il terremoto potrebbe causare l'interruzione delle comunicazioni.

Capitolo 2

Prima iterazione

2.1 Requisiti

Durante l'attività di analisi sono emersi i seguenti requisiti:

- **Visualizzazione di una mappa**

Il soccorritore necessita di una mappa sempre visibile del luogo in quanto potrebbe non conoscere la zona da soccorrere.

- **Rendering di dati vettoriali come heatmap**

La posizione del sensore individuata nella fase d'analisi dati potrebbe non essere corretta, quindi si prevede la possibilità di far visualizzare la heatmap degli RSSI (*Received Signal Strength Indication*), relativi ai sensori rilevati nell'attività di campionatura, all'operatore così da poter fare ulteriori supposizioni in base alle condizioni delle macerie e velocizzare il recupero del disperso.

- **Rendering di marker sulla base delle posizioni dei sensori**

Si necessita l'utilizzo di marker per visualizzare la probabile posizione del sensore.

- **Rendering di dati vettoriali come poligoni**

Si prevede la possibilità di visualizzare poligoni così da delimitare l'area di ricerca in base alle informazioni fornite dall'attività di analisi.

- **Geolocalizzazione del dispositivo**

Le macerie degli edifici potrebbero aver ricoperto le strade perciò individuare la posizione GPS del dispositivo sulla mappa può essere di supporto all'operatore per orientarsi.

- **Recupero dati dell'attività di analisi**

Per visualizzare i dati vettoriali l'applicazione dovrà recuperare i dati risultanti dall'attività di analisi.

- **Mantenimento dello stato nel caso in cui venga terminata l'applicazione**

Si prevede che l'applicativo mantenga il suo stato in caso di chiusura e successivo riavvio. Per esempio, se il dispositivo si dovesse scaricare, al

riavvio l'applicazione dovrà trovarsi allo stato precedente lo spegnimento, così da minimizzare i tempi di attesa.

- **Funzionamento senza comunicazione internet**

Il terremoto potrebbe causare l'interruzione delle comunicazioni, per questo motivo si prevede che l'applicazioni non necessiti di internet per funzionare.

- **Gestione squadre**

I soccorritori potrebbero essere divisi in squadre, per questo motivo si prevede la possibilità di inserire nel sistema le squadre presenti e assegnar loro i sensori di cui si occupano.

In questa iterazione abbiamo scelto di sviluppare questi requisiti (relativo use case diagram in Figura 2.1):

- Visualizzazione di mappe.
- Rendering di dati vettoriali come heatmap. ("Visualizzazione Heatmap")
- Rendering di marker sulla base delle posizioni dei sensori. ("Visualizzazione Marker")
- Visualizzazione delle informazioni inviate dai sensori. ("Visualizzazione lista Marker")
- Funzionamento senza comunicazione internet.

Si prevede che l'eseguibile Rust e l'interfaccia grafica vengano eseguite nel medesimo dispositivo così da condividere i tiles accedendo ad una cartella condivisa. Invece, per il rendering dei dati, per visualizzare heatmap e marker, si utilizza dei file mockup salvati localmente, in quanto non è ancora disponibile il "server" che analizza i dati campionati.

2.2 Modelli

Sulla base dei requisiti che abbiamo scelto di sviluppare in questa iterazione sono stati identificati i seguenti casi d'uso (rappresentazione grafica in Figura 2.1).

- **Visualizzazione mappa**

Viene avviato dall'attore Rescuer in automatico all'avvio dell'applicazione, permette di caricare i tiles per visualizzare la mappa (Nella figura 2.2 sono descritti gli scenari del caso d'uso sotto forma di sequence diagram).

- **Caricamento dati**

È incluso nel caso d'uso *Visualizzazione mappa* e viene avviato per eseguire il caricamento dei dati relativi ai sensori rilevati (Nella figura 2.3 è riportato il diagramma di sequenza relativo al caso d'uso in questione).

- **Visualizzazione marker**

Estende il caso d'uso *Caricamento dati* e permette una rappresentazione grafica dei marker sulla mappa.

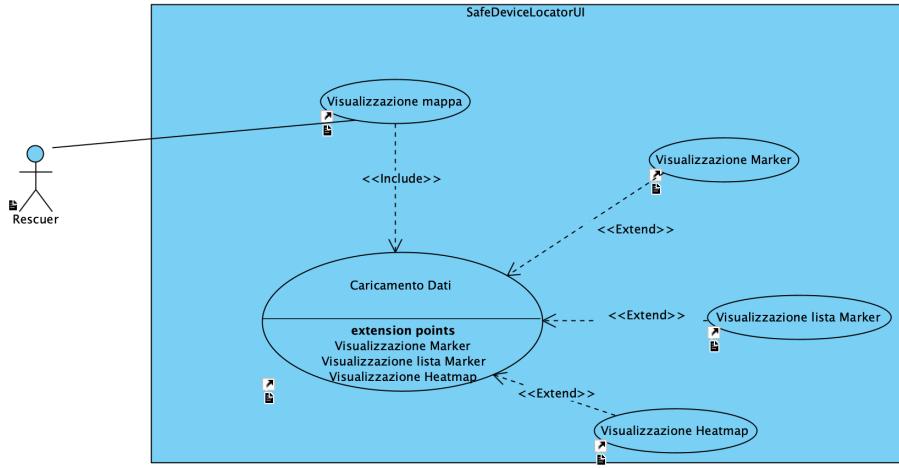


Figura 2.1: Use Case

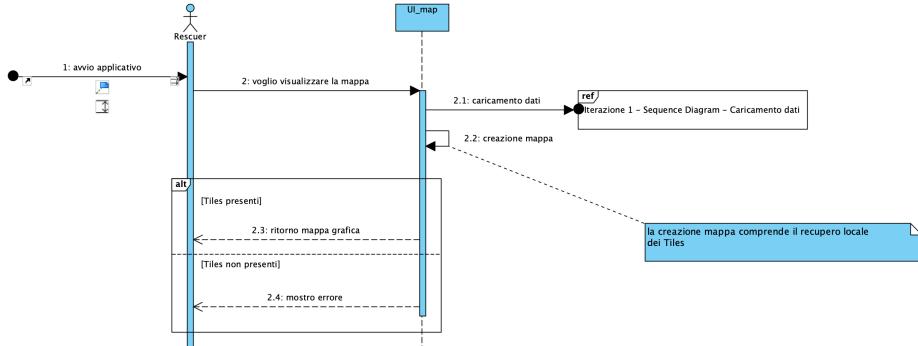


Figura 2.2: SD: Visualizzazioni mappa

- **Visualizzazione lista marker**

Estende il caso d'uso *Caricamento dati* consentendo di visualizzare una lista dei sensori rilevati con le relative informazioni aggiuntive.

- **Visualizzazione Heatmap**

Estende il caso d'uso *Caricamento dati* così da disegnare la heatmap sulla mappa.

2.3 Progettazione e Implementazione

Terminata la prima attività di progettazione si è scelto di utilizzare il framework *Flutter*[3] basato su *Dart*[4] per lo sviluppo del frontend, così da creare un'applicazione nativa compilata sia per mobile che per desktop. Flutter utilizza i widget come elemento principale che consentono di risparmiare tempo nello sviluppo di elementi dell'interfaccia utente di base per ogni schermo e risoluzione. Flutter ha il proprio toolkit widget, ma tutti i componenti vengono renderizzati

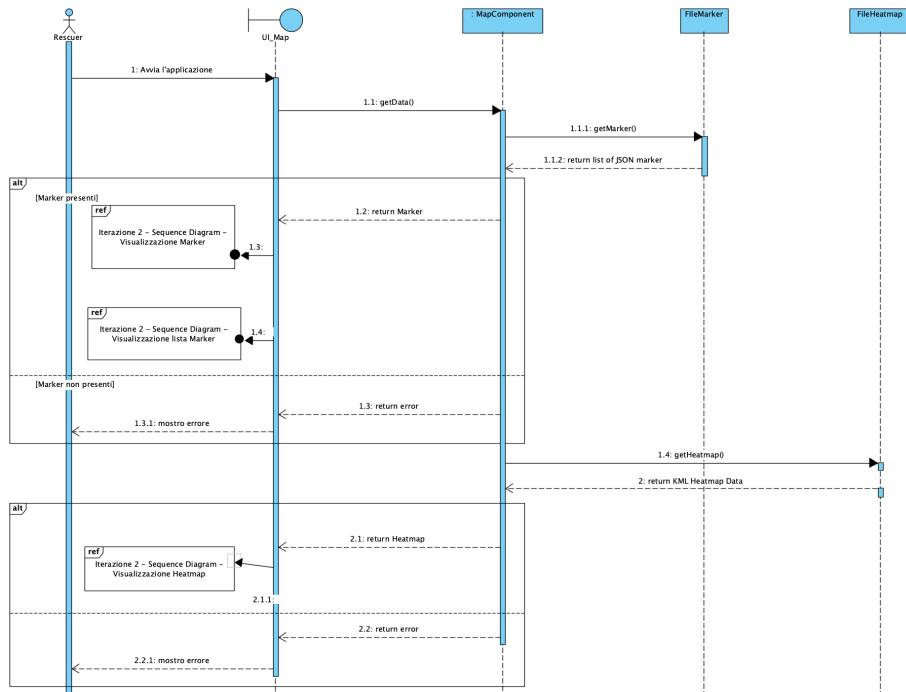


Figura 2.3: SD: Caricamento dati

in modo nativo. Ciò conferisce alle app un aspetto nativo e migliora le prestazioni. Flutter è fondamentalmente un wrapper attorno a un'app che utilizza uno speciale metodo di comunicazione chiamato Platform Channels per connettere i dati alle lingue native. È facile da usare e consente agli sviluppatori di accedere all'hardware in quanto dispone di librerie che consentono di collegarsi al chip GPS, fotocamera e microfono[5].

Per visualizzare la mappa abbiamo deciso di appoggiarci all'implementazione Dart di Leaflet per Flutter denominata *Flutter_Map*[6]; Leaflet è la principale libreria JavaScript open source per mappe interattive ottimizzate per dispositivi mobili. Il pacchetto *Flutter_Map* rende disponibili una serie di implementazioni della classe *LayerOptions*, come per esempio *CircleLayerOptions*, *MarkerLayerOptions*, *PolygonLayerOptions* e *TileLayerOptions*, i quali permettono di fare il rendering della mappa e di aggiungerci informazioni. Purtroppo *Flutter_Map* non dispone di un implementazione di *LayerOptions* per fare il rendering di una heatmap da aggiungere come layer alla mappa. Per questo motivo è stato deciso di adattare la classe *MarkerLayerOptions* così da "creare manualmente" l'heatmap.

```

1   Color _getColor(double value) {
2     int range = 5;
3     int red = (255 * (value / range)).toInt();
4     int green = (255 * (1 - (value / range))).toInt();
5     return Color.fromRGBO(0xFF, red, green, 00);
6   }
7
8   Marker _createHeatmapMarker(LatLng point, double value) {
9     return Marker(

```

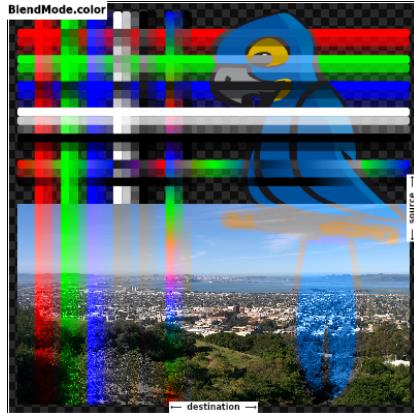


Figura 2.4: BlendMode.color example

```

10      width: 30.0,
11      height: 30.0,
12      point: point,
13      builder: (context) => DecoratedBox(
14          decoration: BoxDecoration(
15              backgroundBlendMode: BlendMode.color,
16              shape: BoxShape.circle,
17              gradient: RadialGradient(
18                  colors: [getColor(value), Colors.transparent
19                      ])),
20          position: DecorationPosition.foreground,
21      );

```

Per disegnare la heatmap creiamo dei *BoxShape* circolari colorati secondo un'apposita funzione, *_getColor(double value)*. Quando si disegna una forma su un canvas è possibile utilizzare diversi algoritmi per fondere i pixel. Nel nostro caso abbiamo scelto di utilizzare l'algoritmo *BlendMode.color* il quale prende la tonalità e la saturazione dell'immagine d'origine e la luminosità dell'immagine di destinazione: l'effetto è quello di colorare l'immagine di destinazione con l'immagine di origine (nella Figura 2.4 riportiamo un esempio di *BlendMode.color*).

Per permettere la visualizzazione dei Tiles, Markers e Heatmap sopra la mappa è necessario passare una lista con i relativi layer al widget *FlutterMap*. Per controllare il rendering della mappa, invece, è necessario fornire un *MapController*, per fare ciò recuperiamo l'oggetto desiderato dal provider: *MapProvider*.

```

1 class _Maps extends State<Maps> {
2     /*
3     ...
4     */
5
6     @override
7     Widget build(BuildContext context) {
8         return FlutterMap(
9             mapController: context.watch<MapProvider>().
10                getMapController(),

```

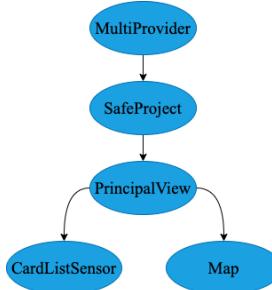


Figura 2.5: wedget tree

```

11     options: MapOptions(
12       plugins: [PopupMarkerPlugin()],
13       center: LatLng(43.140360, 13.068770),
14       zoom: context.watch<MapProvider>().getCurrentZoom(),
15       maxZoom: _maxZoom,
16       pinchZoomThreshold: 1.0,
17       onTap: (_) => _popupLayerController.hidePopup(),
18     ),
19     layers: [
20       TileLayerOptions(
21         urlTemplate: "$tilesPath/{z}/{x}/{y}.png",
22         maxZoom: _maxZoom,
23       ),
24       MarkerLayerOptions(
25         markers: context.watch<SensorProvider>().
26           getSensorMarkers(),
27         MarkerLayerOptions(
28           markers: context.watch<SensorProvider>().
29             getHeatmapMarkers(),
30         )
31       );
32     }
33   }
34 }
```

Si è scelto di utilizzare il pacchetto *provider*[7], per lo state management, così da condividere i dati più facilmente nel widget tree (widget tree in Figura 2.5).

Utilizzando la keyword *with*, creiamo due classi provider: "*MapProvider*" e "*SensorProvider*", che utilizzano la classe mixin "*ChangeNotifier*". Mixin si riferisce all'abilità di aggiungere le capacità di una o più classi alla propria classe, senza ereditare da quelle classi. I metodi di quelle classi possono essere chiamati sulla propria classe e il codice verrà eseguito. Di seguito riportiamo il codice della classe *MapProvider* come esempio.

```

1  class MapProvider with ChangeNotifier {
2
3   MapController _mapController = MapController();
4   /*
5   ...
6   */
7
8   MapController getMapController() {
9     return _mapController;
10  }
11 }
```

```

12     zoomOut() {
13         _currentZoom = _currentZoom - 1;
14         _mapController.move(_mapController.center, _currentZoom);
15         notifyListeners();
16     }
17
18     zoomIn() {
19         _currentZoom = _currentZoom + 1;
20         _mapController.move(_mapController.center, _currentZoom);
21         notifyListeners();
22     }
23 }

```

Come è possibile notare alle righe 15 e 21 viene richiamato il metodo *notifyListeners()*, il quale fa parte della classe *ChangeNotifier* e consente di notificare il cambiamento di stato a tutti i componenti del widget tree interessati.

Per permettere l'accesso globale ai provider nel widget tree abbiamo instanziato un oggetto *MultiProvider* a cui passiamo una lista con le implementazioni dei provider e il widget root *SafeProject*.

```

1 void main() {
2     runApp(MultiProvider(providers: [
3         ChangeNotifierProvider(create: (context) => MapProvider()),
4         ChangeNotifierProvider(create: (context) => SensorProvider()),
5         child: SafeProject())));
6 }
7
8 class SafeProject extends StatelessWidget {
9     @override
10    Widget build(BuildContext context) {
11        return MaterialApp(
12            title: "SafeProject",
13            home: PrincipalView());
14    }
15 }

```

Per la generazione di pacchetti contenenti le mappe da utilizzare offline si è scelto di creare un eseguibile in *Rust*[8]. Rust è un linguaggio di programmazione multi-paradigma incentrato sulla sicurezza e sulle prestazioni, velocità ed efficienza. Inoltre è orientato al multithreading per il quale fornisce delle primitive di sincronizzazione efficaci e semplici da implementare[9].

L'eseguibile in questione prende in input delle coordinate geografiche, scarta i tiles in formato *.png* (256*256 pixel) e li organizza in cartelle secondo lo standard OpenStreetMap, conosciuto come "Slippy Map Tilenames" o "XYZ":

```
./tiles/{z}/{x}/{y}.png
```

dove {z} indica lo zoom dell'immagine, mentre {x} e {y} indicano rispettivamente latitudine e longitudine.

- X va da 0 (180° W) a $2^{zoom} - 1$ (180° E).
- Y va da 0 (85.0511° N) a $2^{zoom} - 1$ (85.0511° s). ¹

Per determinare i nomi dei tiles bisogna seguire la seguente procedura ²:

¹il numero 85.0511 è il risultato di $\arctan(\sinh(\pi))$. Usando questo limite, l'intera mappa diventa un quadrato (molto grande).

²lat e lon sono in radianti

1. Proiettare le coordinate nella proiezione di Mercatone.

$$x = lon$$

$$y = \ln[\tan(lat) + \sec(lat)]$$

2. Trasformare l'intervallo di x e y in $[0, 1]$ e spostare l'origine nell'angolo in alto a sinistra.

$$x = \frac{1 + \frac{x}{\pi}}{2}$$

$$y = \frac{1 - \frac{y}{\pi}}{2}$$

3. Calcolare il numero di tiles sulla mapp $n = 2^{zoom}$.

4. Moltiplicare x e y per n .

5. Arrotondare i risultati per difetto così da ottenere TileX e TileY.

Alleghiamo la nostra implementazione della funzione Rust per determinare i path dei tiles.

```

1 fn deg2num(lat_deg: f32, lon_deg: f32, _zoom: i32) -> (i32, i32) {
2     let lat_rad = lat_deg.to_radians();
3     let mut n: f64 = 2.00;
4     n = n.powi(_zoom);
5     let x_tile = ((lon_deg as f64 + 180.0) / 360.0 * n) as i32;
6     let y_tile = ((1.0 - (((lat_rad as f64).tan()).asinh()) / std::
7         f64::consts::PI) / 2.0 * n) as i32;
8     return (x_tile, y_tile);
9 }
```

Una volta terminato il processo di download la directory generata viene compressa nel file *tiles.zip*. Si è deciso di scaricare le immagini da zoom 15 (scala 1:15 mila) a zoom 22 (scala 1:125) in un raggio di 5km dal punto centrale. Questa scelta è motivata dal fatto che vogliamo dare una visione generale di un quartiere o di un paese fino ad arrivare ad avere una visione dettagliata di singoli edifici.

Per scaricare la mappa da rendere disponibile offline ci appoggiamo a *Thunderforest*[10]: un fornitore di tiles renderizzati dai dati di *OpenStreetMap*[11] (di seguito riportiamo il codice relativo al download dei tiles).

```

1 const PARALLEL_REQUESTS: usize = 128;
2
3 pub async fn get_map_tiles(coordinates: HashSet<TileCoords>){
4     let client = Client::new();
5     let bodies = stream::iter(coordinates)
6         .map(|coords| {
7             let client = client.clone();
8             tokio::spawn(async move {
9                 let url = format!("http://tile.thunderforest.com/
10                     cycle/{}//{}//{}.png", coords.zoom, coords.x,
11                     coords.y);
12                 let mut headers = HeaderMap::new();
13                 headers.insert(USER_AGENT, "...".parse().unwrap());
14                 let resp = client.get(url.as_str()).headers(headers)
15                     .send().await?;
16                 match resp.bytes().await {
17                     Ok(b) => Ok((b, coords)),
18                 }
19             })
20         })
21         .collect();
22 }
```

```

15             Err(e) => Err(e),
16         }
17     })
18 }
19 .buffer_unordered(PARALLEL_REQUESTS);
20
21 let output_dir = get_data_dir();
22
23 bodies
24 .for_each(|b| async {
25     match b {
26         Ok(Ok((b, c))) => save_image(c, b, &output_dir),
27         Ok(Err(e)) => eprintln!("Got an error:{}", e),
28         Err(e) => eprintln!("Got an error:{}", e)
29     }
30 })
31 .await;
32 }

```

Per ottimizzare il tempo d'attesa durante il download paralleziamo 128 richieste in quanto per generare un pacchetto è necessario scaricare un numero elevato di immagini; se utilizzassimo un'approccio sequenziale avremmo dei tempi d'attesa non consoni al dominio del progetto.

2.4 Validazione e rilascio

Terminata l'attività di sviluppo del prototipo abbiamo soddisfatto quasi tutti i requisiti che ci eravamo prefissati per questa iterazione. Come si può notare dalla Figura 2.6 il prototipo visualizza una mappa con dei marker che cambiano il proprio aspetto in base allo stato del sensore: verde se è stato verificato, rosso altrimenti. Sulla sinistra si può notare la lista dei sensori con i relativi dati, in caso di tap sulla card del sensore la mappa sposta la visuale sopra le coordinate del sensore selezionato.

L'unico requisito che non è stato soddisfatto a pieno è stato quello di disegnare l'heatmap, in quanto necessitavamo di visualizzare una mappa di calore che, in base alla scala della mappa, mettesse in relazione la distanza dei punti e l'RSSI rilevato.

Come si può notare nella Figura 2.6, abbiamo disegnato dei punti che tenevano conto dell'RSSI ma non siamo stati in grado di tenere in considerazione la densità dei punti. Questo ci portava a visualizzare una heatmap che: se avevamo tanti punti ravvicinati ma con intensità molto bassa venivano disegnati di verde, invece necessitavamo che in questa circostanza la zona interessata venisse colorata di rosso.

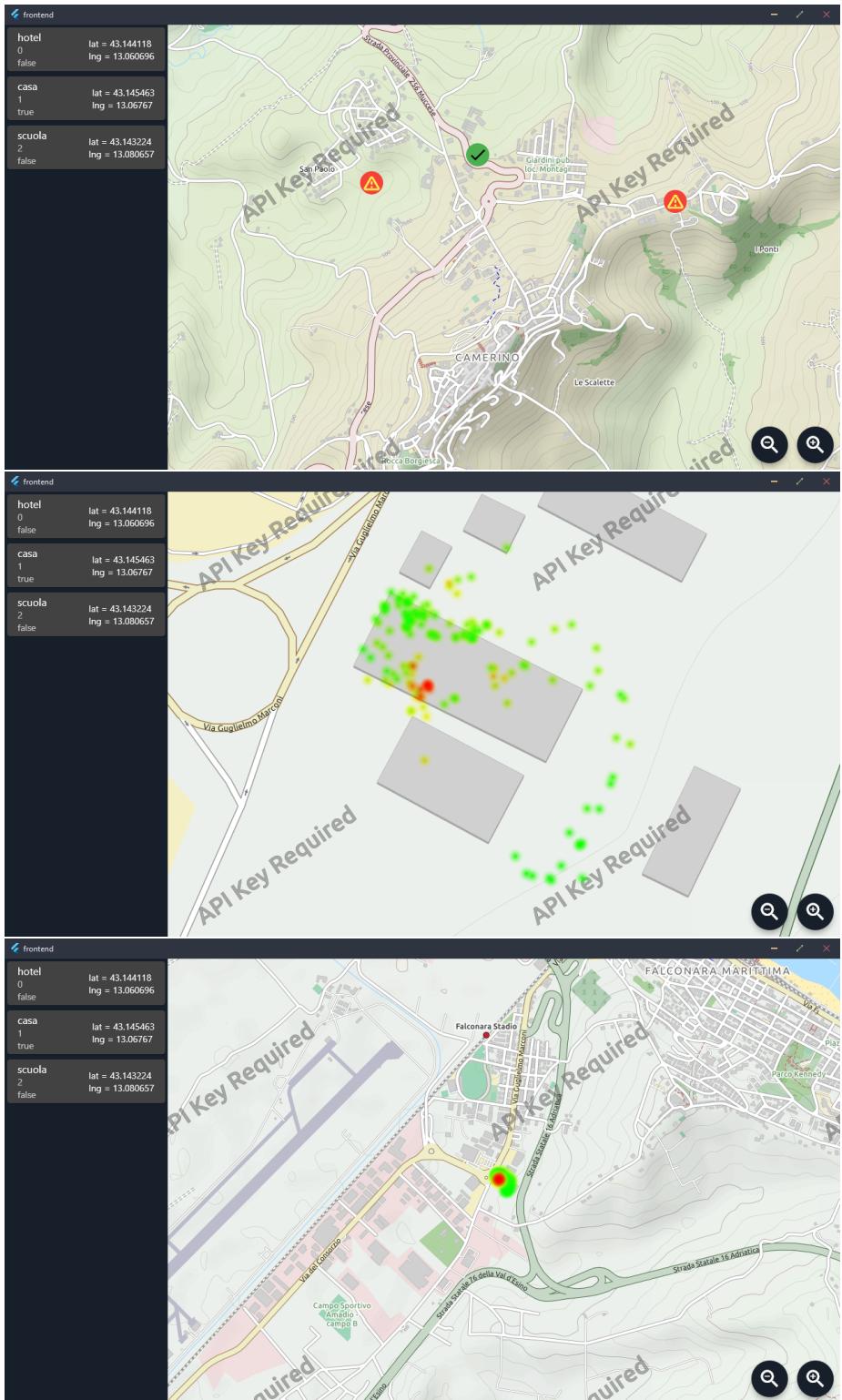


Figura 2.6: Prototipo Flutter

Capitolo 3

Seconda iterazione

3.1 Requisiti

Visti i feedback della precedente iterazione abbiamo deciso di mantenere i medesimi requisiti riguardanti il rendering della mappa ma di rivedere progettazione e implementazione cambiando tecnologie utilizzate. Questo cambiamento è motivato dal fatto che implementare un'heatmap che soddisfa il relativo requisito avrebbe richiesto più tempo che sviluppare un nuovo prototipo con una tecnologia differente, e per questo avremmo rischiato di non rispettare le scadenze. Della precedente iterazione abbiamo mantenuto l'eseguibile per rendere disponibili i tiles offline.

3.2 Progettazione e Implementazione

In questa iterazione si è scelto di utilizzare il framework *Ionic React*[12]: versione *React*[13] di *Ionic*[14], per sviluppare il frontend. Abbiamo scelto Ionic React così da creare app per dispositivi mobili e desktop utilizzando *Capacitor*[15] e *Electron*[16].

Ionic è un framework di riferimento per lo sviluppo industriale. Le applicazioni aziendali sono, per loro natura, semplici e raramente funzionano con milioni di utenti: devono essere affidabili e intuitive. Ionic, quindi, risulta essere un framework affidabile che consente di ridurre i costi e la durata dello sviluppo pur offrendo un'esperienza utente eccellente e prestazioni stabili. [17]

Ionic React utilizza standard web ed è compatibile con librerie web come *OpenLayers*[18]: libreria che permette di visualizzare tiles, marker e dati vettoriali come heatmap e centroidi.

Di seguito riportiamo il codice relativo al rendering della mappa utilizzando OpenLayers. Si può notare che *MapComponent* è una React Functional Components: una funzione i cui vincoli sono accettare props e restituire JSX. Questo tipo di componenti si distingue dalle classi per la mancanza di metodi di stato e ciclo di vita. Questo è il motivo per cui i componenti funzionali sono anche comunemente indicati come componenti senza stato. I componenti funzionali producono meno codice e bundle più veloci; rimuovendo lo stato a livello di funzione, rendiamo i componenti più facili da usare e più ampiamente applicabili.

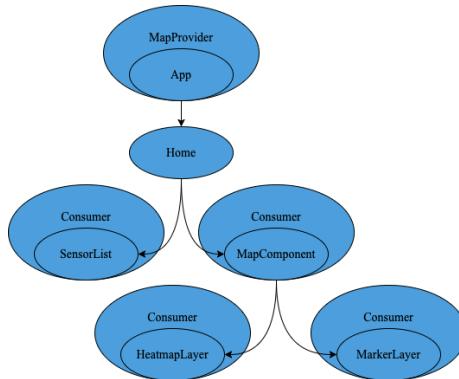


Figura 3.1: React Context

```

1  export const MapComponent: React.FC = () => {
2      const mapDivRef = useRef<HTMLDivElement>(null);
3      const [map] = useState<Map>(new Map({
4          layers: [
5              new TileLayer({
6                  source: new XYZ({
7                      url: "/assets/tiles/{z}/{x}/{y}.png",
8                  })
9              }),
10         ],
11         view: new View({
12             center: fromLonLat([0,0]),
13             zoom: 15,
14             maxZoom: 22,
15             minZoom: 2
16         })
17     }));
18
19     return (
20         <div className="map" ref={mapDivRef}>
21             <HeatmapLayer map={map} />
22             <MarkerLayer map={map} />
23         </div>
24     );
25 }

```

Per lo state management sfruttiamo *Context provides*[19] il quale fornisce un modo per passare dati attraverso l'albero dei componenti senza dover passare manualmente le prop ad ogni livello. Il Context è progettato per condividere dati che possono essere considerati "globali" per un albero di componenti, come l'utente autenticato corrente, il tema o nel nostro caso tutti gli elementi necessari per controllare il rendering della mappa. (component tree in Figura 3.1).

```

1  export const MapContext = React.createContext<ContextType | null>(
2      null);
3
3  export const MapProvider: React.FC<React.ReactNode> = ({ children
4      }) => {
5      const [sensors, setSensorsLocal] = React.useState(new
6          VectorSource());
5      const [centerChangeListeners] = React.useState<Array<
7          MapCenterListener>>(new Array());
6

```

```

7      const setSensors = async (vector: VectorSource) => {
8          setSensorsLocal(vector);
9      }
10
11     const goToLocation = (location: LocationType) => {
12         centerChangeListeners.forEach((it) => it.notify(location));
13     }
14
15     return (
16         <MapContext.Provider value={{goToLocation, sensors,
17             setSensors}}>
18             {children}
19         </MapContext.Provider>
20     );
21 };

```

Come mostrato in Figura 3.2 si è scelto di mantenere la medesima struttura del primo prototipo per quanto riguarda la User Interface.

3.3 Validazione e rilascio

Terminata l'attività di implementazione abbiamo soddisfatto tutti i requisiti che ci eravamo prefissati di sviluppare in questa iterazione.

Si è notato che le informazioni aggiunte alla mappa, come marker e heatmap, possono essere elemento di disturbo nella situazione in cui l'operatore voglia visualizzare la mappa utilizzando una scala molto grande, per questo nelle prossime iterazioni si è scelto sviluppare un menu per selezionare quali layer mostrare.

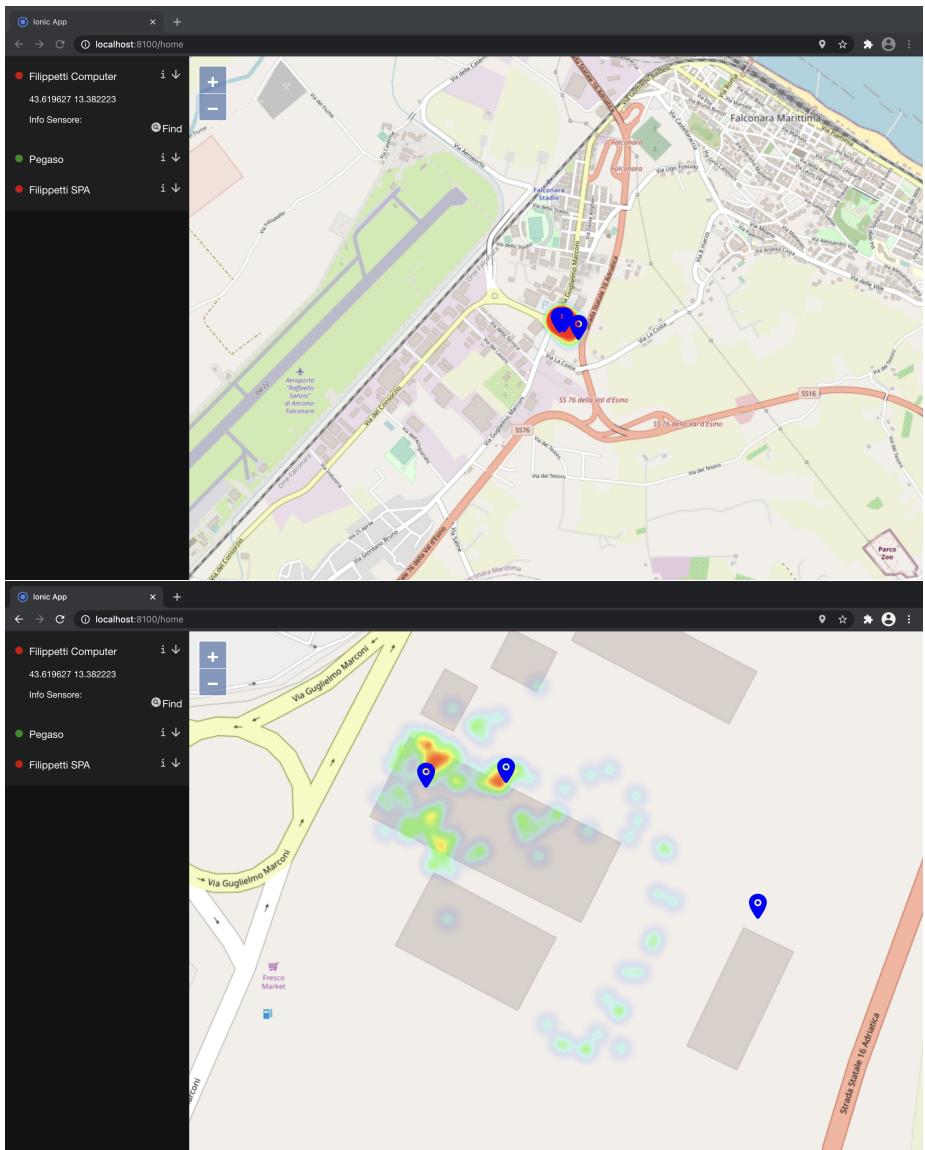


Figura 3.2: IonicReact

Capitolo 4

Terza iterazione

4.1 Requisiti

Tra i requisiti individuati inizialmente abbiamo deciso di aggiungere al prototipo validato nella precedente iterazione, le funzionalità riguardanti la geolocalizzazione del dispositivo e implementare le chiamate Rest per il recupero dei dati dal "server" *Device Locator* (Figura 4.4). Da questa iterazione lo sviluppo non si basa più su file mockup salvati localmente ma sono state rese disponibili le Rest API attraverso cui reperire le informazioni.

Inoltre, si è deciso di implementare la funzionalità risultante dai feedback della seconda iterazione: un menù per selezionare quali layer mostrare sulla cartina.

4.2 Modelli

Nel diagramma dei casi d'uso della terza iterazione sono stati aggiunti due casi d'uso (Figura 4.1):

- **Geolocalizzare dispositivo**

Viene avviato dall'attore *Rescuer*, permette di centrare la mappa nella posizione del dispositivo (Rispettivo sequence diagram nella figura 4.2).

- **Modificare layers**

Viene avviato dall'attore *Rescuer* quando desidera modificare quali layers visualizzare tra: MarkerLayer, HeatmapLayer e GeolocationLayer (Diagramma di sequenza corrispondente in figura 4.3).

4.3 Progettazione e Implementazione

Per individuare la posizione geografica del dispositivo utilizziamo *Cordova Plugin Geolocation*[20] il quale sfrutta il sistema GPS o network signal: WiFi e GSM, per dedurre latitudine e longitudine.

```
1 import { Plugins } from "@capacitor/core";
2 ...
3 ...
```

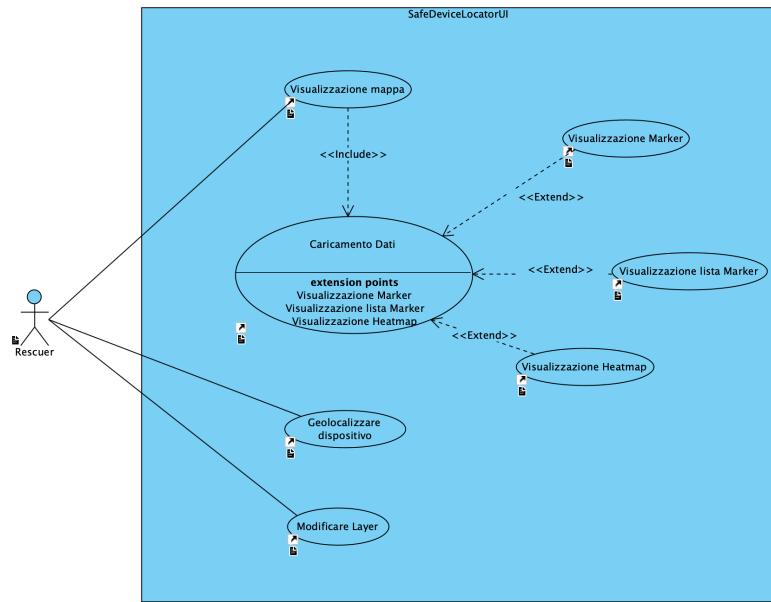


Figura 4.1: Use Case

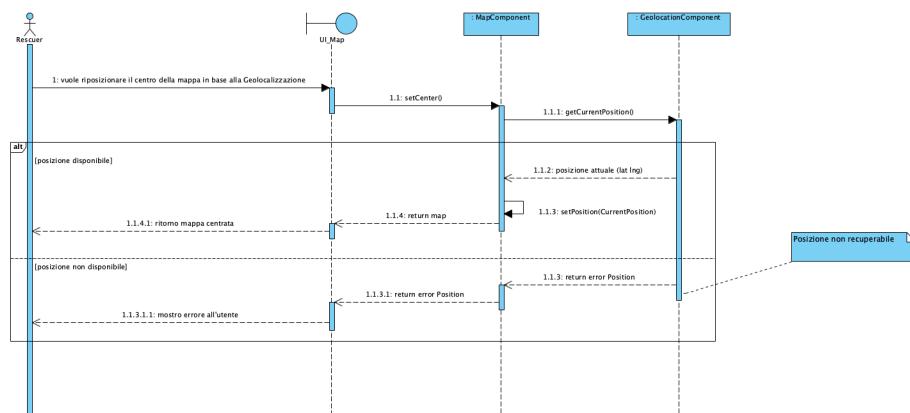


Figura 4.2: SD: Geolocalizzare dispositivo

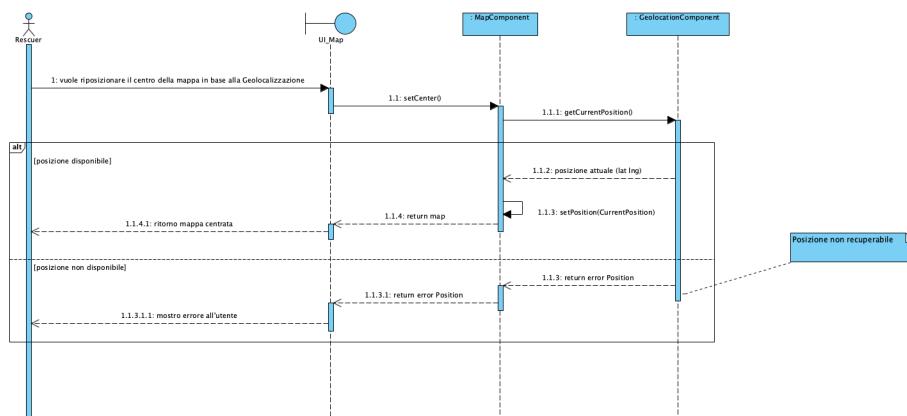


Figura 4.3: SD: Modificare Layer

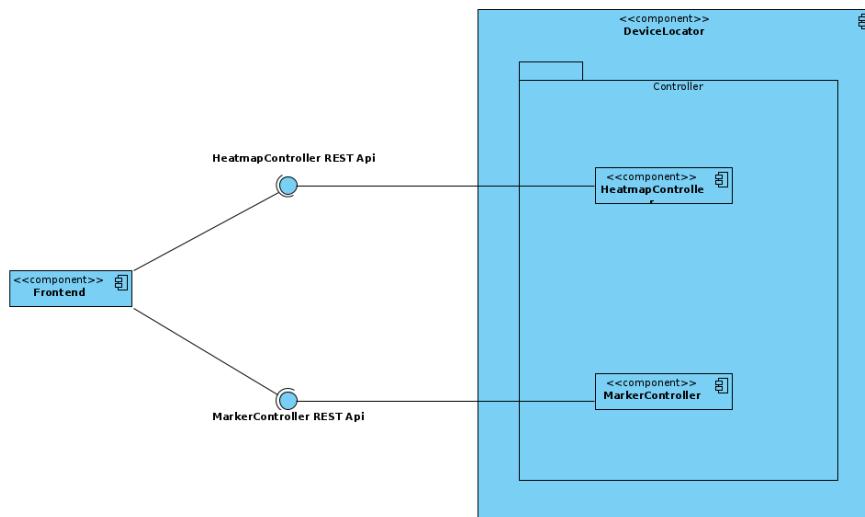


Figura 4.4: Modello dei componenti

```

4      *\` 
5  const { Geolocation } = Plugins;
6
7  export const GeolocationLayer: React.FC<MapLayerProps> = ({ map })
8    => {
9      const { locationVisible, geolocation, startLocationListeners }
10        = useContext(MapContext) as ContextType;
11      const [iconFeature] = useState<Feature>(new Feature({
12        geometry: new Point(fromLonLat([geolocation.lon,
13          geolocation.lat])),
14        name: 'Null Island'
15      }));
16      const [layer] = useState<BaseLayer>(new VectorLayer({
17        source: new VectorSource({
18          features: [iconFeature],
19        }),
20        className: "position",
21        visible: locationVisible,
22      }));
23      useEffect(() => {
24        iconFeature.setStyle(new Style({
25          image: new Icon({
26            src: "./assets/icon/location.svg",
27            color: "green"
28          }),
29        }));
30        map.addLayer(layer);
31        startLocationListeners();
32      }, []);
33
34      layer.setVisible(locationVisible);
35      iconFeature.setGeometry(new Point(fromLonLat([geolocation.lon,
36        geolocation.lat])));
37
38      return null;
39    }

```

Osservando la riga 8 è possibile notare che al *MapProvider* sono stati aggiunti elementi necessari per la geolocalizzazione. All'avvio dell'applicativo la mappa viene centrata sulla posizione geografica del dispositivo e viene avviata la modalità *LocationListeners* (Riga 31): la mappa segue gli spostamenti del device. Però, se l'operatore decide di spostare la visuale della mappa su un punto diverso da quella del dispositivo viene interrotta la modalità *LocationListeners*. Per centrare nuovamente la visuale della mappa sul dispositivo e riabilitare la modalità "inseguimento" è sufficiente premere l'apposito pulsante.

Come mostrato nella Figura 4.5 sono stati aggiunti due bottoni sul lato destro dell'interfaccia. Il pulsante in alto a destra permette di visualizzare un menu all'interno di un popover contenente la lista dei layer disponibili e i relativi switch per abilitarne o meno la sovrappressione della mappa. Mentre, il pulsante nell'angolo inferiore destro abilita le funzioni di geolocalizzazione.

Per quanta riguarda il recupero dei dati da renderizzare sulla mappa è stato sufficiente sostituire l'url locale dei file mockup con l'indirizzo dei dati reali presenti sul server Device Locator. Ispezionando il codice relativo all'*HeatmapLayer* è possibile notare che la riga 5 contiene l'indirizzo reale del server.

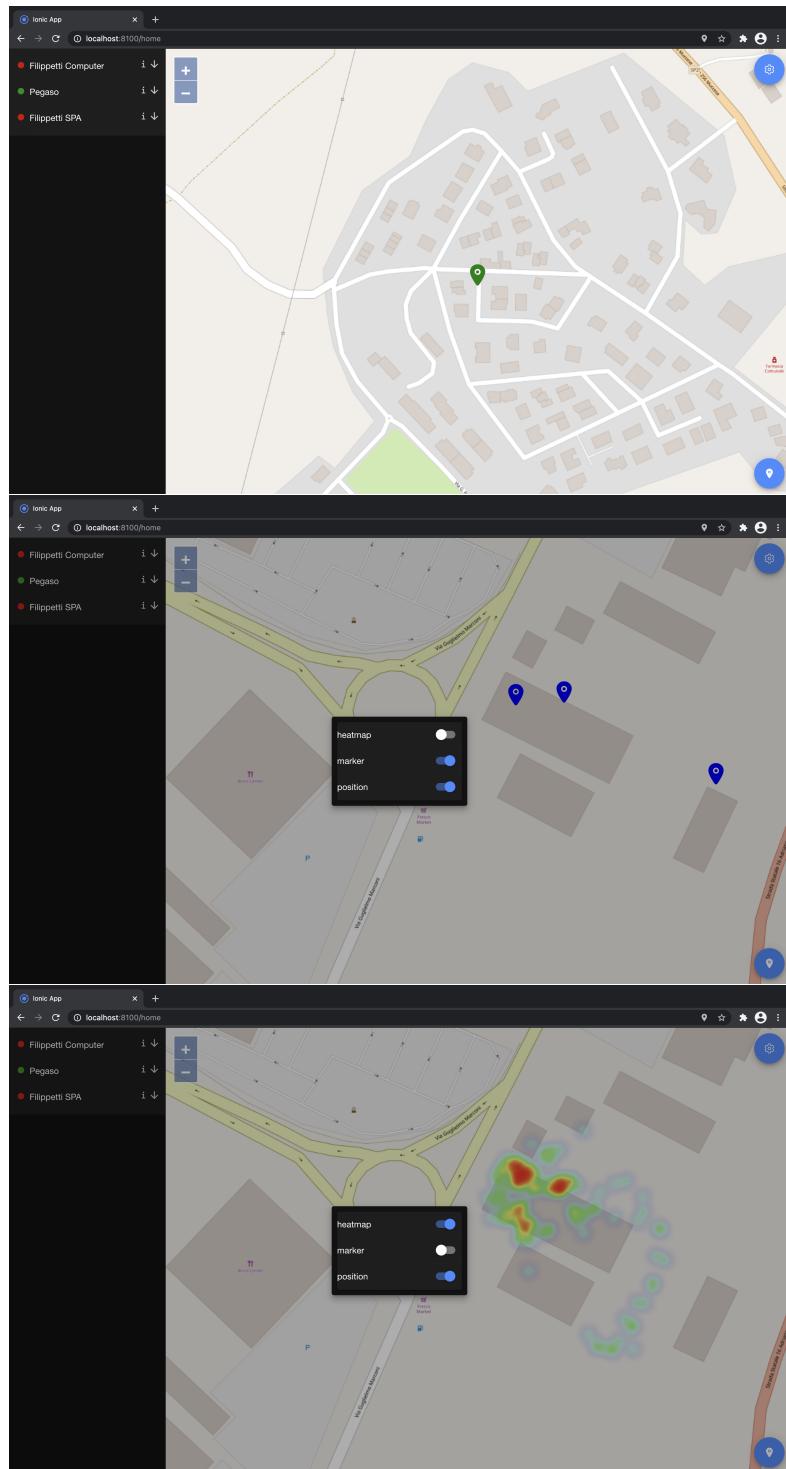


Figura 4.5: Geolocalizzazione e menu layers

```

1  export const HeatmapLayer: React.FC<MapLayerProps> = ({ map }) => {
2    const { heatmapVisible, radius, blur } = useContext(MapContext)
3      as ContextType;
4    const [source] = useState<VectorSource>(
5      new VectorSource({
6        url: 'http://135.181.224:5000/api/sampling/heatmap?sensor
7          =0504&format=kml',
8        format: new KML({
9          extractStyles: false,
10         }),
11       })
12     );
13   const [layer] = useState<Heatmap>(
14     new Heatmap({
15       className: "heatmap",
16       visible: heatmapVisible,
17       source: source,
18       blur: blur,
19       radius: radius,
20       weight: function (feature) {
21         return feature.get('name');
22       },
23     })
24   );
25   useEffect(()=>{
26     map.addLayer(layer);
27   },[]);
28   layer.setVisible(heatmapVisible);
29   layer.setBlur(blur);
30   layer.setRadius(radius);
31   return null;
32 }

```

4.4 Validazione e rilascio

Conclusa l'implementazione sono stati rispettati tutti i requisiti prestabiliti in questa iterazione. Utilizzando il prototipo si è notato che in alcune situazioni il soccorritore potrebbe voler visualizzare layer differenti relativi a sensori differenti perciò nelle iterazione successivi si prevede di sviluppare un menu per gestire i layer individualmente per ogni sensore.

Capitolo 5

Quarta iterazione

5.1 Requisiti

Nell'attuale iterazione sono stati selezionati i seguenti requisiti da implementare:

- Rendering di dati vettoriali come poligoni.
- Mantenimento dello stato in caso di terminazione.
- Menu layer individuale per ogni sensore

5.2 Modelli

Nel diagramma dei casi d'uso della quarta iterazione è stato aggiunto il caso d'uso relativo alla visualizzazione di poligoni. Il caso d'uso in questione estende il caso d'uso *Caricamento dati* e permette di rappresentare graficamente di poligoni sopra la mappa (Figura 5.1).

5.3 Progettazione e Implementazione

Per il mantenimento dello stato dell'applicazione è stato necessario memorizzare informazioni come Layers visibili, Radius e Blur delle heatmaps oppure se è stato controllato o meno un sensore, all'interno del dispositivo. Ionic mette a disposizione due diverse opzioni per l'archiviazione dei dati all'interno di un'app: *Ionic Secure Storage* e *Ionic Storage*. Ionic Secure Storage serve per creare app mission-critical o che richiedono supporto per la crittografia, invece Ionic Storage serve per tutte quelle applicazioni che non richiedono né la crittografia né il supporto a dati relazionali, come nel nostro caso.

```
1 class StorageService {
2     storage = new Storage();
3
4     constructor() {
5         this.storage.create();
6     }
7
8     async saveSensorLocal(value: Array<Sensor>) {
9         this.storage.set("SensorsLocal", JSON.stringify(value));
10    }
```

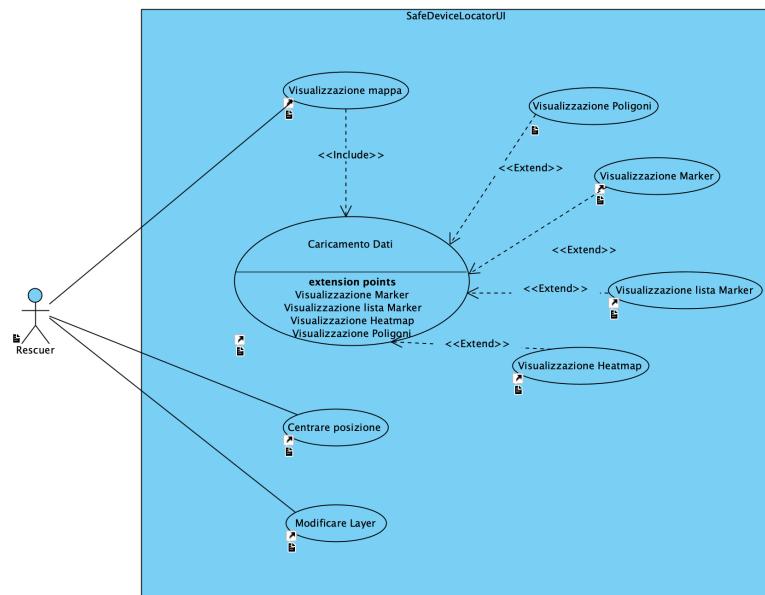


Figura 5.1: Use Case

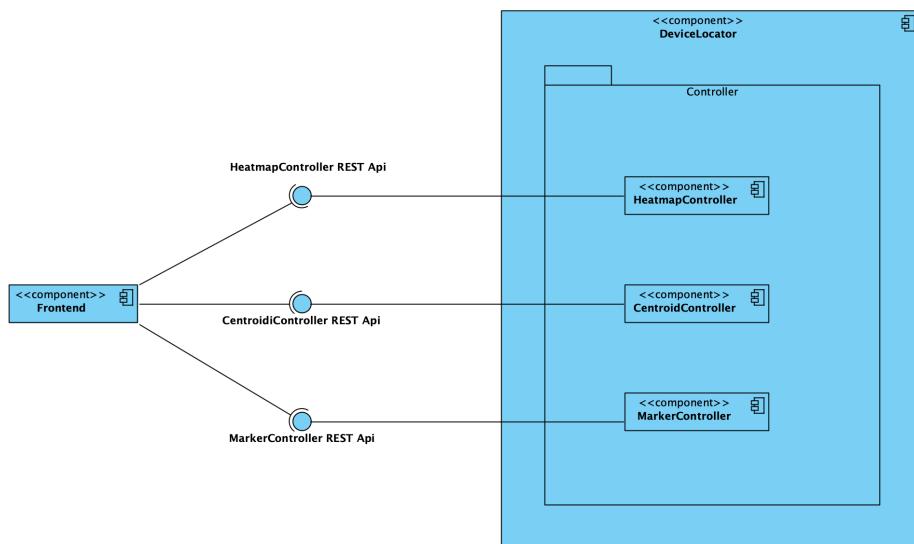


Figura 5.2: Modello dei componenti

```

10     }
11
12     async getSensorLocal(): Promise<Array<Sensor>> {
13         const sensorsLocal = await this.storage.get("SensorsLocal")
14         ;
15         if(!sensorsLocal) return new Array();
16         return JSON.parse(sensorsLocal);
17     }
18
19     /*
20     ...
21     */
22 }

```

Osservando il codice è possibile notare che è stata implementata un'apposita classe per la memorizzazione dei dati, al cui interno sono stati implementati due metodi per ogni parametro: uno per archiviare i dati e uno per recuperarli, nel caso in cui il dato non è presente nello store viene restituito un valore di default.

Nell'esempio sopra riportato si possono vedere i metodi relativi a memorizzare le informazioni relative ai sensori. Rispettivamente per ogni sensore memorizziamo l'id, lo stato (se è stato recuperato o meno), e le informazioni per visualizzare i layer sulla mappa corrispondenti al sensore. Per fare ciò abbiamo implementato la seguente interfaccia con all'interno le informazioni precedentemente elencate.

```

1 export interface Sensor {
2     id: string;
3     status: boolean;
4     isHeatmapVisible: boolean;
5     isMarkerVisible: boolean;
6     isCentroidVisible: boolean;
7     heatmapRadius: number;
8     heatmapBlur: number;
9 }

```

Memorizzando tali informazioni e modificando i props dei layer: HeatmapLayer, MarkerLayer e CentroidLayer, saremo in grado di disegnare e gestire la visibilità dei layer individualmente per sensore. I layer, nello specifico, oltre a prendere in input il riferimento alla mappa prenderanno anche il riferimento all'oggetto sensore che dovranno rappresentare e "monitorando" le informazioni memorizzate al suo interno si mostrerà o meno tale layer.

```

1 export const CentroidsLayer: React.FC<SensorLayerProps> = ({ map,
2     sensor }) => {
3
4     var source = new VectorSource({
5         format: new GeoJSON(),
6     });
7
8     const [layer] = React.useState<VectorLayer>(new VectorLayer({
9         source: source,
10        style: [
11            new Style({
12                stroke: new Stroke({
13                    color: "blue",
14                    width: 3,
15                }),
16                fill: new Fill({

```

```

16         color: "rgba(0, 0, 255, 0.1)",
17     }),
18   },
19   new Style({
20     image: new CircleStyle({
21       radius: 5,
22       fill: new Fill({
23         color: "orange",
24       }),
25     }),
26   ],
27 });
28 });
29
30 useEffect(() => {
31   fetch(
32     "http://127.0.0.1:1234/api/centroid-area?sensor=" + sensor.id
33     + "&format=json"
34   )
35     .then((response) => response.json())
36     .then((response) => {
37       console.log("TEST", JSON.stringify(response));
38       source.addFeatures(new GeoJSON().readFeatures(response));
39       map.addLayer(layer);
40     });
41   }, []);
42   layer.setVisible(sensor.isCentroidVisible);
43
44   return null;
45 };

```

Riportiamo il layer che si occupa del rendering dei poligoni sulla mappa, sfruttando `useEffect`(Riga 30) comunichiamo a React che il layer deve fare qualcosa dopo il rendering. React ricorderà la funzione passata e la chiamerà dopo aver eseguito gli aggiornamenti del DOM. In questo modo recuperiamo le coordinate dei punti dei poligoni chiamando il modulo *DeviceLocator* e aggiungiamo il nuovo layer alla mappa. Per consentire al soccorritore di modificare la visibilità dei layer per ogni singolo sensore è stato modificato il *popover* implementato nella precedente iterazione al quale passiamo attraverso i props il riferimento al sensore che dovranno gestire.

5.4 Validazione e rilascio

Conclusa l'iterazione abbiamo terminato quasi tutti i requisiti che ci eravamo prefissati di sviluppare a inizio progetto, osservando Figura 5.3 è possibile notare che dal menu relativo al sensore "0504" è stata nascosta la rispettiva heatmap, con il medesimo meccanismo è possibile modificare la visibilità dei layer che si riferiscono al sensore stesso.



Figura 5.3: Terzo prototipo Ionic

Capitolo 6

Quinta iterazione

6.1 Requisiti

Nella seguente iterazione andiamo ad implementare le funzionalità per la gestione delle squadre e il download dei pacchetti contenenti le mappe così da soddisfare tutti i requisiti inizialmente prefissati. Inoltre, per concludere il progetto si è deciso di aggiungere dei controlli inerenti alla mancanza di rete nella fase di download dei tiles e la gestione di eventuali errori di comunicazione con l'applicativo che ha il compito di passare i dati dei sensori, infine si vuole apportare delle migliorie grafiche in quest'ultima iterazione.

6.2 Modelli

Per effettuare il download delle mappe sul dispositivo si prevede la presenza di un server il quale metterà a disposizione i pacchetti *.zip* contenenti tiles attraverso chiamate Rest (Figura 6.2).

Nel diagramma dei casi d'uso della quinta iterazione sono stati aggiunti due casi d'uso (Figura 5.1):

- **Inserimento squadre**

Viene avviato dal Rescuer e consente di inserire le squadre disponibili per le attività di soccorso.

- **Assegnare sensori alle squadre**

Viene avviato dal Rescuer consentendogli di associare alle squadre i sensori di cui dovranno occuparsi.

- **Download mappe**

Viene avviato dal Rescuer permettendo di scaricare le mappe offline sul dispositivo.

6.3 Progettazione e implementazione

Per la gestione delle squadre prevediamo che un soccorritore inserisca l'elenco delle squadre all'interno del sistema e successivamente assegni ad ogni squadra

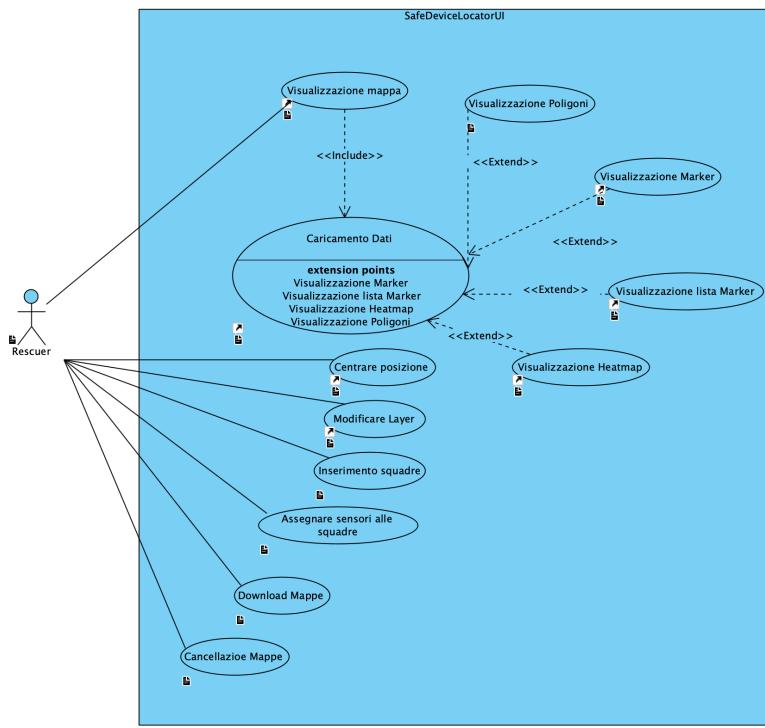


Figura 6.1: Use Case

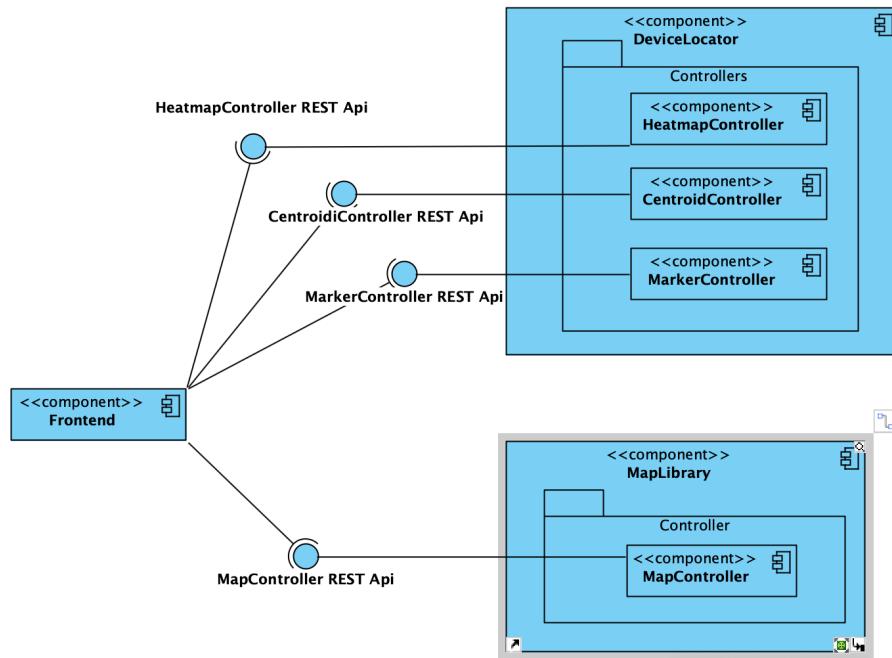


Figura 6.2: Component Diagram

i sensori di cui dovranno occuparsi. È stato necessario introdurre una nuova schermata nell'applicazione (Figura 6.3) al cui interno è presente un pulsante che abilita un popover per inserire le squadre, una barra di ricerca per filtrare le squadre già inserite e per ogni squadra inserita sarà possibile visualizzare i sensori disponibili. Per fare ciò memorizziamo la lista delle squadre all'interno dello store dell'applicazione e aggiungere un campo all'interno dei sensori così da associare la squadra al sensore. Mediante l'utilizzo di filtri, durante la fase di accoppiamento sensori-squadre, è possibile ricercare i sensori per id e visualizzare solamente i sensori ancora non assegnati a nessuna squadra. Nel caso in cui una squadra venga eliminata dal sistema tutti i sensori precedentemente assegnati ad essa verrano resi nuovamente disponibili per essere assegnati ad altre squadre, di seguito riportiamo la funzione che si occupa di fare l'update dei sensori e la cancellazione della squadra.

```

1 function removeTeam() {
2     sensors.map((e) => {
3         if (e.team === team) e.team = "";
4     });
5     setSensors(sensors);
6     teams.splice(teams.indexOf(team), 1);
7     setTeams(teams);
8 }

```

In merito al download dei pacchetti contenenti le mappe è stato necessario implementare un plug-in nativo android denominato *JarvisTransfer*. Ionic consente di sviluppare progressive web app, cioè applicazioni web che si comportano in modo simile alle applicazioni native quando utilizzate su un dispositivo mobile. Proprio per questa caratteristica di Ionic, essendo di base un'applicazione web, non ha accesso diretto al FileSystem e non ci consente di effettuare il download e l'unzip del pacchetto contenente la mappa. Il plug-in in questione è stato scritto in Java e non in Kotlin in quanto Capacitor utilizza per impostazione predefinata Java. Nel caso in cui si volesse utilizzare Kotlin, sulla documentazione ufficiale di Capacitor è riportato che: dopo aver generato il plug-in Java, attraverso Android Studio bisogna convertire il file Java in file Kotlin.

```

1 public class JarvisTransfer {
2     private final DownloadEventListener listener;
3     private final Unzipper unzipper;
4
5     public JarvisTransfer(DownloadEventListener listener) {
6         this.listener = listener;
7         this.unzipper = new Unzipper();
8     }
9
10    public void downloadFile(String url, String zipPath){
11        new FileDownloader(listener).execute(url, zipPath);
12    }
13
14    public void unzip(String zipPath) throws IOException {
15        unzipper.unzip(zipPath);
16        unzipper.deleteZipFile(zipPath);
17    }
18
19    public boolean reset(String folderPath){
20        File file = new File(folderPath);
21        return file.delete();
22    }
23 }

```

	▼
<input type="text"/> Filter Team	INSERT NEW TEAM
Pippo	
Pluto	
Paperino	

	▼
<input type="text"/> Filter Team	INSERT NEW TEAM
Pippo	
<input type="text"/> Search	view available
	<input checked="" type="checkbox"/> DELETE TEAM
0505	
0504	
Pluto	
<input type="text"/> Search	view available
	<input checked="" type="checkbox"/> DELETE TEAM
0504	
Paperino	
<input type="text"/> Search	view available
	<input checked="" type="checkbox"/> DELETE TEAM
0505	
0504	

Figura 6.3: Team page

La classe "Facade" JarvisTransfer sopra riportata mette a disposizione tre metodi:

- **downloadFile** Prende in input l'url dove effettuare la chiamata e la path di dove salvare il pacchetto zippato da scaricare.
- **unzip** Presa in input la path del pacchetto effettua l'unzip ed elimina il file zip.
- **reset** Presa in input una path elimina tutti i file presenti in essa.

Sfruttando questi tre metodi è possibile gestire le varie porzioni di mappa da memorizzare sul dispositivo ed eliminarle una volta termiante le operazioni di soccorso così da liberare spazio sul dispositivo stesso.

Per quanto riguarda il server da cui distribuire le mappe offline è stato necessario incorporare il codice del eseguibile Rust della prima iterazione all'interno di un progetto *Rocket*[21]. Rocket è un framework Web per Rust che semplifica la scrittura di applicazioni Web veloci e sicure senza sacrificare flessibilità, usabilità o sicurezza dei tipi. Con il nuovo eseguibile Rust mettiamo a disposizione due end-point:

- **list/available** Restituisce un Json contenente la lista di tutte città disponibili per il download.
- **download/<city_name>** Restituisce il pacchetto .zip corrispondente alla città richiesta attraverso <city_name>.

La lista delle mappe disponibili dei download corrisponde alla lista dei pacchetti presenti all'interno del server. Per configurare il server e indicargli quali città dovrà scaricare per rendere disponibili i relativi pacchetti è necessario lanciare l'eseguibile con il comando **cargo run init**. Al primo avvio verrà restituito un warning in cui viene indicata l'assenza del file di configurazione e che ne è stato generato uno di default, tale file viene memorizzato nella path:

```
$HOME/.config/safe-backend/safe-backend.toml
```

Se già esiste un file di configurazione ma risulta corrotto ne viene creato uno nuovo. Una volta modificato il file **safe-backend.toml** sarà possibile eseguire nuovamente il comando **cargo run init** il quale procederà con il download dei pacchetti. Una volta che i pacchetti sono presenti sul server non sarà più necessario eseguire il comando **cargo run init** ma sarà sufficiente lanciare il comando **cargo run** il quale abiliterà gli end-point senza effettuare nuovamente il download. Nella situazione in cui viene eseguito l'eseguibile con il comando **init** ma sono già presenti dei pacchetti all'interno del server, quest'ultimi verranno scaricati nuovamente solamente nel caso in cui abbiano una data di creazione maggiore di sei mesi.

```
1 pub struct AppConfig {  
2     pub cities: Vec<String>,  
3     pub min_zoom: i32,  
4     pub max_zoom: i32,  
5     pub api_key: String,  
6 }
```

All'interno del file `safe-backend.toml` viene memorizzata la struttura dati sopra riportata composta dalla lista di città con il rispettivo CAP, il livello minimo e massimo di zoom che dovranno essere scaricati per ogni pacchetto e in fine l'api key, richiesta dall'API per determinare la posizione delle città. Di seguito riportiamo un esempio di file di configurazione.

```
1 cities = ["camerino 62032", "cingoli 62011"]
2 min_zoom = 15
3 max_zoom = 22
4 api_key = '6124b12559354447bfa6fd61c1316325'
```

Al fine di conoscere le coordinate geografiche di ogni città da scaricare sfruttiamo L'API OpenCage Geocoding il quale fornisce API Rest che ci permettono di ottenere non solo le coordinate del centro ma anche informazioni più dettagliate come la sua estensione.

Per effettuare il download dei pacchetti sul dispositivo l'utente dovrà visualizzare l'apposita pagina *Map* dove verranno visualizzate tutte le mappe disponibili e premendo sul relativo pulsante del pacchetto desiderato verrà avviato il download (Figura 6.4).

Per la gestione degli errori causati dalla mancanza di rete (Figura 6.5), durante l'attività di download dei pacchetti, si è utilizzato il Plugin di Capacitor *NetworkPlugin*. Nella fase iniziale, se non vengono trovati tiles localmente, viene effettuato un controllo sullo stato della rete, fino a quando non è disponibile la connessione internet viene mostrato un Popover il quale mostra un messaggio di errore e richiede il ricarcamento della pagina. Un'altro controllo effettuato riguarda il recuperare dei sensori all'avvio (Figura 6.6), se si riceve un'eccezione dalla richiesta Rest verso il DeviceLocator per il recupero dei sensori, si mostra un Popover dove viene richiesto di attivare l'applicativo. Se localmente sono presenti i dati dei sensori ma il DeviceLocator non risponde, verrà visualizzato il messaggio d'errore corrispondente (Figura 6.7).

Relativamente alle migliori grafiche, dietro indicazioni del cliente, abbiamo aggiunto i loghi del progetto nelle varie schermate e sfruttando i css siamo andati a customizzare i componenti messi a disposizione da Ionic così da rendere l'applicazione più user-friendly (Figura 6.8).

6.4 Validazione e rilascio

Conclusa questa iterazione riteniamo di aver soddisfatto a pieno tutte le funzionalità richieste e che non ci sia la necessità di iniziare nuove iterazioni.



S.A.F.E.
[red icon]

castelfidardo_60022	[blue box with white checkmark]
camerino_62032	[blue box with white checkmark]
cingoli_62011	[blue box with white checkmark]



S.A.F.E.
[red icon]

castelfidardo_60022	[blue box with white checkmark]
camerino_62032	[blue box with white checkmark]
cingoli_62011	[blue box with white checkmark]



Figura 6.4: Map page

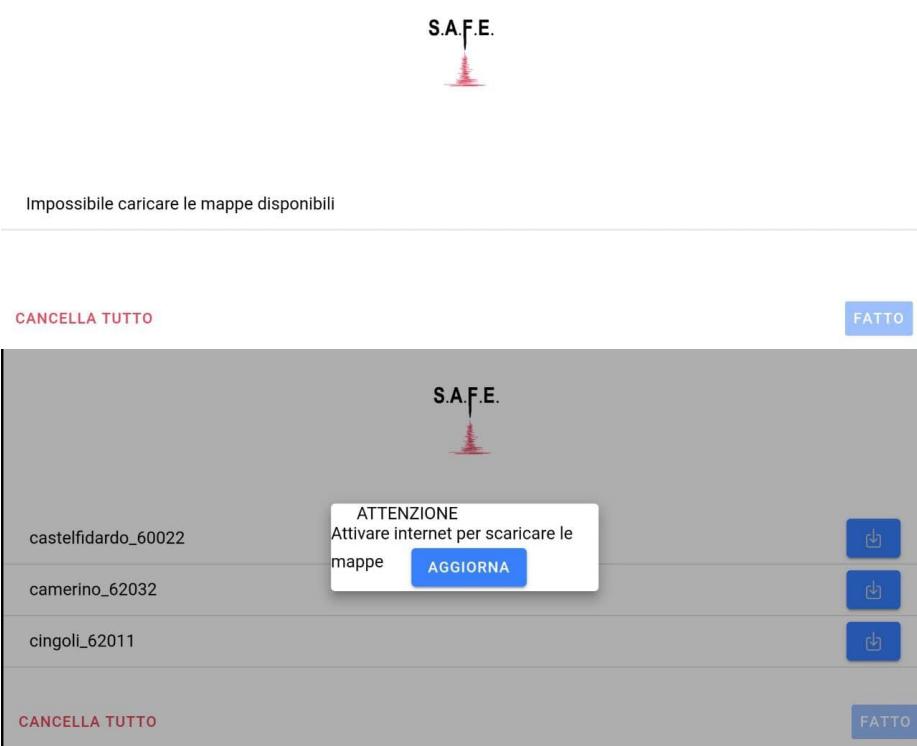


Figura 6.5: internet error

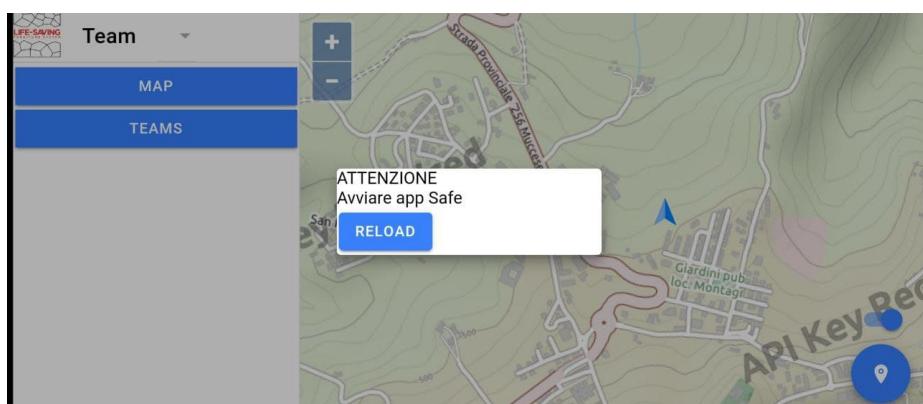


Figura 6.6: sensors first init error

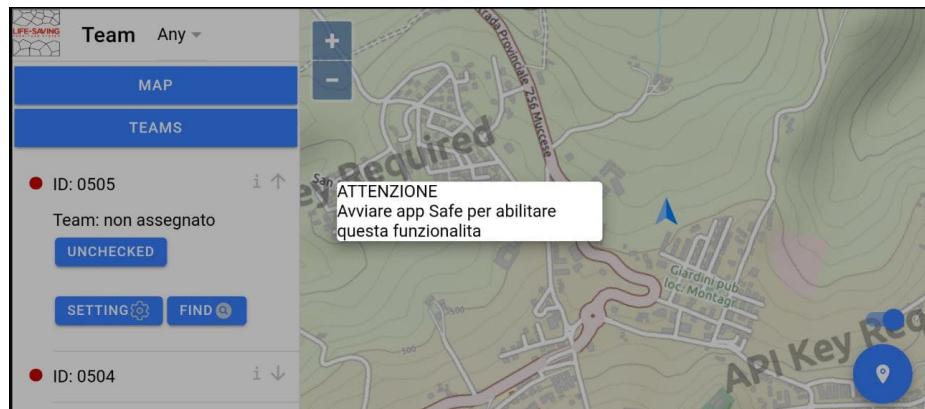


Figura 6.7: sensor location error

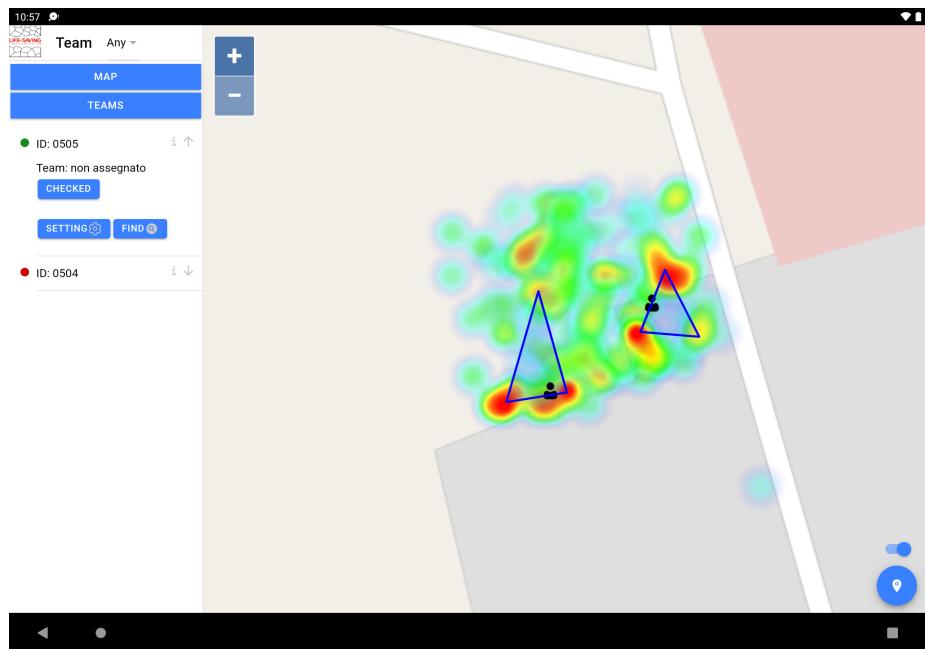


Figura 6.8: Home page

Capitolo 7

Deployment

7.1 Configurazione

7.1.1 Frontend

Per effettuare la compilazione dell'applicazione frontend è necessario avere installato sul proprio computer Nodejs, il quale è possibile scaricarlo direttamente dal sito ufficiale. È possibile controllare che Nodejs sia installato correttamente lanciando il seguenti comandi:

```
1 $ node --version  
2 $ npm --version
```

Successivamente è necessario installare il framework Ionic, è possibile installarlo usufruendo dal package manager di Node npm utilizzando il comando¹:

```
1 $ npm install -g @ionic/cli
```

Frontend

Ionic permette di compilare l'applicazione per android generando un file .apk che è possibile installare su un qualsiasi dispositivo android. Per fare ciò è necessario installare Android Studio che si può scaricare dal sito ufficiale. Al termine dell'installazione avviare Android Studio che tramite una procedura guidata permetterà di installare gli Android SDK che saranno necessari per la compilazione.

Per configurare il progetto Android lanciare il seguente comando dalla directory "frontend" del progetto:

```
1 $ ionic capacitor add android
```

Questo aggiungerà al progetto la cartella "android" che conterrà il progetto android.

¹l'argomento *-g* indica che l'installazione verrà eseguita a livello globale quindi su alcuni sistemi come linux o mac potrebbe essere necessario lanciare il comando da una shell con privilegi elevati (oppure utilizzare sudo).

7.1.2 Backend

Il backend è scritto in Rust, per compilare il programma è necessario installare gli strumenti di sviluppo di Rust tramite il sito ufficiale e seguire le istruzioni per il proprio sistema operativo. In particolare è importante che siano disponibili nel proprio sistema i comandi:

- **rustup**: strumento utilizzato per il download e l'aggiornamento di rust.
- **cargo**: strumento per compilare ed eseguire il codice rust.

7.2 Compilazione

Frontend

Dopo aver terminato la fase di configurazione è possibile compilare il progetto android lanciando i seguenti comandi dalla cartella **frontend** del progetto.

```
1 $ ionic build
2 $ ionic capacitor copy android --no-build
3 $ ionic capacitor open android
```

Una volta lanciato l'ultimo comando, si sarà avviato Android Studio. Da qui è possibile generare un file di installazione .apk dal menù:
Build->Build Boundle(s) / APK(s)->Build APK(s).

Backend

Conclusa la fase di configurazione è possibile procedere con la compilazione dell'eseguibile Rust lanciando il comando sotto riportati nella cartella **backend**.

```
1 $ cargo build --release
```

Terminata la compilazione, all'interno della cartella **backend/target/release**, sarà presente l'eseguibile **backend**.

```
1 $ ./backend init
2 $ ./backend
```

Spostandoci all'interno della cartella **release** ed eseguendo il primo comando sopra riportato verrà avviata l'inizializzazione del server e successivamente saranno esposti gli end-point. Mentre, eseguendo il secondo comando (senza "init") verranno solamente esposti gli end-point.

7.3 Requisiti

Frontend

Per eseguire i test dell'applicativo, abbiamo utilizzato un tablet android con le seguenti caratteristiche :

- Snapdragon 865 Plus
- Display 12.4"
- RAM 6GB

- Android 11

Visto anche ulteriori test svolti con vari emulatori, consigliamo l'utilizzo di un device con almeno le seguenti specifiche :

- processore 1.9GHz
- Display di 7"
- 2 GB
- Android 10/11/12
- sistema di geolocalizzazione integrato

Backend

Per il back end (utilizzato per inizializzare i Tiles in un momento di pace), si consiglia l'utilizzo di un qualunque server cloud (Windows o Linux) per gestire il servizio, che sia dotato di una buona connessione di rete e una media potenza computazionale per un server (utilizzata per effettuare il zip dei tiles).

Bibliografia

- [1] *S.A.F.E Project.* URL: <http://www.safeproject.it>.
- [2] Ramon Sanchez-Iborra et al. «Performance Evaluation of LoRa Considering Scenario Conditions». In: *Sensors* 18.3 (2018). URL: <https://www.mdpi.com/1424-8220/18/3/772>.
- [3] Google. *Flutter*. URL: <https://flutter.dev/>.
- [4] Google. *Dart*. URL: <https://dart.dev/>.
- [5] Mobindustry. *Top 7 Reasons to Choose Flutter for Your Cross-platform App Development Project*. URL: <https://medium.com/mobindustry/top-7-reasons-to-choose-flutter-for-your-cross-platform-app-development-project-62cf35133d37>.
- [6] John Ryan. *Flutter_Map*. URL: https://pub.dev/packages/flutter_map.
- [7] Dash-Overflow. *provider*. URL: <https://pub.dev/packages/provider>.
- [8] Rust Project Developers. *Rust*. URL: <https://www.rust-lang.org/>.
- [9] Fortune Ikechi. *Why is Rust so popular?* URL: <https://blog.logrocket.com/why-is-rust-popular/>.
- [10] Andy Allan. *Thunderforest*. URL: <https://www.thunderforest.com/>.
- [11] Steve Coast. *OpenStreetMap*. URL: <https://www.openstreetmap.org/>.
- [12] Drifty. *Ionic React*. URL: <https://ionicframework.com/docs/react>.
- [13] Facebook e community. *React*. URL: <https://it.reactjs.org>.
- [14] Drifty. *Ionic*. URL: <https://ionicframework.com/>.
- [15] Ionic. *Capacitor*. URL: <https://capacitorjs.com>.
- [16] GitHub Inc. *Electron*. URL: <https://www.electronjs.org>.
- [17] CPO Kirill Yusov. *Ionic vs React Native: How to Make The Right Choice*. URL: <https://jelvix.com/blog/ionic-vs-react-native>.
- [18] OSGeo. *OpenLayers*. URL: <https://openlayers.org>.
- [19] Facebook e community. *Context provides*. URL: <https://reactjs.org/docs/context.html>.
- [20] apache. *Cordova Plugin Geolocation*. URL: <https://github.com/apache/cordova-plugin-geolocation#readme>.
- [21] Sergio Benitez. *Rocket*. URL: <https://rocket.rs>.