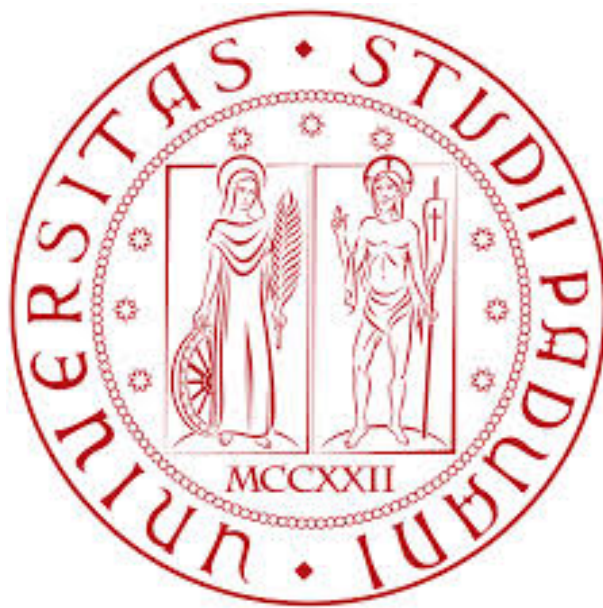


UNIVERSITA' DEGLI STUDI DI PADOVA

**Corso di Laurea Magistrale in Ingegneria
dell'Automazione**

Tesi di laurea

Sistema di Monitoraggio da Remoto per Imbarcazioni:
Interfaccia Utente



Tesista: Conti Matteo
Relatore: Prof. Stefano Vitturi

Anno Accademico 2014-2015

Indice

1	Introduzione	5
2	eKEEPER2, applicazione per Smartphone Android	6
2.1	Breve descrizione dello strumento eKEEPER2	6
2.2	Struttura di un'applicazione Android	8
2.2.1	Activity	9
2.2.2	Ciclo vita di un'applicazione	9
2.2.3	Struttura di un'applicazione	11
2.2.4	AndroidManifest	12
2.2.5	La cartella <i>res</i>	13
2.2.6	<i>strings.xml</i> e multilingua	14
2.3	Programmazione dell'applicazione Android	15
2.3.1	Java ed XML	15
2.3.2	La classe <i>R</i>	16
2.4	Creazione di un'Activity	16
2.5	Grafica di base	17
2.5.1	View e Layout	17
2.5.2	TextView	18
2.5.3	EditText	19
2.5.4	Button	20
2.5.5	ListView	21
2.6	Android API	22
2.6.1	Preparare il progetto	23
2.6.2	Ottenere una API KEY	24
2.6.3	Modificare il Manifest.xml	26
2.7	Gestione dei database SQLite in Android	26
2.7.1	Implementare un database nell'applicazione	26
2.7.2	La classe Adapter	28
2.7.3	Creazione di un record	29
2.7.4	Modifica di un record	30
2.7.5	Eliminazione di un record	30
2.7.6	Utilizzo del database all'interno dell'Activity	31
3	eKEEPER2, utilizzo del server	34
3.1	Creazione di un ambiente LAMP su Linux	34
3.2	Creazione delle tabelle MySQL	34
3.3	Creazione degli script PHP	37
3.3.1	Creazione utente	37
3.3.2	Download ed upload stato imbarcazione	40
3.3.3	Download ed upload numeri abilitati ed allertabili	44
3.3.4	Download ed upload di temperatura ed umidità nel corso delle 24 ore	45
3.3.5	Download ed upload viaggio	46
3.3.6	Script PHP per funzioni ausiliarie	48
4	L'applicazione eKEEPER2	51
4.1	Schermata <i>Home</i> e creazione di un account	51
4.2	Schermata <i>Home</i> ed accesso all'applicazione	55
4.3	TabularActivity, il contenitore di activity	56
4.4	StatoActivity, monitoraggio dell'imbarcazione	59

4.5	DispositiviActivity, la gestione da remoto	61
4.6	MappaActivity, visualizzazione dei viaggi effettuati	67
4.7	Impostazioni: gestione dell'account e dei numeri	72
4.8	SMSActivity, la gestione degli SMS	75
4.9	Analisi consumo dati	78
5	Sviluppi futuri	81

Sommario

eKEEPER2 è un progetto imprenditoriale che si vuole avviare nel settore della strumentazione per le unità da diporto nautico. L'idea di business prevede la realizzazione e vendita di uno strumento per il controllo di alcune funzioni e stati da remoto delle imbarcazioni da diporto, con un approccio al cliente di tipo non convenzionale. Il prodotto è caratterizzato dalla possibilità di controllare da remoto, mediante smartphone, alcune utenze tipiche delle imbarcazioni, ad esempio richiedere ed ottenere informazioni relative a parametri tipo localizzazione della barca, stato delle batterie, presenza di alimentazione elettrica da rete, temperatura ed umidità, oppure ricevere allarmi in caso di situazioni anomale come il superamento del livello di soglia dell'acqua in sentina, spostamento della barca dalla posizione di riferimento o di intrusione. Lo studio del prototipo qui presentato si articola in tre distinti punti:

1. Individuazione dei componenti elettronici per la realizzazione del prototipo, e relativa programmazione
2. Creazione di un'applicazione per Smartphone con sistema operativo Android
3. Allestimento di un server al fine di permettere la comunicazione tra eKEEPER2 e smartphone via internet

Verranno presentati gli aspetti fondamentali dell'implementazione di ognuno dei tre punti, affiancando ad essi i risultati ottenuti dai test sul prodotto ultimato.

1 Introduzione

L'obiettivo che l'azienda Aitech s.r.l si pone è quello di realizzare uno strumento utile al cliente diportista nautico per monitorare la propria imbarcazione da remoto, in qualsiasi istante dell'anno e da ogni luogo in cui si trovi.

L'interfaccia tra strumento e cliente sarà lo smartphone, dispositivo estremamente diffuso, il quale sfrutterà la comunicazione via SMS e/o via internet per mezzo di un'applicazione appositamente studiata. Lo strumento andrà ad integrare la normale strumentazione di bordo, senza sostituirsi ad essa, fornendo un servizio di controllo da remoto con la migliore tecnologia disponibile.

In fase di progettazione sono stati intrapresi i seguenti step:

- Selezione di componenti e fornitori adatti allo scopo
- Realizzazione di un prototipo dello strumento da collaudare e certificare prima del lancio nel mercato del prodotto finito
- Implementazione del software per il funzionamento del prodotti e per la gestione dei prodotti venduti
- Realizzazione dell'applicazione per smartphone

La missione di Aitech s.r.l., pertanto, è fare di eKEEPER2 un punto di riferimento per chi vuole integrare la propria strumentazione di bordo con un sistema di controllo da remoto che gli permetta di tenere attivo un filo diretto con la propria barca. Questo si concretizza realizzando un prodotto base personalizzabile a seconda dei desideri dell'armatore che potrà scegliere il prodotto finale da ordinare aggiungendo altri moduli che gli permetteranno di ottenere le funzioni che desidera.

L'area che Aitech s.r.l. ritiene di poter servire è per il momento circoscritta all'Italia, per estendersi in fasi successive anche alle zone costiere delle nazioni bagnate dall'Adriatico e dal Tirreno, secondo una logica di estensione progressiva del mercato.

Il cliente tipo è il possessore di una unità da diporto di lunghezza tra i 10 e i 18 metri, nel caso di unità a vela, con limite inferiore estendibile agli 8 metri nel caso di unità a motore. In ogni caso si concentrerà lo studio sui cabinati ormeggiati nelle strutture italiane.

Basandosi sul rapporto UCINA 2014 (Confindustria Nautica), su un totale di 409 marine e porticcioli, in Italia sono disponibili 156.606 posti barca: di questi il 10% deve essere per normativa lasciato libero per i transiti, quindi si può considerare un totale di 140.945 posti barca disponibili che vengono ridotti di un ulteriore 10% relativo ai posti barca non venduti (per effetto della crisi/migrazione) ottenendo un totale di barche ormeggiate pari a 126.850.

Di tutte le barche ormeggiate si può considerare che una parte sia del tipo cabinato e le altre siano dei day cruiser che quindi in linea di massima non necessitano della strumento proposto. Nel complesso, si ritiene di non sbagliare di molto se si considera come mercato potenziale di riferimento le 104.000 unità immatricolate (per unità di lunghezza maggiore di 10 m è obbligatoria l'immatricolazione) che corrisponde al 66.4% dei posti barca disponibili in Italia. Si può riassumere quanto sopra osservando la Tabella 1:

N. posti barca in Italia		156.606
Posti liberi transito	10%	15.661
Posti occupati	90%	126.851
Mercato Potenziale	82%	104.018

Tabella 1: Mercato potenziale

2 eKEEPER2, applicazione per Smartphone Android

eKEEPER2 fornisce un'interfaccia semplice, intuitiva ed ordinata per facilitare le operazioni sull'imbarcazione da parte del cliente: tramite il proprio smartphone, dispositivo largamente diffuso, ogni possessore di eKEEPER2 potrà inviare e ricevere informazioni circa lo stato e la posizione dell'imbarcazione, verificarne costantemente la situazione di sicurezza e le condizioni meteo presenti.

Tali informazioni vengono scambiate prevalentemente attraverso la rete internet, su rete 2G per quanto riguarda eKEEPER2, mentre per quanto riguarda lo smartphone dell'utente una connessione internet è sufficiente.

Nel caso in cui sia impossibile accedere alla rete internet mobile alcune funzionalità non saranno più disponibili, mentre altre operazioni potranno ancora essere effettuate attraverso l'utilizzo di messaggi SMS. Il criterio secondo il quale si è scelto di disattivare o meno alcune funzionalità riguarda principalmente la quantità di dati da trasmettere la quale, se ingente, difficilmente è trasferibile via SMS senza incorrere in pagamenti eccessivi per via delle tariffe dell'operatore. Le scelte effettuate verranno presentate e giustificate man mano che verrà presentata l'applicazione nelle sezioni successive.

Per quanto concerne le operazioni via internet si è deciso di introdurre un server per il mantenimento di alcune informazioni chiave (si veda sezione 3): oltre a fornire una funzione di backup, il server si è rivelato un'ottima soluzione per lo scambio rapido di informazioni tra eKEEPER2 ed i dispositivi mobili.

Lo schema di comunicazione tra dispositivo a bordo dell'imbarcazione e cellulare dell'utente è riassunto nello schema in Fig. 1:

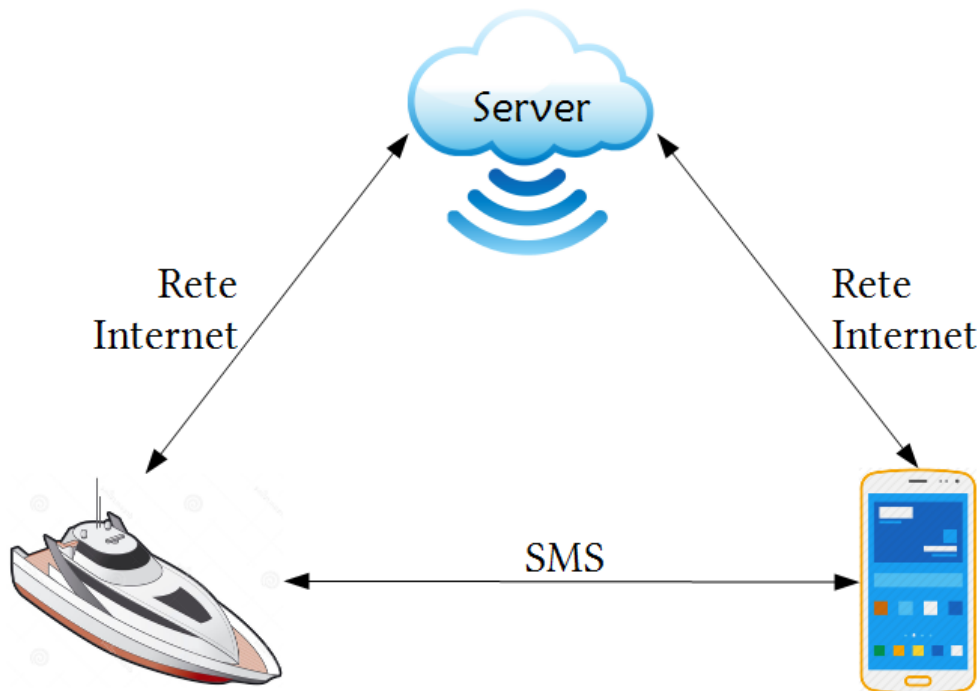


Figura 1: Schema connessione tra smartphone ed eKEEPER2

2.1 Breve descrizione dello strumento eKEEPER2

Lo strumento collocato a bordo dell'imbarcazione è costituito da tre blocchi fondamentali:

- Microprocessore: per la prototipazione in tempi brevi e per la praticità di programmazione è stata utilizzata la scheda Arduino Due (Fig. 2), la quale dispone di una discreta capacità

di calcolo unita ad una grande quantità di input ed output digitali ed analogici. Essa dispone inoltre di periferiche di interfacciamento quali seriale, *SPI* ed *I²C*, consentendo l'utilizzo di moduli esterni che comunicano secondo tali protocolli

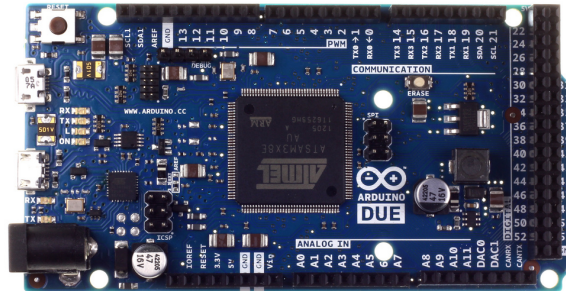


Figura 2: Scheda Arduino Due

- Elettronica di interfacciamento: per collegare gli ingressi ed uscite di Arduino Due al campo (ossia l'imbarcazione) è necessario progettare un apposito circuito elettrico. Questo consente di misurare tensioni, azionare contatti o ricevere e memorizzare dati assicurandosi che la scheda Arduino Due non venga danneggiata da sovratensioni o sovracorrenti. Il prototipo di tale scheda è riportato in Fig. 3:

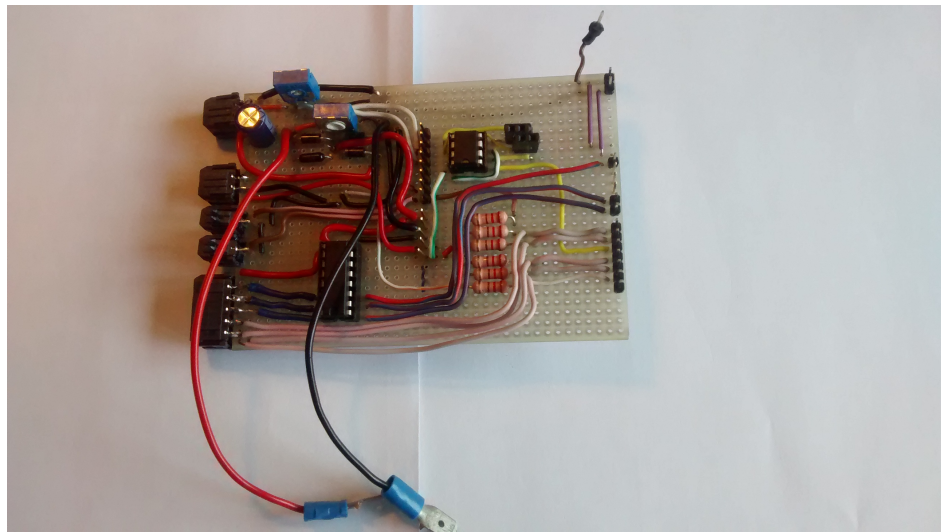


Figura 3: Scheda di interfacciamento elettrico

- Scheda di comunicazione: la comunicazione descritta in Fig. 1 è resa possibile da un modulo GSM/GPRS/GPS (Fig. 4) compatibile con Arduino Due. Lo scambio dati con Arduino, sia in ingresso che in uscita, avviene tramite interfaccia seriale rendendo possibile il collegamento di Arduino Due alla rete cellulare, dove può scambiare SMS con l'applicazione oppure collegarsi al server per leggere o scrivere informazioni.



Figura 4: Scheda GSM/GPRS/GPS

La descrizione completa della progettazione ed implementazione dello strumento è riportata in [16].

2.2 Struttura di un'applicazione Android

Nelle seguenti sezioni viene riportata una breve panoramica dei principali contenuti di Android per facilitare la comprensione da parte del lettore delle sezioni dedicate all'applicazione sviluppata.

I contenuti completi e dettagliati sono consultabili in [1]. All'interno di una applicazione Android è possibile identificare quattro componenti principali: ciascun componente è un diverso punto attraverso cui il sistema può interagire con l'applicazione. Non tutti questi punti sono anche punti di accesso reali per l'utente, ma ciascuno di essi possiede una propria entità e svolge un determinato ruolo, contribuendo a definire il comportamento generale dell'applicazione. Tali componenti sono:

- **Activity:** è l'elemento principale da cui iniziare per costruire la maggior parte delle applicazioni. La maggior parte delle applicazioni risulta formata da più Activity, le quali sono indipendenti tra loro. L'utilizzo di diverse Activity permette di creare porzioni di codice che svolgono determinati compiti e che verranno invocate solo quando richiesto. Nella sezione successiva verrà approfondito maggiormente l'argomento legato alle Activity.
- **Services:** componenti privi di interfaccia grafica (posseduta invece da un'Activity), solitamente eseguiti in background. Un esempio del loro utilizzo può essere la connessione ad internet effettuata in background mentre l'utente interagisce con altre finestre dell'applicazione.
- **Broadcast Receiver:** è un componente che non svolge alcuna operazione prestabilita, tuttavia rimane in ascolto e reagisce conseguentemente ad un evento. Nel contesto dell'applicazione per eKEEPER2, il Broadcast Receiver svolge le funzioni di ricezione e gestione di SMS. Gli annunci sono trasmessi dal sistema operativo per notificare degli eventi alle applicazioni.
- **Content Provider:** rende disponibili un insieme di dati di una certa applicazione alle altre applicazioni (ad esempio i contatti della rubrica). Solitamente i dati che si vogliono rendere accessibili ad altre applicazioni vengono salvati all'interno di un database SQLite.

Come si può notare dalla loro descrizione, Services, Broadcast Receiver e Content Provider sono componenti adatti a particolari tipi di operazioni, rispettivamente per svolgere servizi in

background, per reagire a particolari eventi notificati dal sistema operativo o per condividere dati. L'elemento che si trova alla base della App è l'Activity. Se si rende necessario infatti visualizzare delle informazioni, configurare delle impostazioni o richiedere dei dati all'utente tramite interfaccia grafica, non si può prescindere dall'utilizzare questo componente.

2.2.1 Activity

Vengono ora presentate le caratteristiche principali di un'Activity (la trattazione completa è consultabile in [2],[3] e[4]). Un'Activity è definita come una finestra contenente l'interfaccia grafica dell'applicazione che ha lo scopo di gestirne l'interazione con l'utente [5]. La singola applicazione può essere composta da un qualsiasi numero di Activity, questo allo scopo di trattare diverse parti del software: ognuna svolge una particolare funzione e deve essere in grado di salvare il proprio stato in modo da poterlo ristabilire successivamente come parte del ciclo di vita dell'applicazione (si veda sezione 2.2.2).

Tipicamente, all'inizio di un progetto, viene specificata l'Activity principale, cioè quella che viene presentata all'utente quando l'applicazione viene avviata per la prima volta. Da ogni Activity è possibile avviarne altre per svolgere diverse funzioni ma, ogni volta che una nuova Activity viene eseguita, quella precedente viene arrestata ed il sistema la conserva in una pila detta back stack.

L'Activity in esecuzione è detta in foreground, mentre quelle che sono in pausa e conservate nello stack si dicono in background e potrebbero essere terminate dal sistema operativo qualora la memoria RAM fosse insufficiente. La back stack funziona con il principio delle code LIFO, quindi qualora l'utente termini l'Activity corrente e prema il pulsante indietro, quella precedente viene riattivata. Questa regola può essere alterata predisponendo apposite istruzioni che modificano il comportamento LIFO del back stack.

2.2.2 Ciclo vita di un'applicazione

Dal momento in cui un'applicazione viene lanciata, fino al momento in cui essa viene messa in background, l'Activity passa attraverso varie fasi che costituiscono il suo ciclo di vita. Per assicurare un corretto funzionamento dell'App, capire quali sono le fasi di cui essa è composta è di fondamentale importanza. Allo scopo di poter definire un'Activity, la classe creata dovrà estendere la classe superiore Activity, che fornisce una serie di metodi per gestire tutti gli eventi che ne governano il ciclo di vita. La Fig. 5 illustra i metodi fondamentali per la definizione di un'Activity e ne mostra gli eventi associati:

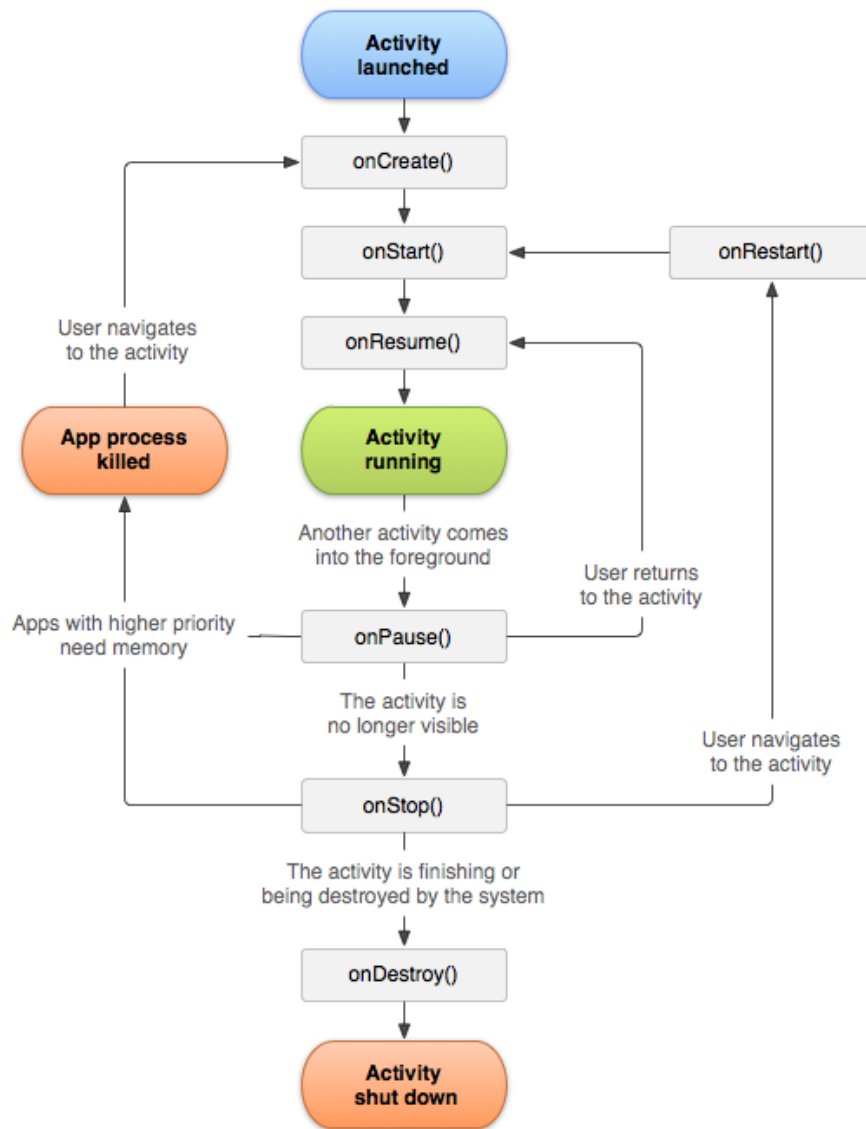


Figura 5: Ciclo di vita di un'Activity

Il primo metodo che viene eseguito quando l'Activity viene invocata è `onCreate()`, che ha lo scopo di definire gli elementi di base per il funzionamento. Al suo interno si richiama il metodo `setContentView()`, il quale carica uno specifico layout che può essere personalizzato a piacere. Una volta che l'esecuzione di `onCreate()` viene portata a termine, viene richiamato il metodo `onStart()`, il quale corrisponde al momento in cui l'Activity diventa visibile all'utente. Qualora l'utente inizia l'interazione con l'applicazione verrà richiamato il metodo `onResume()`. Da questo momento in poi verrà eseguito normalmente il codice contenuto all'interno dell'Activity, il quale costituisce il funzionamento della stessa, fino a quando non sopraggiungerà un altro evento come, ad esempio, la chiamata ad un'altra Activity. In questo caso verrà invocato il metodo `onPause()`, dove solitamente si salvano le modifiche ai dati, e contemporaneamente il contesto dell'Activity verrà salvato nel back stack. Se essa poi dovesse venire richiamata, riprenderà la sua esecuzione dal metodo `onResume()`, mentre se diventerà non più visibile all'utente verrà invocato il metodo `onStop()` il quale, in caso di mancanza di memoria, provvede ad eliminare le Activity in sospeso non necessarie. Nel caso in cui l'applicazione venga riaperta dopo l'esecuzione del metodo `onStop()`, questa dovrà ricominciare la sua esecuzione, tramite il metodo `onRestart()`, dal metodo `onStart()`. Infine è presente il metodo `onDestroy()`, che permette di specificare quali operazioni devono essere effettuate prima che l'App venga terminata.

(dall'utente o per mancanza di memoria).

2.2.3 Struttura di un'applicazione

Dopo aver analizzato brevemente la teoria relativa alla costruzione e gestione nel tempo di un'applicazione Android, oltre ai componenti principali che la costituiscono, vengono ora presentati i file per lo sviluppo del codice.

Al momento della creazione del nuovo progetto viene generata una struttura a cartelle e sottocartelle, che si mantiene man mano che l'applicazione si espande. Fra le cartelle ed i file riportati in Fig. 6 vanno menzionati:

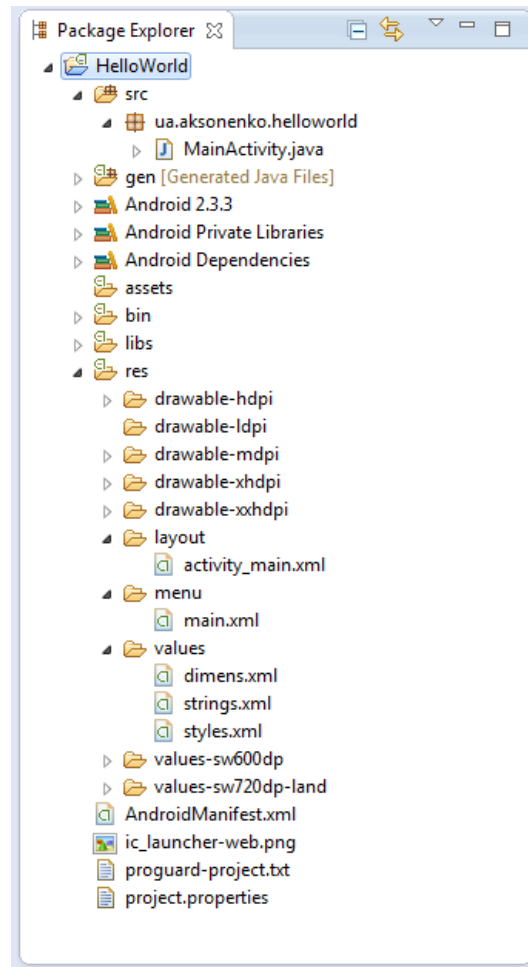


Figura 6: Struttura di un'applicazione Android

1. La cartella *src*, che contiene il codice sorgente dell'applicazione, il codice per l'interfaccia grafica, le classi etc
2. La cartella *res*, che contiene tutti i file e le risorse necessarie del progetto: immagini, loghi, video, layout xml, stringhe di testo multilingua. Può inoltre essere strutturata in sottocartelle per una migliore organizzazione delle risorse, come ad esempio *drawable* per raccogliere gli oggetti disegnabili (come i loghi), *layout* delle diverse finestre dell'applicazione, stringhe raggruppate per lingua
3. La cartella *gen*, con all'interno una serie di elementi di codice generati automaticamente al momento della compilazione del progetto, necessari al controllo delle risorse dell'applicazione

4. La cartella *assets*, che contiene tutti gli altri file ausiliari necessari all'applicazione, ad esempio file di configurazione o di dati
5. Un file *AndroidManifest.xml*, situato nella cartella principale, nel quale vengono descritti il comportamento e la configurazione dell'applicazione, le risorse di cui richiede l'utilizzo (accesso ad internet, accesso alla memoria interna, ...) e la sua identificazione (nome, versione, logo)

2.2.4 AndroidManifest

Il file *AndroidManifest.xml* è di fondamentale importanza dal momento che in sua assenza o in caso di una sua errata compilazione può compromettere l'utilizzo dell'intera applicazione. Esso viene generato automaticamente dal plugin ADT una volta creato il progetto ed è memorizzato nella directory principale. Si distingue dagli altri file XML per la presenza nell'istestazione del tag `< manifest >`, che serve per includere i nodi che definiscono i componenti dell'applicazione, l'ambiente di sicurezza e tutto ciò che fa parte dell'applicazione.

Listing 1: Dichiarazione del Manifest

```
1 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
2     package="com.example.eKEEPER2"
3     android:versionCode="1"
4     android:versionName="1.0" >
```

Nella sua interezza contiene tutte le informazioni specificate durante la creazione del progetto, l'icona, la versione dell'applicazione e tutte le Activity e i Services che compaiono al suo interno. Per dichiarare un'Activity si utilizza la seguente porzione di codice:

Listing 2: Dichiarazione di un'Activity

```
1 <activity
2     android:name=".MainActivity"
3     android:screenOrientation="portrait"
4     android:label="@string/title_activity_main" >
5 </activity>
```

dove alla riga 2 viene specificato il nome dell'Activity, alla riga 3 si imposta la visualizzazione verticale forzata ed infine alla riga 4 si specifica il nome del titolo dell'Activity, richiamata dal file *string* il quale contiene le stringhe multilingua.

Un'altra funzione importante dell'AndroidManifest è la dichiarazione dei permessi di cui necessita l'applicazione. Di default infatti, l'accesso a particolari funzioni del dispositivo quali l'accesso ad internet, l'accesso alla memoria in lettura o scrittura, la lettura dello stato del sistema operativo per la ricezione di messaggi o chiamate, non sarebbe permesso. Pertanto queste risorse per poter essere utilizzate devono essere dichiarate proprio all'interno di questo file.

In questo modo il gestore dei pacchetti è a conoscenza dei privilegi necessari all'applicazione, e l'utente è sempre avvisato prima dell'installazione su quali risorse l'applicazione potrà andare ad accedere.

All'interno dell'applicazione Android per eKEEPER2 sono concessi i permessi riportati in Listing 3:

Listing 3: Dichiarazione dei permessi per l'applicazione

```
1 <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

```

2 <uses-permission android:name="android.permission.INTERNET" />
3 <uses-permission
    android:name="com.google.android.providers.gsf.permission.READ_GSERVICES" />
4 <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
5 <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
6 <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
7 <uses-permission android:name="android.permission.READ_CONTACTS" />
8 <uses-permission android:name="android.permission.READ_PHONE_STATE" />
9 <uses-permission android:name="android.permission.SEND_SMS" />
10 <uses-permission android:name="android.permission.RECEIVE_SMS" />
11 <uses-permission android:name="android.permission.READ_SMS" />
12 <uses-permission android:name="android.permission.WRITE_SMS" />
13 <uses-permission android:name="android.permission.RECEIVE_MMS" />
14 <uses-permission android:name="android.permission.WRITE" />
15 <uses-permission android:name="android.permission.VIBRATE" />

```

In questo modo è stato dichiarato nel Manifest che l'applicazione potrà conoscere le informazioni sullo stato della rete, accedere ad internet, usufruire dei servizi Google, scrivere la memoria, accedere alla posizione del cellulare e ricevere ed inviare SMS.

2.2.5 La cartella *res*

All'interno della cartella *res* Android presenta un'organizzazione lineare per la gestione delle risorse. Questo significa che non sono consentite sottocartelle oltre il secondo livello di gerarchia. Le funzioni delle diverse directory presenti di default all'interno di *res* sono le seguenti:

- */layout*: *layout* contiene i file XML che definiscono i vari componenti della GUI (Graphic User Interface) utilizzati dall'applicazione. Essi possono essere sia definizioni di layout completi (LinearLayout, RelativeLayout, ...), che di soli controlli grafici (Button, EditText, ListView, ...).
Gli ambienti di sviluppo per applicazioni permettono di inserire i vari elementi dell'interfaccia grafica attraverso appositi tool tramite i quali è possibile creare un layout trascinando gli elementi nelle posizioni desiderate; allo stesso tempo un compilatore genera automaticamente porzioni di codice relative a questi stessi oggetti. Tuttavia essi possono anche essere definiti direttamente da codice senza l'utilizzo delle risorse XML.
- */drawable*: Le tre cartelle *drawable* – *, rispettivamente *hdpi* per schermi ad alta densità di pixel, *mdpi* per quelli a media densità e *ldpi* per quelli a bassa densità, sono indicate per contenere le risorse grafiche (icone, sfondi, ...), utilizzate nell'applicazione, in diversi formati come GIF, PNG o JPEG. In questo modo il sistema potrà scegliere in automatico quelle più adatte alle differenti risoluzioni e densità dei display, prelevandole dalla cartella apposita. Recentemente, per ampliare il supporto alle nuove risoluzioni HD e FullHD e ai nuovi dispositivi di più ampie dimensioni come i tablet, è stata introdotta una nuova directory *xhdpi* per supportare i display ad altissima densità di pixel.
- */menu*: La cartella *menu*, come intuibile dal nome stesso, contiene i file XML che definiscono la struttura dei vari menù (ad esempio quello delle opzioni).
- */values*: L'ultima cartella, *values*, contiene file XML che definiscono risorse di diversa natura, per esempio stili, stringhe utilizzate dall'applicazione, colori e tante altre tipologie di dati. Di particolare importanza è il file *strings.xml*, il quale viene descritto in maggior dettaglio nella sezione 2.2.6.

2.2.6 *strings.xml* e multilingua

Il file *strings.xml* è di fondamentale importanza, sia per chiarezza nel codice, sia per facilità di implementazione del multilingua. Si tratta ovviamente di una risorsa di tipo XML, la cui funzione è quella di raccogliere tutte le stringhe che si intendono visualizzare nell'applicazione. In questo file vengono associate a delle variabili di tipo stringa le relative parole o frasi che compariranno nelle schermate dell'applicazione: è sufficiente richiamare da *strings.xml* l'apposita variabile ed essa richiama la stringa a cui è collegata. In Listing 4 è riportato un esempio che ne chiarisce l'utilizzo:

Listing 4: File *string.xml*

```
1 <?xml version="1.0" encoding="utf-8"?>
2   <resources>
3     <string name="hello">Hello World, ExampleActivity!</string>
4     <string name="app_name">Example</string>
5 </resources>
```

In Listing 5 è mostrato come richiamare nell'applicazione una determinata stringa:

Listing 5: Richiamare una stringa da *string.xml*

```
1 String helloWorld = R.string.hello;
```

Oltre ad essere un comodo ed ordinato raccoglitore di stringhe da utilizzare all'interno del codice, la presenza del file *strings.xml* risulta essere estremamente utile per la traduzione dell'applicazione in più lingue. eKEEPER2, infatti, non è rivolto al mercato di soli utenti italiani, bensì considera la possibilità in cui anche un armatore non italiano possa desiderare tale strumento. L'applicazione, dunque, prevede la presenza almeno della lingua inglese oltre a quella italiana.

Per ottenere il multilingua si parte dalla cartella *values*, la quale identifica la lingua di default dell'applicazione, in questo caso l'inglese. Per aggiungere altre lingue è sufficiente creare delle ulteriori cartelle con nome *values – xx*, dove *xx* è il suffisso della nuova lingua.

In fase di configurazione dell'applicazione il cliente potrà selezionare la lingua che preferisce. Le cartelle presenti in *res* sono quindi *values* per l'inglese e *values – it* per l'italiano: dentro ad entrambe le cartelle è presente il relativo file *string.xml*. In Fig. 7 sono mostrate la struttura della cartella *res* e le due cartelle sopra elencate contenenti i file *strings.xml* tradotti nelle due lingue.

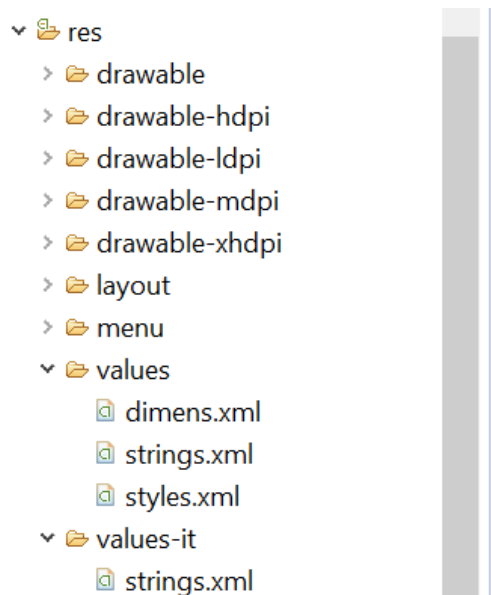


Figura 7: Struttura della cartella *res*

2.3 Programmazione dell'applicazione Android

2.3.1 Java ed XML

Inizialmente è necessario inquadrare bene il ruolo dei due linguaggi all'interno della programmazione di applicazioni [6].

- Java: da quando Google nel 2005 ha preso possesso di Android ha avuto un ruolo fondamentale nello sviluppo di App per questo sistema operativo. Sebbene quest'ultimo non supporti la JVM (Java Virtual Machine) originale, ma ne possieda una versione proprietaria modificata, la DVM (Dalvick Virtual Machine), a livello di sviluppo del codice non si notano particolari differenze dato che sono presenti la maggior parte delle librerie. La principale differenza col classico sviluppo di codice Java risiede però nel fatto che non è presente un entry point (il classico metodo main) da dove normalmente un programma comincia a caricare le sue parti software e avviarsi: tutto è pensato per essere un componente pilotato dagli eventi (Event Driven).

Il principale vantaggio consiste nell'ottimizzazione delle risorse da parte del sistema operativo, ad esempio rinunciando a caricare risorse e hardware non supportati o non prioritari perchè inutilizzati.

Il ruolo di Java, in questo contesto, è quello dello sviluppo delle parti dinamiche del programma come, ad esempio, la gestione degli eventi in seguito agli input dell'utente e al suo interagire con l'interfaccia grafica, la modifica in real time del layout e così via. [7]

- XML: eXtensible Markup Language è un linguaggio di markup o linguaggio marcatore, basato su un meccanismo sintattico che consente di definire e controllare il significato degli elementi contenuti in un documento o in un testo. Il suo ruolo nella programmazione Android è complementare a quello di Java. Mentre quest'ultimo viene utilizzato per le parti dinamiche, XML descrive le parti statiche. Esse possono essere quelle caratteristiche che non cambiano durante l'esecuzione dell'applicazione, come per esempio il colore dello sfondo della schermata, la posizione dei bottoni, delle caselle di testo, ecc. [8]

2.3.2 La classe *R*

Finora sono state introdotte da una parte le risorse XML che definiscono le parti statiche dell'applicazione, come la GUI, i parametri, gli stili (contenute nella cartella *res*), e dall'altra il codice Java che permette di gestire eventi e le interazioni con l'utente (nella cartella *src*). Il ponte fra questi due mondi viene gestito attraverso la classe *R*, contenuta nella cartella *gen* e generata in modo dinamico ad ogni compilazione del codice. Essa permette appunto di identificare in modo dinamico tutte le risorse (da stringhe a layout, fino agli stili) dichiarate via XML, per poterle modificare ed utilizzare all'interno di classi Java [9]. Ad esempio per importare il layout di un componente grafico per la visualizzazione di un testo come una *TextView* è possibile utilizzare l'istruzione riportata in Listing 6:

Listing 6: Inizializzazione di un oggetto *TextView*

```
1 TextView text = (TextView)findViewById(R.id.text);
```

in cui *text* è l'ID assegnato ad una *TextView* dichiarata all'interno di un file XML nella cartella *res/layout* alla riga 7 del Listing 7:

Listing 7: Dichiarazione di *TextView* in *layout.xml*

```
1 <? xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3   android:layout_width="fill_parent"
4   android:layout_height="fill_parent"
5   android:orientation="vertical">
6
7   <TextView android:id="@+id/text"
8     android:layout_width="fill_parent"
9     android:layout_height="wrap_content"/>
```

2.4 Creazione di un'Activity

In questa sezione viene mostrato come creare un'Activity generica e come associare ad essa un layout. Per fare ciò è necessario creare una nuova classe che dovrà essere sottoclasse di Activity (o di una sua sottoclasse esistente). Qui è necessario implementare metodi di callback che il sistema chiama quando avvengono le transizioni tra i vari stati del suo ciclo di vita (si veda sezione 2.2.2). La struttura essenziale di un'Activity è mostrata in Listing 8:

Listing 8: Struttura di un'Activity

```
1 public class ExampleActivity extends Activity {
2   @Override
3   public void onCreate(Bundle savedInstanceState)
4   {
5     //L'Activity e' stata creata
6     super.onCreate(savedInstanceState);
7   }
8
9   @Override
10  protected void onResume()
11  {
12    //L'Activity e' diventata visibile
13    super.onResume();
14  }
```

```

15
16 @Override
17 protected void onPause()
18 {
19     //L'Activity sta per essere distrutta
20     super.onPause();
21 }
22 }

```

Una volta creata la struttura dell'Activity, allo scopo di voler associare ad essa un layout chiamato *main.xml*, situato all'interno di *res/layout*, si invoca il metodo *setContentView()* della classe Activity. Tale operazione è consigliabile utilizzarla all'interno di *onCreate()*. L'istruzione corrispondente è illustrata in Listing 9:

Listing 9: Caricamento del layout

```

1 setContentView(R.layout.main);

```

2.5 Grafica di base

In questa sezione vengono brevemente introdotti e descritti i diversi oggetti che verranno impiegati per comporre l'applicazione, al fine di rendere più semplice la descrizione di quest'ultima. Il materiale è stato tratto da [4], [10] e [11].

2.5.1 View e Layout

Una volta creata l'Activity, ad essa non è ancora associato un layout che ne caratterizza gli elementi presenti, le dimensioni ed altre caratteristiche.

View è la classe di base per la definizione dell'interfaccia grafica e la gestione degli eventi nati dall'interazione dell'utente con la UI. L'interfaccia grafica di un'Activity viene in genere definita da una o più *View*, organizzate in una struttura ad albero e rappresentate sullo schermo tramite il *Layout*. Questi gruppi di *View* vengono chiamati *ViewGroup*. Una *View* può quindi essere considerata come una porzione dell'interfaccia dell'Activity e spesso consiste in controlli grafici come *Button*, *TextView*, *EditText*, ecc., che sono delle sottoclassi con particolari funzionalità che verranno analizzate in seguito.

Le *View* sono di quattro tipi:

- **LinearLayout**: Dispone ogni elemento contenuto al suo interno lungo una riga se l'attributo *orientation* è impostato su *vertical* e su colonna se l'attributo è impostato su *horizontal*.
In pratica il layout viene considerato come un campo con tanti rettangoli ordinati per riga oppure per colonna in cui disporre i vari componenti.
- **TableLayout**: ci permette di disporre gli oggetti *View* usando una griglia composta da righe e colonne, come è tipico di una tabella.
- **RelativeLayout**: è il layout più flessibile tra quelli nativi perchè permette di definire le posizioni di ogni oggetto *View* in relazione agli altri oggetti presenti.
Il primo viene posizionato in alto a sinistra, successivamente per ogni nuovo elemento aggiunto deve essere definita la posizione relativa a questo.
- **FrameLayout**: è la tipologia più semplice di layout, perchè posiziona in ordine di inserimento tutti gli oggetti *View* partendo dall'angolo in alto a sinistra dello schermo.

In base alla situazione, alla necessità di rendere funzionale il layout e al tipo di oggetto da posizionare, occorre effettuare un'accurata scelta del miglior tipo di layout. L'obiettivo ultimo è quello di adattare bene allo schermo di ogni dispositivo le informazioni, aumentando la possibilità che gli utenti apprezzino il lavoro di progettazione.

2.5.2 TextView

La *TextView* rappresenta il classico componente etichetta, la cui funzione è quella di visualizzare del testo sul display.

Con l'esempio di codice riportato in Listing 10 si mostra l'utilizzo di tale oggetto:

Listing 10: Dichiarazione di un *TextView* in *layout.xml*

```
1 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2   xmlns:tools="http://schemas.android.com/tools"
3   android:layout_width="match_parent"
4   android:layout_height="match_parent"
5   android:orientation="vertical"
6   android:gravity="top" >
7
8   <TextView
9       android:id="@+id/textTitle"
10      android:layout_width="wrap_content"
11      android:layout_height="wrap_content"
12      android:layout_gravity="center_horizontal"
13      android:layout_marginTop="20dp"
14      android:text="@string/welcome"
15      android:textAppearance="?android:attr/textAppearanceLarge" />
```

Alla riga 8 viene dichiarato l'oggetto *TextView* a cui vengono assegnati i seguenti attributi:

- riga 9: l'ID, assegnato alla variabile *textTitle*
- riga 10 – 13: attributi che riguardano la grandezza del *TextView*, il suo allineamento e dimensione
- riga 14 – 15: il testo richiamato dal file *strings.xml* con relativo attributo dimensionale

Il codice incorporato all'interno dell'Activity è riportato in Listing 11:

Listing 11: Inserimento di *TextView* in Activity

```
1 @Override
2 public void onCreate(Bundle savedInstanceState)
3 {
4     super.onCreate(savedInstanceState);
5     setContentView(R.layout.main);
6
7     //Richiamo il TextView titolo
8     TextView titolo = (TextView)findViewById(R.id.textTitle);
9 }
```

Alla riga 5 viene richiamato l'apposito layout contenente le istruzioni della schermata che si vuole creare, alla riga 8 si inizializza l'oggetto *TextView* associandolo a quello contenuto nel file XML.

Infine, in Fig. 8, si mostra il risultato ottenuto su dispositivo Android:

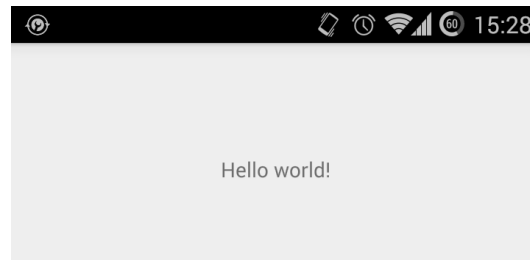


Figura 8: Esempio di TextView

2.5.3 EditText

In molte applicazioni spesso vi è la necessità di inserire del testo, ad esempio per inserire delle credenziali o altre informazioni. Per fare questo è presente l'oggetto *EditText*, il quale funziona come campo di testo. Tramite questo componente grafico è possibile inserire del testo e gestire gli eventi relativi al suo contenuto. Si riporta in Listing 12 un esempio:

Listing 12: Dichiarazione di EditText in *layout.xml*

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res /android"
3   android:layout_width="fill_parent"
4   android:layout_height="fill_parent"
5   android:orientation="vertical">
6
7   <EditText android:id="@+id/edit1"
8     android:layout_width="fill_parent"
9     android:layout_height="wrap_content"
10    android:hint="Insert Text Here"/>

```

La proprietà *hint* è il messaggio iniziale presente nella casella di testo, solitamente una parola che suggerisce quale dato deve essere inserito in un certo campo di testo. L'*hint* viene visualizzato in grigio chiaro e scompare quando si comincia a scrivere il contenuto. E' possibile ricavare il testo inserito nell'*EditText* tramite il metodo *getText()*, il quale restituisce la sequenza di caratteri contenuta nel campo, come mostrato in Listing 13

Listing 13: Inserimento di EditText in Activity

```

1 @Override
2 public void onCreate(Bundle savedInstanceState)
3 {
4     super.onCreate(savedInstanceState);
5     setContentView(R.layout.main);
6
7     TextView text = (TextView)findViewById(R.id.testo);
8     EditText edit1 = (EditText)findViewById(R.id.edit1);
9
10    String contenutoText = text.getText();
11 }

```

Il risultato dell'inserimento di un campo *editText* è mostrato in Fig. 9, dove è stato inserito un campo di testo corredato di opportuno *hint*:

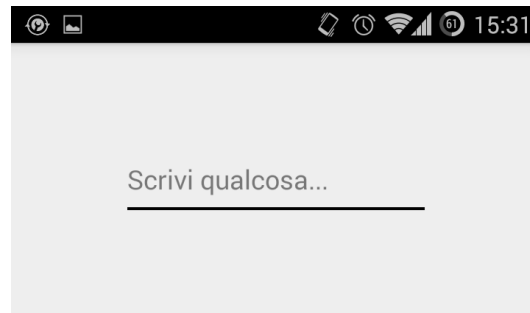


Figura 9: Esempio di EditText

2.5.4 Button

Il Button (pulsante) è un altro dei componenti grafici classici presente in molti programmi. Può essere costituito da testo o da un'icona (o da entrambi) al quale è associata l'azione che si verifica quando l'utente interagisce con esso tramite un tocco. ImageButton è la classe che permette di realizzare un bottone con un'immagine di sfondo ed è in comportamento e in struttura simile alla classe Button.

Nei segmenti di codice riportati in Listing 14 e 15 è descritto come includere un Button all'interno del programma e come gestire gli eventi ad esso associato:

Listing 14: Dichiarazione di Button in Layout

```

1 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2   xmlns:tools="http://schemas.android.com/tools"
3   android:layout_width="match_parent"
4   android:layout_height="match_parent"
5   android:orientation="vertical"
6   android:gravity="top">
7
8   <Button
9     android:id="@+id/button"
10    android:layout_width="fill_parent"
11    android:layout_height="wrap_content"
12    android:layout_marginTop="150dp"
13    android:text="@string/button"/>

```

Listing 15: Inserimento di Button in Activity

```

1 @Override
2 protected void onCreate(Bundle savedInstanceState)
3 {
4   super.onCreate(savedInstanceState);
5   setContentView(R.layout.activity_home);
6
7   (Button) button = (Button)findViewById(R.id.button);
8
9   //Gestione del tocco sul pulsante
10  button.setOnClickListener(new View.OnClickListener()
11  {
12    public void onClick(View v)
13    {
14      //Codice da eseguire in caso di tocco
15

```

```

16      //Notifica a schermo di avvenuto tocco
17      Toast.makeText(getApplicationContext(), "Pulsante premuto",
18          Toast.LENGTH_LONG).show();
19  }
20 }

```

Nel codice è stato gestito l'evento legato al click sul componente: questa operazione è stata gestita attraverso un *Listener* (gestore eventi), che permette di intercettare la pressione avvenuta del tasto (evento *onClick*), settando il bottone nello stato *OnClickListener* (in attesa di essere premuto).

In caso di avvenuta pressione l'applicazione mostra una notifica *Toast*, ossia un riquadro di avviso, con scritto Pulsante premuto. Il risultato è mostrato in Fig. 10.

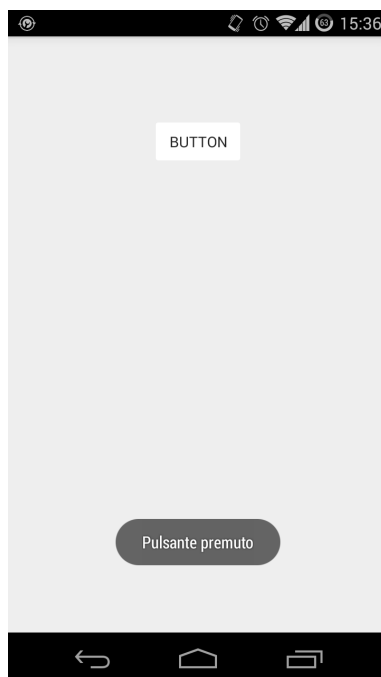


Figura 10: Esempio *onClick()* su pulsante

2.5.5 ListView

La *ListView* è un componente che permette di visualizzare, come dice il nome stesso, una lista di elementi. Le voci contenute vengono caricate in automatico da un *Adapter*, un oggetto che si occupa sia dell'accesso ai dati, sia della loro visualizzazione. Si riporta in Listing 16 un esempio:

Listing 16: Dichiarazione di *ListView* in Layout

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:tools="http://schemas.android.com/tools"
4     android:layout_width="fill_parent"
5     android:layout_height="fill_parent"
6     android:background="#ffffff"
7     android:orientation="vertical" >
8
9     <ListView

```

```

10     android:id="@android:id/list"
11     android:layout_width="match_parent"
12     android:layout_height="match_parent"
13     android:layout_above="@+id/button"
14     android:layout_below="@+id/titolo"
15     android:layout_marginTop="15dp" >
16 </ListView>

```

All'interno dell'Activity risulta più conveniente utilizzare il `ListAdapter`, componente pensato per adattarsi meglio a questo tipo di componente grafico. Più precisamente si occupa di associare ad ogni elemento della lista un layout predefinito, in questo caso *row_layout*, e di caricare nel componente grafico l'array di stringhe *cols* che è stato definito. Tale operazione viene svolta dal metodo *setListAdapter* (riga 9 del Listing 17).

Listing 17: Inserimento di `ListView` in Activity

```

1 @Override
2 public void onCreate(Bundle savedInstanceState)
3 {
4     super.onCreate(savedInstanceState);
5     setContentView(R.layout.main);
6     String[] cols = new String[] {"Elemento 1", "Elemento 2", "Elemento 3",
7                                     "Elemento 4"};
8
9     myAdapterAlert = new ArrayAdapter<String>(this, R.layout.row_layout, cols);
10    setListAdapter(myAdapterAlert);
11 }

```

2.6 Android API

In informatica con il termine Application Programming Interface API (Interfaccia di Programmazione di un'Applicazione) si indica un insieme di procedure disponibili allo sviluppatore, raggruppate per formare un set di strumenti, adatti all'espletamento di un determinato compito all'interno di un programma.

Spesso indica le librerie software messe a disposizione per un determinato linguaggio di programmazione, in questo specifico caso per il Sistema Operativo Android.

Lo scopo è quello di ottenere un'astrazione a più alto livello, di solito tra l'hardware e il programmatore o tra software a basso e quello ad alto livello, semplificando così il lavoro di sviluppo del codice. Le API permettono infatti di evitare ai programmatori di riscrivere ogni volta tutte le funzioni necessarie al programma dall'inizio (dal basso livello) e rientrano quindi nel più ampio concetto di riutilizzo del codice.

Il software che fornisce una certa API è detto implementazione dell'API. Nello specifico il Sistema Operativo Android mette a disposizione numerose API per far interagire l'applicazione con le funzionalità hardware dello smartphone e il software di sistema [14].

Le principali API permettono di:

- leggere/scrivere SMS, gestione delle chiamate in entrata e uscita
- modificare impostazioni di sistema
- leggere dati della rubrica, del calendario, dai social network, ecc.
- leggere scrivere sulla memoria interna o sulla memoria SD

- registrare file audio
- scattare fotografie tramite la fotocamera del dispositivo
- usare l'NFC, la geolocalizzazione GPS, il WiFi ed il Bluetooth

Vi sono inoltre una serie di API che sono state messe a disposizione dalle varie applicazioni che si sono sviluppate nel tempo per la piattaforma. Ad esempio quelle di Facebook e Google+ per la condivisione sui social network, quelle di Dropbox, Google Drive e Skydrive per il salvataggio dei dati in cloud, quelle di Gmail per l'invio di mail, ecc. Nei paragrafi successivi si porrà il focus dell'attenzione sull'utilizzo delle API per l'utilizzo delle mappe di Google Maps, inserite nell'applicazione per fornire all'utente la posizione corrente della sua imbarcazione, oltre alla possibilità di visualizzare i viaggi registrati da eKEEPER2.

2.6.1 Preparare il progetto

Le API di Google Maps non sono presenti di default negli editor per la programmazione di applicazioni, ma sono distribuite come parte della SDK Google Play services: per ottenerla, occorre aprire Android Sdk Manager e scaricare i seguenti pacchetti:

- Android Support Library
- Google Play services

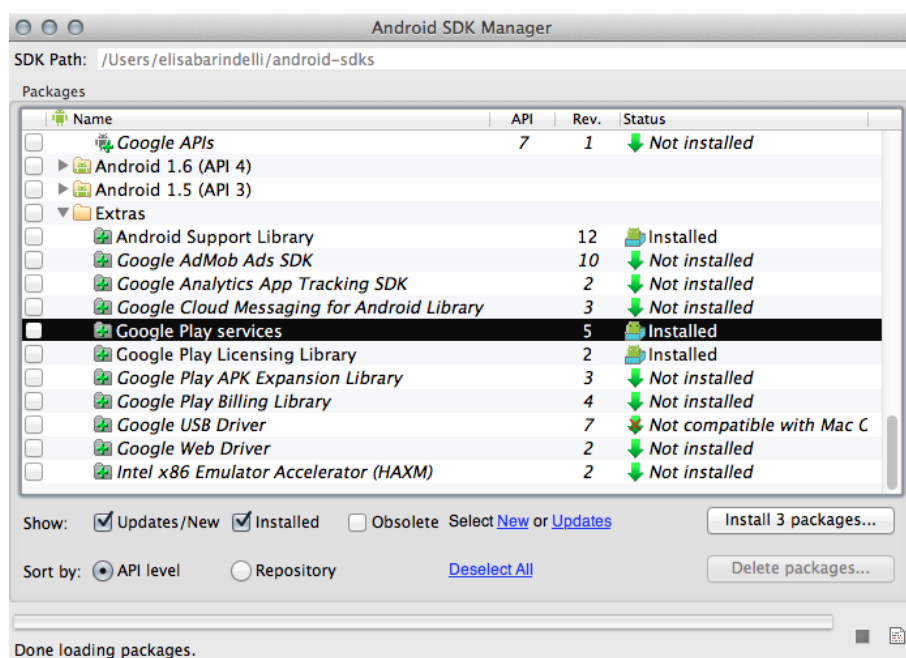


Figura 11: Aggiunta dei pacchetti necessari all'interno di Android SDK Manager

Dall'editor che si utilizza per la programmazione si crea un nuovo progetto al quale, per semplicità, si farà riferimento con il nome di *MapsApiExample*.

Si procede quindi alla creazione di un secondo progetto, che fungerà da libreria per il primo. Per farlo, occorre copiare la cartella *google – play – services _lib*, che è presente al momento dell'installazione dell'SDK, all'interno della cartella *workspace*, dove sono ospitati i progetti. La cartella si trova sotto il percorso:

¹ <android-sdk-folder>/extras/google/google_play_services/libproject/google-play-services_lib

Ora, affinché l'applicazione riceva l'effettivo permesso per l'utilizzo del codice di Google Maps, è necessario eseguire la seguente procedura:

1 **File > Import > Android > Existing Android Code into Workspace**

Nelle proprietà del progetto, si dichiara l'utilizzo delle librerie desiderate come mostrato in Fig. 12.

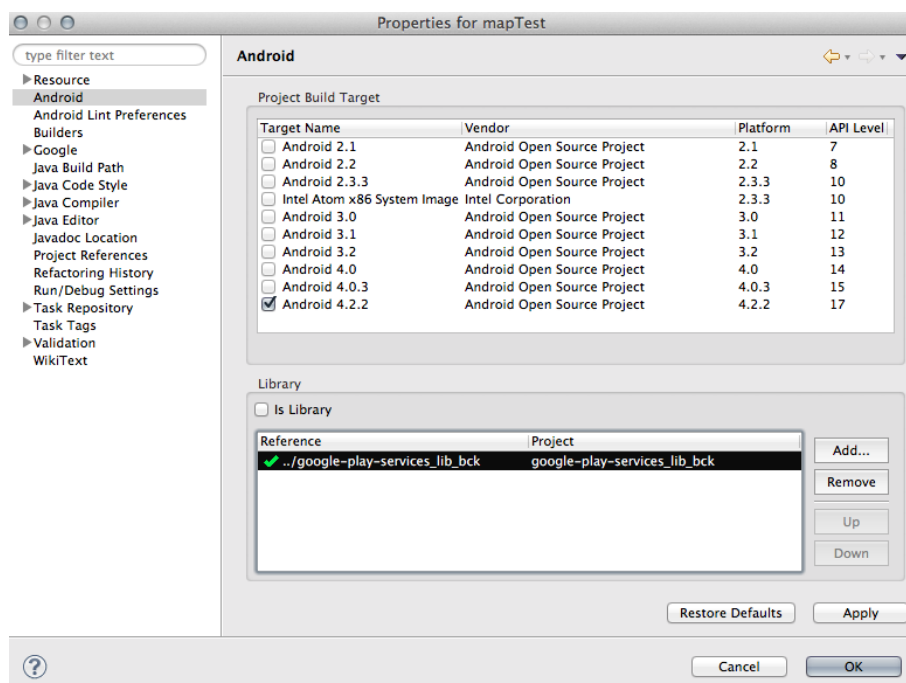


Figura 12: Aggiunta del codice proveniente da sorgente esterna

2.6.2 Ottenere una API KEY

Per poter fare qualsiasi richiesta alle API di Google Maps, è inoltre richiesta una API Key. Ad ogni *key* è assegnato un codice unico, chiamato *Fingerprint SHA – 1*, il quale è reperibile all'interno dell'editor di programmazione (Fig. 13):

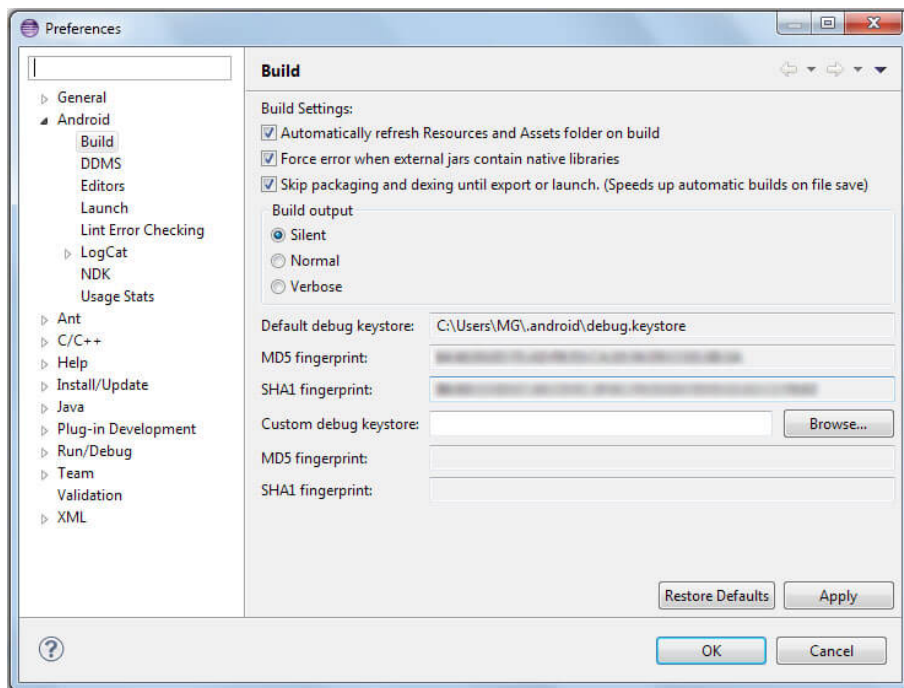


Figura 13: Reperimento del codice Fingerprint SHA-1

Il passaggio successivo avviene all'interno di una piattaforma online che Google dedica agli sviluppatori che richiedono le API: si apre la sezione Services e si abilita il servizio Google Maps Android API v2 (Fig. 14)

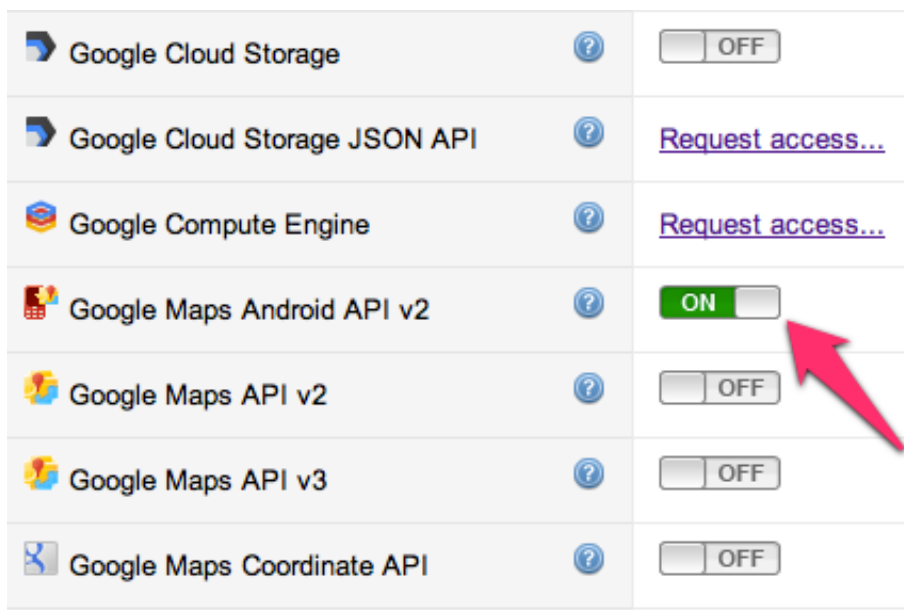


Figura 14: Abilitazione dell'API per Google Maps V2

Per quanto riguarda la procedura online, infine, si deve accedere alla sezione API Access e cliccare il pulsante Create new Android key. Nella finestra che si aprirà, si inserisce il codice SHA-1 ottenuto al punto precedente ed il nome del package della app in sviluppo, separati dal carattere “;”, ad esempio

¹ [Codice SHA-1];[com.example.helloworld]

A questo punto la API key sarà creata ed apparirà nell'elenco.

2.6.3 Modificare il Manifest.xml

Una volta ottenuta l'API key, serve dichiararla nell'applicazione ed impostare i permessi corretti perché possa essere utilizzata.

Questa operazione si effettua nel file *AndroidManifest.xml* dove sotto il tag `< application >` si aggiunge la dichiarazione

```
1 <meta-data android:name="com.google.android.maps.v2.API_KEY"
  android:value="api_key"/>
```

2.7 Gestione dei database SQLite in Android

La trattazione completa e dettagliata dei seguenti argomenti è presente in [12].

All'interno dell'applicazione è necessario poter mantenere in memoria un numero di informazioni che devono essere sempre disponibili ed ordinate per poterle presentare correttamente all'utente. I dati in questione sono di tipo numerico, stringhe oppure stringhe di numeri e riguardano rispettivamente lo stato dell'imbarcazione oltre ad i numeri dei dispositivi mobili del cliente, le credenziali di accesso all'applicazione ed infine le coordinate dei viaggi dell'applicazione.

Le informazioni appena citate sono quindi raccolte ordinatamente all'interno di tabelle generate da database SQLite: SQLite è considerato il motore di database più diffuso al mondo. Si tratta, in realtà, di una libreria software che permette di gestire in un unico file un database relazionale. Per avere un database SQLite nella propria App Android non è necessario installare componenti aggiuntivi, infatti la libreria SQLite risulta essere già inclusa nel sistema operativo e le API disponibili nel framework offrono tutto il supporto necessario. La documentazione necessaria per l'implementazione è stata presa da [12] e [13]

2.7.1 Implementare un database nell'applicazione

Quando si vuole implementare un database in un'applicazione è buona pratica creare una classe *Helper* e una classe *Adapter* in modo da semplificare le interazioni con esso. In questo modo si introduce un livello di astrazione che fornisce metodi intuitivi, flessibili e robusti per inserire, eliminare e modificare i record del database. Il compito della classe *Helper* è quello di memorizzare le costanti statiche come il nome della tabella e dei vari campi, mentre quello della classe *Adapter* consiste nel fornire *query* e metodi per la creazione del database e la gestione delle connessioni (apertura e chiusura) con esso.

La classe *Helper* dovrà estendere *android.database.sqlite.SQLiteOpenHelper* e contiene appunto il nome del database, la versione, il codice di creazione delle varie tabelle e la loro successiva modifica. Una classe di questo tipo deve essere creata appositamente per ciascun database necessario all'applicazione.

Il codice utilizzato è riportato in Listing 18:

Listing 18: Creazione del DatabaseHelper, 1/2

```
1 import android.content.Context;
2 import android.database.sqlite.SQLiteDatabase;
3 import android.database.sqlite.SQLiteOpenHelper;
4
5 public class DatabaseHelper extends SQLiteOpenHelper
6 {
7     //nome del database
8     private static final String DATABASE_NAME = "example.db";
9     //versione database
```

```

10 private static final int DATABASE_VERSION = 1;
11
12 //statement SQL di creazione del database
13 private static final String TABLE_CREATE = "create table example (_idviaggio
        integer primary key autoincrement, field1 text not null , field2 text not
        null);";
14
15 // Costruttore
16 public DatabaseHelper(Context context)
17 {
18     super(context, DATABASE_NAME, null, DATABASE_VERSION);
19 }
20
21 ...

```

In questa prima parte, nelle due costanti *DATABASE_NAME* e *DATABASE_VERSION* vengono indicati rispettivamente, il nome del database (example.db) e il numero di versione (1). Quest'ultimo è un intero che viene incrementato ogni volta che, in un nuovo rilascio del software, la base di dati viene modificata nella sua struttura. In questo modo quando l'utente aggiorna la propria applicazione, la classe riesce a capire se il database della versione precedentemente installata deve essere aggiornato oppure no. Sono presenti una serie di stringhe, una per ogni tabella, che verranno utilizzate nei metodi *onCreate()* e *onUpgrade()* per la loro creazione e modifica all'interno del database:

Listing 19: Creazione del DatabaseHelper, 2/2

```

1 ...
2
3 // Creazione del database
4 @Override
5 public void onCreate(SQLiteDatabase database)
6 {
7     database.execSQL(TABLE_CREATE);
8 }
9
10 // Upgrade del database , es. incremento il numero di versione
11 @Override
12 public void onUpgrade(SQLiteDatabase database, int oldVersion, int newVersion)
13 {
14     database.execSQL("DROP TABLE IF EXISTS example");
15     onCreate(database);
16 }
17 }

```

In quest'ultima parte di codice sono presentati i metodi *onCreate()* e *onUpdate()*, i quali devono necessariamente essere implementati, dato che si sta estendendo la classe astratta *SQLiteOpenHelper*. Il primo viene richiamato quando l'applicazione è stata appena installata e non fa altro che eseguire lo script SQL di creazione della base di dati definito dalle costanti di tipo stringa dichiarate nella prima parte di codice. Questo avviene mediante il metodo *execSQL()* degli oggetti di tipo *SQLiteDatabase*, che permette di eseguire del codice SQL arbitrario.

onUpdate(), invece, viene richiamato quando il database è già presente sul sistema, ma stando al numero di versione richiesta, risulta obsoleto. Di solito ciò avviene dopo aver cambiato

la versione di un'applicazione già installata. I parametri *oldVersion* e *newVersion* indicano rispettivamente la versione già installata del database e quella ancora da installare. In questo esempio il metodo *onUpdate()* effettua semplicemente il *DROP* (eliminazione) delle tabelle esistenti e la sostituisce con la nuova definizione delle stesse richiamando il metodo *onCreate()*.

2.7.2 La classe Adapter

Dopo aver definito una classe *Helper* è utile definirne una, denominata *DbAdapter*, che fornisce l'interfaccia diretta per manipolare il database. L'Adapter è un livello di astrazione: fornisce una serie di metodi che agevolano parecchio la gestione del database. Si riporta in Listing 20 un esempio:

Listing 20: Creazione del DatabaseAdapter

```
1 public class DbAdapter
2 {
3     private Context context;
4     private SQLiteDatabase database;
5     private DatabaseHelper dbHelper;
6
7     // Database fields
8     private static final String EXAMPLE_TABLE = "example";
9     public static final String KEY_ID = "_id";
10    public static final String KEY_FIELD1 = "field1";
11    public static final String KEY_FIELD2 = "field2";
12    ...
13
14    public DbAdapter(Context context)
15    {
16        this.context = context;
17    }
```

Nella prima parte sono state definite alcune proprietà ed alcune costanti che verranno utilizzate spesso all'interno della classe: la prima costante definisce il nome della tabella mentre le altre definiscono il nome di ogni colonna della stessa.

In questo modo in tutto il codice utilizzato per implementare l'Adapter verranno utilizzate tali costanti all'interno delle query: il principale vantaggio consiste nel poter modificare il nome di una variabile senza dover modificare ogni richiamo ad essa nel codice.

La parte di codice di Listing 20 riguarda la definizione del costruttore: l'unico elemento obbligatorio da configurare al suo interno è il *Context*, che deve essere inserito come argomento di ogni istanza di un nuovo *Adapter* per eseguire delle query. In ambiente Android il *Context* è un elemento fondamentale ed è rappresentato da una classe astratta, la cui implementazione è fornita dal sistema.

Esso permette di accedere alle risorse riservate all'applicazione, come ad esempio le Activity, gli Intent, ecc.

Listing 21: Creazione dei metodi *open* e *close*

```
1 public DbAdapter open() throws SQLException
2 {
3     dbHelper = new DatabaseHelper(context);
4     database = dbHelper.getWritableDatabase();
5     return this;
6 }
```

```

7
8 public void close()
9 {
10     dbHelper.close();
11 }

```

Nel codice riportato in Listing 21 sono implementati i metodi *open()* e *close()*. Nel metodo *open()* viene istanziato un oggetto di tipo *DatabaseHelper* che, come visto nel paragrafo 2.7.1, fornisce l'interfaccia di creazione/aggiornamento/gestione del database. Ora è sufficiente richiamare il metodo *getWritableDatabase()* definito in *SQLiteOpenHelper*, il quale restituisce un oggetto database in modalità lettura/scrittura attivo fino alla chiamata del metodo *close()*. La prima volta che viene richiamato *getWritableDatabase()* all'interno del codice, il database viene aperto e vengono automaticamente richiamati i metodi *onCreate()*, *onUpgrade()* e se necessario anche il metodo *onOpen()* (di cui non viene fornita un'implementazione). Il metodo *close()* invece, non fa altro che richiamare il metodo *close()* della classe *SQLiteOpenHelper*, il quale chiude ogni oggetto database aperto.

E' buona norma, ogni volta che all'interno del codice occorre interagire con la base di dati, chiudere le comunicazioni subito dopo aver terminato, in modo da risparmiare le limitate risorse dei dispositivi mobili.

I dati verranno memorizzati nella *cache memory*, per cui chiudere ed eventualmente riaprire il database quando necessario non rallenta l'esecuzione dell'applicazione.

Listing 22: Implementazione metodo per creazione record in database

```

1 private ContentValues createContentValues(String field1, String field2,...)
2 {
3     ContentValues values = new ContentValues();
4     values.put(KEY_FIELD1, field1);
5     values.put(KEY_FIELD2, field2);
6     ...
7     return values ;
8 }

```

Prima di trattare l'implementazione vera e propria dei metodi che si occupano di interrogare la base di dati, è consigliato implementare un altro metodo per migliorare l'efficienza e la comodità di utilizzo dell'Adapter, ovvero il metodo *createContentValues()* (Listing 22). Esso ha il compito di memorizzare un insieme di valori che lo strumento *ContentResolver* potrà poi processare per fornirne l'accesso condiviso da parte di altre applicazioni.

Qualora fosse necessario accedere ai dati di un Content Provider si può utilizzare l'oggetto *ContentResolver* all'interno del Context dell'applicazione e quindi comunicare con il Provider del database. Quest'ultimo, una volta ricevuta la richiesta, la gestisce e restituisce i risultati prodotti.

Nel Content Value riguardante l'esempio vengono inseriti un numero arbitrario di campi che costituiscono le colonne della tabella che si sta creando (*field - i*).

Una volta definito l'insieme di dati utilizzabili nelle query dell'applicazione attraverso il metodo *createContentValues()*, si approfondiscono nel dettaglio i metodi per creare, aggiornare e cancellare le tabelle.

2.7.3 Creazione di un record

Listing 23: Metodo per la creazione di un record

```

1 //create tables

```

```

2 public long createRecord(String field1, String field2,...)
3 {
4     ContentValues initialRecord = createContentValuesRecord(field1, field2,...);
5     return database.insert(EXAMPLE_TABLE, null, initialRecord);
6 }

```

Come visibile nel Listing 23, il metodo *createRecord()* richiede tanti parametri quanti sono le colonne della tabella. Nel codice di definizione della tabella, inoltre, è stata dichiarata una chiave primaria auto incrementale (*_id*), la quale durante la fase di creazione di un nuovo record verrà automaticamente incrementata, in modo da distinguere ogni riga della tabella.

Il metodo, dopo aver richiamato *createContentValuesRecord()*, utilizza l'istanza *SQLiteDatabase* impostata nel metodo *open()* per richiamare la funzione *insert()*. Essa permette di inserire un record nel database e come parametri prevede il nome della tabella in cui inserire la riga, un valore che indica cosa eseguire nel caso in cui i valori iniziali siano vuoti ed infine i valori da inserire.

Il parametro restituito è l'*id*, ovvero la chiave primaria del record appena creato.

2.7.4 Modifica di un record

Listing 24: Metodo per l'aggiornamento di un record

```

1 //update tables
2 public boolean updateRecord(long recordID, String field1, String field2, ...)
3 {
4     ContentValues updateValues = createContentValuesRecord(field1, field2, ...);
5     return database.update(EXAMPLE_TABLE, updateValues, KEY_ID + "=" + recordID,
6         null)>0;
7 }

```

L'aggiornamento di un record avviene in modo molto simile alla creazione. Il metodo *updateRecord()* (Listing 24) richiede gli stessi parametri del metodo *createRecord()*, oltre ad un parametro che indica l'identificatore della riga da aggiornare, *recordID*.

Anche in questo caso la prima operazione consiste nel richiamare il metodo *createContentValues()* e, successivamente, la funzione *update()* della classe *SQLiteDatabase* sull'istanza del database. Questo metodo richiede 3 parametri: il nome della tabella, l'oggetto *ContentValues* che contiene i valori del record da aggiornare e la dichiarazione della clausola *WHERE*. In questo caso si utilizza: *KEY ID + " = " + recordID*, che equivale alla stringa *id = identificatore del record* passato come parametro.

La clausola *WHERE* è opzionale: è possibile decidere di passare il parametro speciale *null*, il quale provoca l'aggiornamento di tutti i record della tabella.

2.7.5 Eliminazione di un record

Listing 25: Metodo per l'eliminazione di un record

```

1 //delete tables
2 public boolean deleteRecord(long recordID)
3 {
4     return database.delete(EXAMPLE_TABLE, KEY_ID + "=" + recordID, null)>0;
5 }

```

Il metodo *deleteRecord()* è piuttosto intuitivo: come mostrato in Listing 25, richiede un singolo parametro, ovvero l'*id* del viaggio da eliminare, e richiama il metodo *delete()* della classe *SQLiteDatabase*: questo non fa altro che eliminare il record con identificativo uguale al valore passato come parametro.

Dichiarando come parametro *recordID = null*, invece, vengono cancellate tutte le righe della tabella.

2.7.6 Utilizzo del database all'interno dell'Activity

Dopo aver trattato in modo generale i database ed i relativi metodi, diventa semplice implementare un *Helper* ed un *Adapter* molto efficaci e flessibili capaci di semplificare notevolmente l'utilizzo dei database nell'applicazione.

I database creati all'interno dell'imbarcazione contengono le informazioni descritte nelle tabelle seguenti.

- Tabella 2: credenziali utenti. Il primo database che viene creato contiene le informazioni configurata dall'utente. In tabella 2 sono riportati e descritti i valori in esso memorizzati:

Nome colonna	Tipo dato	Descrizione
username	String	Username applicazione
password	String	Password applicazione
numeroEKEEPER2	String	Numero SIM eKEEPER2
userServer	String	Username server
passwordServer	String	Password server
lingua	String	Lingua applicazione
logged	String	Memorizza credenziali di accesso all'applicazione

Tabella 2: Tabella credenziali utente

- Tabella 3: stato dell'imbarcazione. Questa tabella conserva tutti i dati che nel complesso costituiscono lo stato dell'imbarcazione e dello stato di funzionamento di eKEEPER2. Il gran numero di informazioni concentrato in tabella 3 verrà richiamato solo parzialmente da alcune Activity che sono dedicate alla gestione di una particolare sezione di questi dati.

Nome colonna	Tipo dato	Descrizione
dataAttuale	String	Data dell'informazione
oraAttuale	String	Orario dell'informazione
acquaSentina	Boolean	Presenza di acqua in sentina
effrazionePorta	Boolean	Apertura di un ingresso sorvegliato
staccoBatteria	Boolean	Chiusura dei contatti di isolamento della batteria preposta all'avvio del motore
taglioCavi	Boolean	Assenza del segnale di alimentazione dovuto al taglio dei cavi
assenzaRete230	Boolean	Assenza del segnale di alimentazione dalla rete 230V della banchina
batteriaServizi	Double	Tensione della batteria servizi
batteriaMotore	Double	Tensione della batteria motore
batteriaTampone	Double	Tensione della batteria tampone
temperatura	Double	Temperatura atmosferica
umidita	Double	Umidità atmosferica
rele1	Boolean	Accensione/spegnimento relè 1
rele2	Boolean	Accensione/spegnimento relè 2
rele3	Boolean	Accensione/spegnimento relè 3
trackON	Boolean	Abilitazione/disabilitazione del tracciamento degli spostamenti dell'imbarcazione
funzionamento	Integer	Valore di funzionamento di eKEEPER2
stato	Integer	Valore di stato di eKEEPER2
online	Boolean	Comunicazione di eKEEPER2 via server
ultimeCoordinate	String	Ultima coordinata dell'imbarcazione registrata

Tabella 3: Tabella stato imbarcazione

- Tabella 4: numeri abilitati ed allertabili. La tabella contiene la rubrica dei numeri abilitati a comunicare con eKEEPER2 ed i numeri contattabili da eKEEPER2 in caso di allarme. La rubrica prevede 5 slot per i numeri abilitati ed altrettante per gli allertabili. Nella tabella 4 è riportata la descrizione di un singolo slot per tipologia di numero.

Nome colonna	Tipo dato	Descrizione
abilitato1	String	Numero di cellulare abilitato alla comunicazione con eKEEPER2
allertabile1	String	Numero di cellulare allertabile da eKEEPER2 in caso di allarme

Tabella 4: Tabella numeri abilitati ed allertabili

- Tabella 5: Temperatura ed umidità delle ultime 24 ore. In ambito nautico avere la possibilità di monitorare per un periodo di tempo temperatura ed umidità risulta estremamente utile per prevenire la formazione di umidità, e quindi di muffe, all'interno del cabinato. L'applicazione mantiene in memoria una tabella dedicata a queste informazioni, dove ogni colonna è dedicata ad un'ora della giornata. In tabella 5 sono riportate solo le colonne dedicate alla memorizzazione delle misure di temperatura ed umidità relative alla prima ora della giornata:

Nome colonna	Tipo dato	Descrizione
temperatura1	Double	Temperatura registrata all'1 : 00
umidita1	Double	Umidità registrata all'1 : 00

Tabella 5: Tabella temperatura ed umidità 24 ore

- Tabella 6: coordinate di un viaggio. L'ultima tabella implementata nell'applicazione riguarda i viaggi registrati da eKEEPER2. La lista di record che verranno memorizzati in questo database presentano una struttura come quella descritta in tabella 6, dove il viaggio è identificato dalla data in cui è stato effettuato e le coordinate sono memorizzate all'interno di una lunga stringa dove ogni coppia di coordinate è separata dalla successiva da un carattere separatore

Nome colonna	Tipo dato	Descrizione
data	String	Data del viaggio
coordinate	String	Elenco di coordinate registrate durante il viaggio

Tabella 6: Tabella coordinate di viaggio

3 eKEEPER2, utilizzo del server

Prima di procedere con la descrizione completa dell'applicazione, è opportuno descrivere a fondo anche l'ultimo ma fondamentale agente responsabile dello scambio di informazioni tra eKEEPER2 ed il cliente: il cloud server.

Come si approfondirà nelle sezioni successive, la quantità di dati da memorizzare non è elevata, di conseguenza Aitech s.r.l. ha ritenuto sufficiente optare per l'affitto di un server base, che prevede 2 CPU da 3.01 GHz, 2 GB di memoria RAM, 30 GB di spazio di archiviazione, banda di 100 Mbit/s e traffico illimitato. Il sistema operativo installato è Ubuntu ed il servizio offre un backup settimanale.

3.1 Creazione di un ambiente LAMP su Linux

L'utilità del cloud server all'interno del progetto è dettata dalla possibilità di poter ricevere ed inviare informazioni sia ad eKEEPER2 che al cliente sull'applicazione. Per rendere tale operazione possibile occorre creare degli opportuni database MySQL (sezione 3.2) e sviluppare una serie di script PHP che interagiscano con le tabelle descritte in precedenza restituendo i valori desiderati (sezione 3.3).

Il primo step per realizzare tutto questo consiste nell'installare determinati strumenti sulla postazione Linux affittata da Aitech s.r.l. con distribuzione Debian-based, creando quello che viene chiamato un ambiente LAMP (Linux, Apache, MySQL e PHP).

3.2 Creazione delle tabelle MySQL

Le tabelle memorizzate all'interno del server sono molto simili a quelle già descritte per l'applicazione (sezione 2.7.6). In questa sezione verranno presentati brevemente i database generati sul server e verranno sottolineate le differenze rispetto alle tabelle conservate nell'applicazione, spiegandone l'utilità nel contesto delle operazioni sul cloud server.

- Database clienti: il primo database che viene creato è utilizzato da Aitech s.r.l. per creare un registro dei propri clienti ed associarli all'eKEEPER2 che hanno acquistato. Oltre ad alcune informazioni base del cliente, questo database contiene username e password assegnate al cliente per accedere alla propria sezione dedicata sul server. In tabella 7 sono riportate le voci del database clienti:

Nome colonna	Tipo dato	Descrizione
username	String	Username per l'accesso al server
password	String	Password per l'accesso al server
nome	String	Nome del cliente
cognome	String	Cognome del cliente
dataNascita	String	Data di nascita del cliente
luogoNascita	String	Luogo di nascita del cliente
domicilio	String	Indirizzo di domicilio del cliente
telefono	String	Numero di telefono dove il cliente è reperibile
email	String	Indirizzo email del cliente

Tabella 7: Tabella credenziali utente

Le informazioni più importanti in questa tabella sono *username* e *password*, le quali costituiscono l'unica chiave di accesso del cliente al server dove comunicare con il proprio

eKEEPER2. da notare che, per una questione di riservatezza e sicurezza, al cliente verrà notificato solamente l'username ad esso dedicato: sarà l'applicazione sullo smartphone, attraverso una routine automatica che è trattata in sezione 4.1, a collegarsi al server richiamando uno script PHP dedicato al recupero della password. In questo modo il cliente non è a conoscenza di entrambi i dati sensibili, ed è l'applicazione che recupera e memorizza autonomamente l'informazione di cui necessita.

Questa procedura previene anche il possibile furto di identità, in quanto non è possibile accedere ai database con le informazioni di eKEEPER2 conoscendo solamente l'username del cliente e non la password associata.

- Database stato eKEEPER2: come all'interno dell'applicazione, anche nel cloud server viene creato un database che conserva le informazioni circa lo stato dell'imbarcazione provenienti da eKEEPER2. La tabella è pressoché identica alla tabella 3, fatta eccezione per le colonne di tabella 9:

Nome colonna	Tipo dato	Descrizione
username	String	Identifica l'username al quale è dedicato il record nel database
password	String	Identifica la password associata all'username al quale è dedicato il record nel database

Tabella 8: Tabella stato eKEEPER2

Questo database è predisposto per contenere lo stato registrato da tutti gli eKEEPER2 operativi: per distinguere i diversi clienti si ricorre alle colonne *username* e *password*, le quali verranno confrontate con le credenziali di accesso fornite dal cliente. Per il dettaglio di questo passaggio si veda la sezione 3.3.2 che descrive lo script PHP associato al database.

- Database stato applicazione: per trasmettere le informazioni dallo smartphone al server si è deciso di utilizzare una tabella differente dalla precedente per evitare che i dati caricati dall'applicazione interferissero con quelli di eKEEPER2. La struttura della tabella è identica a quella del database stato eKEEPER2, ad eccezione di una singola colonna (si veda tabella 9):

Nome colonna	Tipo dato	Descrizione
nuovo	Boolean	Indica se il contenuto del record è stato modificato e quindi richiede di essere letto da parte di eKEEPER2

Tabella 9: Tabella stato eKEEPER2

In questo modo l'applicazione può inserire modifiche richieste dal cliente e segnalare ad eKEEPER2 che dovrà aggiornare il proprio stato ponendo il bit *nuovo* = 1. Attraverso uno script PHP (in sezione 3.3.2) eKEEPER2 preleverà da questa tabella le nuove informazioni, dopodiché porrà il bit *nuovo* = 0 indicando la riuscita dell'operazione.

- Database numeri: questa tabella è dedicata ad accogliere le informazioni provenienti dall'applicazione riguardanti i numeri telefonici abilitati ed allertabili. Come indicato precedentemente, la struttura del database è uguale a quella indicata in tabella 4, con la stessa aggiunta delle colonne *username*, *password* e *nuovo*, le quali hanno le medesime

funzioni descritte nel punto precedente.

Quindi, se il bit *nuovo* è posto a 1, eKEEPER2 eseguirà una richiesta al server per scaricare i numeri indicati dall'utente.

- Database temperatura ed umidità delle ultime 24 ore: in questo database le colonne (che hanno la stessa struttura indicata in tabella 5, con l'aggiunta di *username* e *password*) vengono popolate dalle misure che raccoglie nel corso della giornata eKEEPER2. Un apposito script (si veda sezione 3.3.4) è responsabile della ricezione dei dati e dell'allocazione all'interno del corretto slot temporale.
- Database viaggi: questa tabella contiene le informazioni relative ai viaggi effettuate dal cliente e registrate da eKEEPER2. Nuovamente, la struttura è come mostrato in tabella 6, a cui viene aggiunta la colonna di tabella 10:

Nome colonna	Tipo dato	Descrizione
scaricato	Boolean	Indica se il contenuto del record è stato scaricato dall'applicazione

Tabella 10: Tabella viaggi eKEEPER2

Questo bit viene utilizzato per discriminare un viaggio che è già stato trasferito nella memoria dell'applicazione da uno che al momento è presente solamente nel server: il suo utilizzo verrà approfondito nella sezione 3.3.5.

A differenza delle altre tabelle, dove ai diversi clienti corrisponde una diversa riga in costante aggiornamento, per i viaggi si dedica una singola tabella ad ogni cliente, le cui righe verranno popolate dai viaggi effettuati nel tempo.

Parallelamente a questo database opera una seconda tabella, la quale ha il compito di trasferire la richiesta di caricamento online di un viaggio dall'applicazione ad eKEEPER2. La forma del database è riportata in tabella 11:

Nome colonna	Tipo dato	Descrizione
username	String	Username per l'accesso al server
password	String	Password per l'accesso al server
richiestaCaricamento	Boolean	Indica se, tramite l'applicazione, il cliente ha richiesto ad eKEEPER2 di caricare le coordinate di viaggio registrate

Tabella 11: Tabella richiesta caricamento viaggi

- Database storico: infine, questo database è stato creato per uso esclusivo di Aitech s.r.l. nel compito di assistenza al cliente. In questa tabella vengono conservate tutte le modifiche che subiscono il database relativo allo stato di eKEEPER2, creandone una cronostoria consultabile in caso di problemi tecnici riscontrati dal cliente, ed il database contenente i numeri abilitati ed allertabili, per tenere traccia di quali sono gli utenti che possono interagire con eKEEPER2.

Il database, una per ogni cliente, ha la struttura riportata in tabella 12:

Nome colonna	Tipo dato	Descrizione
username	String	Username per l'accesso al server
password	String	Password per l'accesso al server
Data	String	Identifica l'istante temporale in cui è stata effettuata la modifica
inDatabase	String	Identifica in quale database è stata apportata una modifica
aggiornamento	String	Username per l'accesso al server

Tabella 12: Tabella viaggi eKEEPER2

3.3 Creazione degli script PHP

Si approfondiscono ora gli script PHP tramite i quali è effettivamente possibile lo scambio di informazioni tra l'applicazione ed eKEEPER2. I diversi codici hanno funzioni simili ma agiscono su tabelle concepite per alloggiare informazioni diverse (si vedano le diverse tabelle che illustrano le colonne dei diversi database), quindi ogni script ha un obiettivo specifico, agisce su una determinata tabella e necessita che l'agente (eKEEPER2 o applicazione) che lo interroga fornisca alcuni parametri.

3.3.1 Creazione utente

Seguendo l'ordine della presentazione dei database di sezione 3.2, il primo script PHP è responsabile della creazione ed inserimento di un nuovo utente nei sistemi di Aitech s.r.l..

Lo script in questione è denominato *creaUtente.php*, e viene richiamato da un *form* di una pagina HTML appositamente creata per semplificare l'inserimento dei dati del cliente (Fig. 15):

Creazione nuovo utente

Generalita' nuovo cliente

Nome	
Cognome	
Data di nascita	
Luogo di nascita	
Domicilio	
Telefono	
Email	

Credenziali nuovo cliente

Username: Password:

Figura 15: Pagina HTML per l'inserimento di un nuovo cliente

Dopo aver compilato tutti i campi di testo che riempiranno il record nell'apposito database, si invoca lo script *creaUtente.php* tramite il click del pulsante *Creazione utente*. Tale operazione eseguirà un *form* di HTML, riportato in Listing 26:

Listing 26: Script HTML per la creazione di un cliente

```
1 <form action="creaUtente.php" method="get">
2
3   <input type="text" name="NOM">
4   <input type="text" name="COG">
5   //La lista continua con altri elementi
6
7   <input type="submit" value="Creazione utente">
8
9 </form>
```

Il codice HTML ha il compito di prelevare le informazioni nei campi di testo ed associarle alle relative variabili che saranno fornite allo script PHP. Come si può vedere alla riga 1, il metodo tramite il quale avverrà l'invio dei parametri è di tipo *GET*, coerentemente con quanto programmato all'interno di *creaUtente.php*.

A questo punto il codice PHP procede all'interazione con i database MySQL: inizialmente deve fornire le credenziali di accesso al server per poter apportare delle modifiche ai database, quindi viene eseguito il metodo di accesso tramite apposito comando della libreria MySQL (Listing 27):

Listing 27: Script PHP per l'inserimento in database di un cliente, 1/3

```
1 <?php
2
3   $db_host = '127.0.0.1';
4   $db_user = '*****';
5   $db_pwd = '*****';
6
7   $nome = $_GET['NOM'];
8   $cognome = $_GET['COG'];
9   //GET delle altre variabili
10
11   if (!mysql_connect($db_host, $db_user, $db_pwd))
12       die("Can't connect to database");
```

Nel codice i valori di accesso vengono inseriti nelle variabili *\$db_user* e *\$db_pwd* (righe 4,5) e quindi utilizzate per effettuare la connessione (riga 11) dove, in caso di errore, è prevista una risposta di errore (riga 12).

Le variabili inserite nello script HTML e raccolte dal *form* sono ricevute da PHP tramite il metodo *GET* (righe 7-8).

Una volta in possesso di queste informazioni, *creaUtente.php* deve inserirle all'interno del database dedicato alle informazioni dei clienti, come mostrato nel Listing 28.

Listing 28: Script PHP per l'inserimento in database di un cliente, 2/3

```
1 $res = mysql_query("CREATE DATABASE IF NOT EXISTS db_clienti");
2 $res = mysql_query("USE db_clienti");
3
4 $res = mysql_query("CREATE TABLE IF NOT EXISTS tab_clienti (
5     ...
```

```

6 //Dichiarazione delle colonne della tabella
7 ...");
8
9 $res = mysql_query("SELECT user FROM tab_clienti WHERE user = '$username'");
10 if (!$res)
11 {
12     echo "Could not successfully run query ($sql) from DB: " . mysql_error();
13     exit;
14 }
15
16 if (mysql_num_rows($res) == 0)
17 {
18     echo "NUOVO CLIENTE";
19     $res = mysql_query("INSERT INTO tab_clienti (//lista delle colonne della
20         tabella)
21         VALUES
22         (//valori da inserire nelle colonne)");
23 }
24 else
25 {
26     echo "CLIENTE GIA' PRESENTE";
27 }

```

Per fare questo invoca una query che esegue le seguenti operazioni:

- Crea ed utilizza il database del cliente se questo non è già esistente (riga 1,2)
- All'interno del database selezionato crea la tabella dei clienti, se essa non esiste, inserendo la dichiarazione dei nomi di ogni colonna che compone la tabella (riga 4-7)
- Effettua una verifica sui dati ricevuti da HTML: in particolare se l'*username* associato al cliente dovesse essere per errore un duplicato, viene segnalato un errore ed interrotta la procedura (righe 9-14)
- Nel caso in cui il cliente inserito sia nuovo, e quindi unico, allora lo script PHP procede all'inserimento dei dati del cliente all'interno della tabella opportuna (righe 19-21)

Il passaggio successivo consiste nel creare le diverse tabelle: per brevità viene riportata, in Listing 29, una sezione di codice in cui si fa riferimento ad una *tabella* generica che prende il posto delle varie tabelle elencate in sezione 3.2.

Listing 29: Script PHP per l'inserimento in database di un cliente, 3/3

```

1 //Creazione, se non esiste, della tabella di stato imbarcazione ed inserimento dei
  campi della tabella
2 $res = mysql_query("CREATE TABLE IF NOT EXISTS tabella (//Dichiarazioni delle
  colonne della tabella)");
3
4 //Verifica che l'utente non sia un duplicato
5 $res = mysql_query("SELECT user FROM tabella WHERE user = '$username'");
6 if (!$res) {
7     echo "Could not successfully run query ($sql) from DB: " . mysql_error();
8     exit;
9 }
10 //Se l'username non compare mai nelle righe allora e' nuovo ed inserisco,
    altrimenti segnalo errore

```

```

11  if (mysql_num_rows($res) == 0)
12  {
13      echo "NUOVO UTENTE";
14      $res = mysql_query("INSERT INTO tabella(//lista delle colonne della tabella)
                          VALUES(//Valori da inserire)");
15  }
16  else{
17      echo "UTENTE GIA' PRESENTE ---";
18  }

```

Si sottolinea, alle righe 5 – 14, che il codice prevede l'esecuzione di un check di unicità dell'username inserito, questo ovviamente per prevenire la presenza contemporanea di due identificativi utente uguali.

3.3.2 Download ed upload stato imbarcazione

Il corretto utilizzo dello stato registrato da eKEEPER2 è sicuramente una delle parti più delicate da gestire all'interno di tutta l'applicazione: ciò che si vuole evitare è che l'interazione tra app su smartphone ed eKEEPER2 crei una sovrascrittura dei dati memorizzati dall'uno o dall'altro. Per questo motivo si è scelto di duplicare le tabelle relative allo stato dedicandone una alla scrittura da parte di eKEEPER2 e lettura da parte dell'applicazione, mentre la seconda tabella funziona a ruoli invertiti: in questo modo non è possibile che una delle due tabelle contenga contemporaneamente informazioni provenienti sia da eKEEPER2 che dall'applicazione.

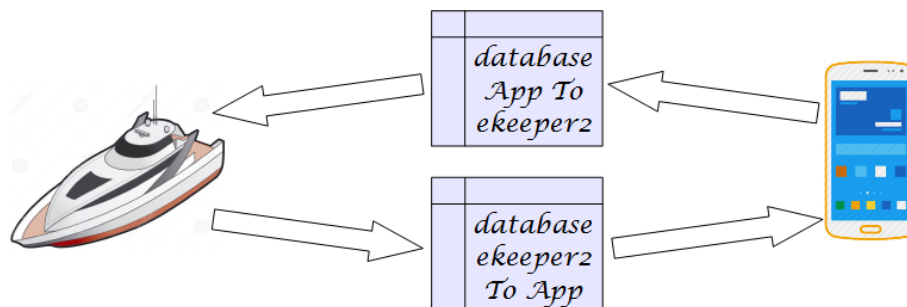


Figura 16: Schema di comunicazione tra eKEEPER2 ed applicazione

Per assecondare lo schema di comunicazione rappresentato in Fig. 16 vengono creati più script PHP: il primo che viene presentato effettua il download da parte dell'applicazione dello stato di eKEEPER2, *downloadStato.php* (Listing 47).

Listing 30: Script PHP per il download dello stato da applicazione

```

1  //Dichiarazione del database selezionato
2  $res = mysql_query("USE db_stato");
3  //Seleziona l'intera riga dedicata all'utente identificato da username e password
4  $res = mysql_query("SELECT * FROM tab_statoeKEEPER WHERE user='$username' AND
                      password='$password'");
5
6  //Ottenuta la risposta della query, la si presenta in una pagina testuale
   contenente le informazioni richieste
7  while($row = mysql_fetch_row($res))
8  {
9      // $row e' un array, foreach(...) inserisce ogni elemento di $row in una
      variabile $cell

```

```

10  foreach($row as $cell)
11      echo "$cell:";
12 }

```

Il risultato finale produce una stringa di testo visualizzata su pagina HTML dove tutti i campi della tabella sono riscritti in riga separati tra di loro da un carattere separatore, in questo caso i due punti. L'applicazione scarica la suddetta stringa e ne suddivide i campi secondo l'ordine prestabilito, quindi li memorizza all'interno del proprio database dedicato allo stato dell'imbarcazione.

Allo stesso modo, per trasmettere ad eKEEPER2 le informazioni relative alle modifiche sullo stato effettuate dall'applicazione, viene utilizzato un apposito script, *downloadStatoeKEEPER.php*: il codice è identico a quello riportato in Listing 47, al quale viene aggiunta la modifica del bit di controllo della colonna *nuovo*, come mostrato in Listing 31.

Listing 31: Script PHP per il download dello stato da eKEEPER

```

1 $sql = "UPDATE tab_stato SET nuovo='0' WHERE user='$username' AND
    password='$password'";
2 $res = mysql_query($sql);

```

Si ricorda che, al fine di segnalare ad eKEEPER2 che l'applicazione ha modificato lo stato, il bit *nuovo* viene impostato a 1 quindi, una volta che lo stato è stato scaricato correttamente da eKEEPER2, esso viene reimpostato a 0.

Per quanto riguarda il lato di caricamento di un nuovo stato, esistono altrettanti codici PHP: il codice presentato in Listing 32, *uploadStatoeKEEPER.php*, mostra la modalità tramite la quale eKEEPER2 aggiorna i database sul server con le informazioni raccolte.

Listing 32: Script PHP per l'upload dello stato da eKEEPER

```

1 $stato = $_GET['STA'];
2
3 list("Nomi delle colonne della tabella") = split(':', $stato);
4
5 $batteriaTampone = substr(((string)((doubleval($battTamp)*15)/1024)),0,3);
6 $batteriaServizi = substr(((string)((doubleval($battServ)*15)/1024)),0,3);
7 $batteriaMotore = substr(((string)((doubleval($battMotore)*15)/1024)),0,3);
8
9 $sql = "UPDATE tab_statoeKEEPER SET colonna=valore WHERE username='$username' AND
    password='$password'";
10
11 $res = mysql_query("SELECT * FROM tab_statoeKEEPER WHERE user='$username' AND
    password='$password'");
12
13 while($row = mysql_fetch_row($res))
14 {
15     //Crea una stringa contenente lo stato completo da inserire nello storico
16     foreach($row as $cell)
17         $risultato .= $cell. ":";
18 }
19
20 $res = mysql_query("USE db_storico");

```

```

21 $username_storico = $username."_storico";
22
23 $sql = "INSERT INTO $username_storico (//Nomi delle colonne) VALUES (//Valori da
    inserire)";
24 $res = mysql_query($sql);

```

Alla riga 1, lo script PHP recupera il valore associato da eKEEPER2 alla variabile *STA* all'interno dell'URL, per poi suddividerlo in un array i cui elementi sono suddivisi dal carattere : (riga 3). Il passaggio delle informazioni al server avviene tramite metodo *GET*.

Nelle righe 5 – 7 si applica il rapporto di partizione ai valori di tensione delle batterie registrati da eKEEPER2.

Successivamente, i valori sono pronti per essere inseriti in memoria all'interno del database tramite il comando *UPDATE* applicato alla riga della tabella individuata in maniera univoca dalla condizione *WHERE username = ' \$username' AND password = ' \$password'* (riga 11). Infine, dalla riga 13 alla 24, lo script si occupa di reperire il nuovo stato completo di eKEEPER2 e di memorizzarlo all'interno del database *db_storico*, agevolando in questo modo il compito di assistenza al cliente da parte di Aitech S.r.l..

Lo script equivalente per caricare sul server lo stato dell'applicazione, *uploadStato.php* (Listing 33 e 34), ricalca in buona parte il funzionamento del codice appena descritto. Lo script richiede quattro variabili fornite tramite metodo *GET*: *username*, *password*, *parametro* e *valore*. I primi due fungono da credenziali d'accesso dell'utente e lo collegano alla riga del database ad egli dedicato, mentre i rimanenti parametri individuano la colonna di cui si deve aggiornare il valore.

Listing 33: Script PHP per l'upload dello stato da applicazione, 1/2

```

1 $username = $_GET['USR'];
2 $password = $_GET['PWD'];
3 $parametro = $_GET['PAR'];
4 $valore = $_GET['VAL'];

```

A suddette istruzioni si aggiungono i seguenti controlli:

Listing 34: Script PHP per l'upload dello stato da applicazione, 2/2

```

1 $res = mysql_query("SELECT nuovo FROM tab_stato WHERE username='$username' AND
    password='$password'");
2
3 $row = mysql_fetch_array($res);
4 if($row['nuovo']=='0')
5 {
6     $res = mysql_query("SELECT * FROM tab_statoeKEEPER WHERE user='$username' AND
        password='$password'");
7
8     $statoConcatenato = "";
9     while($row = mysql_fetch_row($res))
10     {
11         foreach($row as $cell)
12             $statoConcatenato = $statoConcatenato. "$cell:";
13     }
14
15     list("Elementi che compongono lo stato") = split(":", $statoConcatenato);
16

```

```

17 $res = mysql_query("UPDATE tab_stato SET colonna stato=elemento stato");
18
19 $sql = "UPDATE tab_stato SET $parametro='$valore', nuovo='1', WHERE
    username='$username' AND password='$password'";

```

Alla riga 1, lo script PHP legge il valore della variabile nuovo, per riconoscere il caso in cui eKEEPER2 abbia già scaricato le informazioni inviate dalla app (*nuovo* = 0), oppure debba ancora richiamare lo script *downloadStato.php* (*nuovo* = 1).

Nel primo caso, dunque, scarica lo stato attuale di eKEEPER2 (riga 6) e lo concatena in una stringa unica (riga 12). Successivamente inserisce in un array gli elementi dello stato e quindi li carica nel database (righe 13-17), modificando un singolo parametro di interesse (righe 17-19). Sempre alla riga 19, si nota che il valore logico di *nuovo* viene automaticamente impostato ad 1, per segnalare ad eKEEPER2 la presenza di nuove informazioni riguardanti lo stato.

La scelta di modificare un singolo parametro invece di fornire al server una stringa contenente l'intero stato è dettato da due motivazioni:

- minor utilizzo di traffico dati: l'applicazione per comunicare via internet con il server può utilizzare una rete WiFi oppure appoggiarsi alla rete dati dell'operatore telefonico. Nel secondo scenario, il traffico mensile a disposizione dell'utente è limitato a seconda del piano tariffario stabilito. Inoltre, tale traffico è sicuramente consumato anche da altre applicazioni dello smartphone, motivo per cui si è deciso di rendere la comunicazione cellulare-server la più essenziale possibile, richiedendo la modifica di un singolo elemento al massimo
- semplicità nell'aggiornare i database: modificando una sola variabile per volta, l'aggiornamento della relativa colonna del database risulta molto più rapida non dovendo eseguire operazioni di suddivisione delle variabili ed associazione delle stesse alle relative celle all'interno delle tabelle

Infine, esiste uno script PHP per il caricamento dello stato di eKEEPER2 ad opera dell'applicazione. Questo codice, che sembra non seguire la logica di Fig. RIFERIMENTO SCHEMA COMUNICAZIONE, viene impiegato nel caso in cui eKEEPER2 non riesca a comunicare via internet con l'applicazione, e sia quindi costretto ad usufruire della strada alternativa: gli SMS. Ovviamente in questo caso le informazioni scambiate tra smartphone ed eKEEPER2 non giungono al server e, di conseguenza, non vengono inserite nel database che conserva lo storico delle operazioni. Per ovviare a tale problema, dunque, l'applicazione si fa carico di trasferire tali informazioni, ricevute tramite messaggio, al server dove vengono aggiornate e conservate fintanto che eKEEPER2 non è in grado di comunicare via rete cellulare.

I casi in cui viene invocato tale script, denominato *uploadStatoApp.php*, sono limitati: esso interviene solamente nel caso in cui

- uno dei tre dispositivi controllabili da eKEEPER2 cambia il proprio stato
- eKEEPER2 modifica il proprio funzionamento abilitando la protezione dell'imbarcazione
- eKEEPER2 modifica il proprio funzionamento abilitando il tracciamento dell'imbarcazione

Tutti i casi sopra descritti fanno riferimento ad un preciso parametro dello stato, quindi sono associati ad una specifica colonna nella tabella presente sul server: attraverso *uploadStatoApp.php*, lo smartphone aggiorna tale valore puntando l'apposita cella tramite l'istruzione

```

1 $sql = "UPDATE tab_statoeKEEPER SET $parametro='$valore' WHERE user='$username' AND
    password='$password'";

```

Successivamente aggiorna il database *db_storico* secondo le modalità esposte negli script precedenti.

3.3.3 Download ed upload numeri abilitati ed allertabili

La trasmissione dei numeri abilitati ed allertabili tra eKEEPER2, smartphone e server avviene in maniera simile a come vengono gestite le informazioni relative allo stato.

In questo caso, tuttavia, non esistono due distinte tabelle per la memorizzazione delle informazioni in quanto l'unico dispositivo che può apportare delle modifiche alla lista è lo smartphone mentre, d'altra parte, eKEEPER2 può al massimo confermare la lista di numeri impostata dall'utente via SMS.

Il primo script PHP approfondito è *downloadNumeri.php* (Listing 35):

Listing 35: Script PHP per il download dei numeri

```
1 $res = mysql_query("USE db_numeri");
2 $res = mysql_query("SELECT * FROM tab_numeri WHERE user='$username' AND
   password='$password'");
3
4 while($row = mysql_fetch_row($res))
5 {
6     foreach($row as $cell)
7         echo "$cell:";
8 }
```

Questo script viene richiamato dall'applicazione per richiedere al server la lista dei numeri abilitati ed allertabili, separati tra di loro dal carattere `:`.

Il codice preleva l'intero contenuto della riga dedicata all'utente identificato dall'*username* e dalla *password* fornite tramite l'URL (riga 2), quindi crea una pagina testuale dove disporre i dati separati tra di loro dai due punti (righe 4-7).

Lo script *downloadNumerieKEEPER.php* ha il medesimo funzionamento, eccezione fatta per la seguente porzione di codice:

```
1 $sql = "UPDATE tab_numeri SET nuovo='0' WHERE user='$username' AND
   password='$password'";
```

Questa istruzione serve a segnalare che eKEEPER2 ha correttamente scaricato la lista di numeri che deve inserire in memoria, quindi cambia il bit che indica la presenza di un nuovo set di dati ponendolo a 0.

Il caricamento sul server dei numeri avviene, sia per l'applicazione che per eKEEPER2, fornendo tramite l'URL la lista completa dei numeri da inserire nella tabella, separati dal carattere `:`.

All'interno di *uploadNumeri.php* vengono eseguite le istruzioni riportate in Listing 36:

Listing 36: Script PHP per l'upload dei numeri

```
1 list($abil1, $abil2, $abil3, $abil4, $abil5, $alert1, $alert2, $alert3, $alert4,
   $alert5) = split(':', $valore);
2
3 $res = mysql_query("UPDATE tab_numeri SET $parametro='$valore' WHERE
   user='$username' AND password='$password'");
```

In questo modo la riga dedicata all'utente viene aggiornata, sostituendo all'interno di ogni cella il numero ad essa dedicato. Viene inoltre impostato il bit *nuovo* = 1 per segnalare la presenza di nuovi dati.

Lo script prevede, inoltre, l'inserimento di questa operazione all'interno del database dedicato allo storico (Listing 47):

Listing 37: Script PHP per l'aggiornamento dello storico

```
1 $res = mysql_query("SELECT * FROM tab_numeri WHERE user='$username' AND
    password='$password'");
2
3 while($row = mysql_fetch_row($res))
4 {
5     foreach($row as $cell)
6         $risultato .= $cell.":";
7 }
8
9 $sql = "INSERT INTO $username_storico (//Nomi delle colonne) VALUES (//Valori da
    inserire)";
```

L'inserimento dei numeri impostati dal cliente all'interno di *db_stato* serve principalmente in ottica di assistenza al cliente da parte di Aitech s.r.l.

eKEEPER2 richiama uno script del tutto simile per effettuare l'upload dei numeri abilitati ed allertabili che ha memorizzato, *uploadNumerieKEEPER.php*. L'unica eccezione riguarda la porzione di codice responsabile dell'inserimento delle informazioni all'interno della tabella *db_storico*, dove viene specificato che l'informazione proviene da eKEEPER2.

3.3.4 Download ed upload di temperatura ed umidità nel corso delle 24 ore

In questa sezione vengono presentati gli script PHP responsabili dell'inserimento di temperatura ed umidità rilevate nel corso della giornata all'interno delle tabelle dedicate.

Questi script, *uploadTemperatura.php* (Listing 38) ed *uploadUmidita.php*, vengono richiamati da eKEEPER2 periodicamente, il quale fornisce tramite l'URL la misura effettuata durante l'ultima ora.

Listing 38: Script PHP per l'upload della temperatura

```
1 $valore = $_GET['VAL'];
2 $orari = array("dichiarazione delle 24 celle della tabella");
3
4 $indiceArray = substr(date(Hms),0,2);
5
6 $parametro = $orari[intval($indiceArray)];
7
8 $sql = "UPDATE tab_temp24 SET $parametro='$valore' WHERE user='$username' AND
    password='$password'";
```

Una volta ricavato il valore di temperatura (od equivalentemente umidità) tramite *GET* (riga 1), lo script crea un array temporaneo contenente i nomi delle colonne della tabella (riga 2).

Successivamente reperisce l'orario corrente e ne estrapola il valore delle ore (riga 4): tale numero, che varia da 0 a 23, fungerà da puntatore all'interno dell'array precedentemente creato. In questo modo l'orario reperito dal server identifica automaticamente la cella all'interno della quale deve essere associata l'informazione relativa alla temperatura (riga 6). Infine, lo script

inserisce la misura nel database (riga 8).

Gli script che restituiscono le misurazioni delle ultime 24 ore, *downloadTemperatura.php* e *downloadUmidita.php*, sono consultabili solo dal cliente tramite applicazione. Anche in questo caso, il funzionamento è molto semplice, come già visto per il download di stato e numeri:

Listing 39: Script PHP per il download della temperatura

```
1 $res = mysql_query("SELECT * FROM tab_temp24 WHERE user='$username' AND
    password='$password'");
2
3 while($row = mysql_fetch_row($res))
4 {
5     foreach($row as $cell)
6         echo "$cell:";
7 }
```

Lo script individua la riga dell'utente tramite *username* e *password*, quindi ne preleva l'intero contenuto e lo dispone in una pagina di testo dove i valori sono separati da `:`.

3.3.5 Download ed upload viaggio

Per la gestione del database contenente le informazioni relative ai viaggi registrati da eKEEPER2 vengono utilizzati tre script PHP. Il primo che viene approfondito riguarda la modalità di caricamento del viaggio da parte di eKEEPER2 tramite *uploadViaggio.php*, riportato in Listing 40:

Listing 40: Script PHP per l'upload del viaggio

```
1 $coordinate = $_POST['CRD'];
2 $orario = substr(date('d/m/Y'),0,6).substr(date('d/m/Y'),8);
3
4 $sottoStringa="";
5 $viaggioFinale="";
6 $i=0;
7 $dati = array('orario','long','lat','funz','stato');
8
9 while($i<strlen($coordinate))
10 {
11     $inizio = $i;
12     $fine = 1;
13     if(substr($coordinate,$inizio,$fine)!=="~")
14     {
15         $sottoStringa = $sottoStringa.substr($coordinate,$inizio,$fine);
16     }
17     elseif(substr($coordinate,$inizio,$fine)=="~")
18     {
19         $dati = split(':', $sottoStringa);
20         if($dati[2]=="(6)") //Identifica un viaggio, quindi concatena
21         {
22             $viaggioFinale = $viaggioFinale.$sottoStringa;
23         }
24         $sottoStringa="";
25     }
26     $i++;
```

```

27 }
28
29 $res = mysql_query("INSERT INTO $username (//Nomi delle colonne) VALUES (//Valori
    da inserire)");
30
31 $sql = "UPDATE tab_caricaViaggio SET carica='0', oraRef='$orario' WHERE
    user='$username' AND password='$password'";

```

eKEEPER2 fornisce la lista di coordinate registrate durante il viaggio tramite l'URL, quindi PHP reperisce tale informazione effettuando il *POST* della variabile "*CRD*" (riga 1).

Per identificare in maniera univoca un viaggio, esso viene associato alla data in cui viene registrato, la quale viene composta nel formato *gg/mm/aa* (riga 2).

Segue un breve algoritmo che ha lo scopo di eliminare dai dati forniti da eKEEPER2 le informazioni superflue: come si vede alla riga 7, la stringa *CRD* contiene i valori di orario della coordinata, i relativi valori di longitudine e latitudine ed infine i valori che indicano il funzionamento e lo stato di eKEEPER2. I diversi campi di dati sono divisi tra loro dal carattere *:*, mentre le diverse sequenze di dati sono separate dal carattere *" ~"*.

Cercando i caratteri speciali, l'algoritmo scorre l'intera stringa (riga 9), individuando le porzioni contenenti le informazioni del punto registrato (riga 17), ed infine verificando che il funzionamento associato alla posizione sia equivalente a 6 (riga 20). Si ricorda che, durante un viaggio, il funzionamento 6 indica che l'imbarcazione si sta muovendo con velocità superiore ad 1 nodo, mentre il funzionamento 1 consente il tracciamento ma la velocità di spostamento della barca è inferiore alla soglia indicata (si veda sezione PARTE DI CODICE DI ARDUINO).

Se il gruppo di informazioni relative alla singola coordinata individuata dall'algoritmo verifica la condizione di *funzionamento* = 6 (riga 20), allora tale coordinata viene concatenata alla stringa che conterrà l'intero viaggio finale (riga 22), così da mantenere solo le coordinate effettivamente associate al viaggio.

Una volta terminato l'algoritmo, PHP lancia la query per inserire all'interno del database dei viaggi l'intera stringa appena ottenuta, associandola alla data in cui è stato effettuato il viaggio (riga 29). Infine, lo script prevede l'azzeramento del bit che segnala ad eKEEPER2 la richiesta di caricamento viaggio (riga 31).

Lo script *uploadCaricaViaggio.php* (in Listing 41) viene richiamato dall'applicazione qualora l'utente desideri richiedere ad eKEEPER2 di inserire nelle tabelle del server i viaggi che ha in memoria.

Listing 41: Script PHP per l'upload richiesta caricamento viaggio

```

1 $valore = $_GET['VAL'];
2
3 $sql = mysql_query("UPDATE tab_caricaViaggio SET carica='$valore', oraRef='$orario'
    WHERE user='$username' AND password='$password'");

```

Lo script è estremamente semplice e consiste nel cambiare il valore logico del bit *carica* ponendolo uguale a 1 all'interno della tabella dedicata all'utente (riga 3). L'applicazione fornisce il valore del bit tramite il parametro *VAL* (riga 1).

Infine, lo script utilizzato dall'applicazione per scaricare i viaggi è denominato *downloadViaggio.php*, riportato in Listing 42:

Listing 42: Script PHP per il download viaggio

```

1 $res = mysql_query("SELECT * FROM $username WHERE user='$username' AND
    password='$password' AND scaricato=0");
2
3 while($row = mysql_fetch_row($res))
4 {
5     foreach($row as $cell)
6         echo "$cell:";
7 }
8
9 $sql = mysql_query("UPDATE $username SET scaricato='1' WHERE scaricato='0'");

```

Alla riga 1, la query legge nel database viaggi tutte le righe individuate dalla condizione *scaricato* = 0, ovvero tutti i viaggi che non sono stati scaricati e memorizzati dall'applicazione. Il risultato della query fornisce una lunga stringa contenenti i valori dei viaggi, i quali verranno rielaborati all'interno dell'applicazione per renderli presentabili (si veda sezione 4.6). Al termine dello script, il bit *scaricato* viene settato a 1, facendo sì che il viaggio non venga scaricato ripetutamente (riga 9).

3.3.6 Script PHP per funzioni ausiliarie

Per concludere la panoramica sugli script PHP che operano all'interno del server, vengono ora trattati tre ulteriori script PHP che vengono attivati in precisi contesti, ed hanno il compito di fornire un gruppo di informazioni ristretto ma fondamentale per il buon funzionamento delle comunicazioni tra eKEEPER2 e l'applicazione, passando tramite il server.

- *eKEEPEROnline.php*: questo script (si veda Listing 43) viene periodicamente interrogato dall'applicazione, la quale necessita di sapere se eKEEPER2 si è connesso al server di recente e quindi è in grado di reperire le informazioni tramite le tabelle online.

Listing 43: Script PHP per il download del bit eKEEPEROnline

```

1 $username = $_GET['USR'];
2 $password = $_GET['PWD'];
3
4 $res = mysql_query("SELECT eKEEPEROnline FROM tab_statoeKEEPER WHERE
    user='$username' AND password='$password'");
5
6 while($row = mysql_fetch_row($res))
7 {
8     foreach($row as $cell)
9         echo "$cell";
10 }

```

Lo script prevede che l'applicazione si autentichi fornendo *username* e *password*, reperiti tramite *GET* da PHP (righe 1-2). Successivamente, la query che viene avviata restituisce un singolo carattere di risposta (righe 6-9), il quale indica se eKEEPER2 è raggiungibile via server (*eKEEPEROnline* = 1) o meno.

- *downloadPassword.php* (Listing 44): al momento della registrazione all'interno dell'applicazione (si veda sezione 4.1), al cliente è richiesto di inserire il numero seriale dell'eKEEPER2 acquistato. Tale codice è parte delle credenziali di accesso all'area riservata al cliente nel server. Oltre al serial number, che costituisce l'username, è necessario associare una password, la quale non viene consegnata direttamente al cliente per questioni

di tutela delle informazioni da parte di Aitech s.r.l..

Tuttavia, per aggiungere tale informazione ai database dello smartphone senza che il cliente debba fare ulteriori passaggi, l'applicazione richiama questo script PHP il quale restituisce la password associata al numero seriale. In questo modo le informazioni vengono completate automaticamente dallo smartphone senza che il cliente se ne accorga.

Listing 44: Script PHP per il download della password

```
1 $res = mysql_query("SELECT password FROM tab_clienti WHERE user='$username'");
2
3 while($row = mysql_fetch_row($res))
4 {
5     foreach($row as $cell)
6         echo "$cell";
7 }
```

L'esecuzione del codice è standard: viene individuato il campo relativo alla password associata all'username fornito dal cliente (riga 1), quindi viene generata una pagina testuale che l'applicazione provvederà a scaricare ed inserire in memoria (righe 3-6).

- *interrogaDatabase.php* (Listing 45): tra gli script ausiliari, questo è il codice PHP più importante, nonché il responsabile della corretta comunicazione tra eKEEPER2 e l'applicazione. Esso ha la duplice funzione di indicare ad eKEEPER2 su quali database sono avvenuti dei cambiamenti e quindi ci sono nuove informazioni da scaricare e, contemporaneamente, tiene traccia della periodica connessione al server.

Listing 45: Script PHP per il download dei dati aggiornati

```
1 $res = mysql_query("DROP EVENT IF EXISTS aggiornaOnline");
2
3 $res = mysql_query("SELECT nuovo FROM tab_stato WHERE user='$username' AND
4     password='$password'");
5
6 if($row['nuovo']=='1')
7 {
8     echo "aitech.serverlet.com/URLscript:";
9 }
10 else
11 {
12     echo "0:";
13 }
14
15 $res = mysql_query("UPDATE tab_statoeKEEPER SET eKEEPEROnline='1' WHERE
16     user='$username' AND password='$password'");
17
18 $res = mysql_query("SET GLOBAL event_scheduler = ON");
19
20 $res = mysql_query("CREATE EVENT IF NOT EXISTS aggiornaOnline ON SCHEDULE AT
21     NOW() + interval 2 minute DO UPDATE tab_statoeKEEPER SET eKEEPEROnline =
22     '0' WHERE user='$username' AND password='$password'");
```

Quando lo script viene eseguito, questo elimina l'evento *aggiornaOnline* (riga 1), ovvero un evento schedulato associato alla riga della tabella dedicata all'utente che aggiorna il bit *eKEEPEROnline* (riga 18).

Lo script scorre le tabelle di stato, numeri e viaggio, ricercando in essi la presenza di informazioni aggiornate, individuati dalla condizione *nuovo* = 1 (riga 3). Se la ricerca da esito positivo, lo script riporta un URL che eKEEPER2 dovrà visitare per scaricare i nuovi dati (riga 7). In caso contrario restituisce il valore "0".

Una volta effettuato il controllo dei database lo script notifica che eKEEPER2 è in grado di accedere al server quindi imposta il bit *eKEEPEROnline* = 1 (riga 14).

Infine, l'evento precedentemente eliminato viene ricreato e resettato: in questo modo si crea un timer con timeout di 2 minuti, trascorsi i quali il bit *eKEEPEROnline* viene posto a 0, segnalando all'applicazione che tramite il server le informazioni non possono giungere ad eKEEPER (righe 16-18). In quest'ultimo caso la comunicazione tra smartphone e dispositivo a bordo avverrà via SMS.

4 L'applicazione eKEEPER2

Di seguito viene presentata l'applicazione sviluppata per comunicare con eKEEPER2 attraverso smartphone Android.

4.1 Schermata *Home* e creazione di un account

Una volta scaricata da Google Play Store, l'applicazione si presenta come in Fig. 17: l'interfaccia è semplice ed intuitiva, concedendo all'utente la sola possibilità di registrarsi per poter utilizzare le altre funzionalità del programma. La schermata è organizzata secondo un linear layout con i pulsanti posizionati a centro schermo: dei due solamente il secondo è attivo in quanto l'utente è invitato ad inserire le proprie credenziali ed indicare i dati dell'eKEEPER2 che intende associare prima di procedere all'utilizzo.

Selezionando il pulsante *Registrati* l'applicazione richiama una seconda Activity, *SignUpActivity*, raffigurata in Fig. 18. Anche in questo caso si è utilizzato un Linear Layout all'interno del quale sono stati inseriti i campi di testo che l'utente dovrà compilare: tali informazioni servono a popolare le colonne del database trattato in tabella 2.

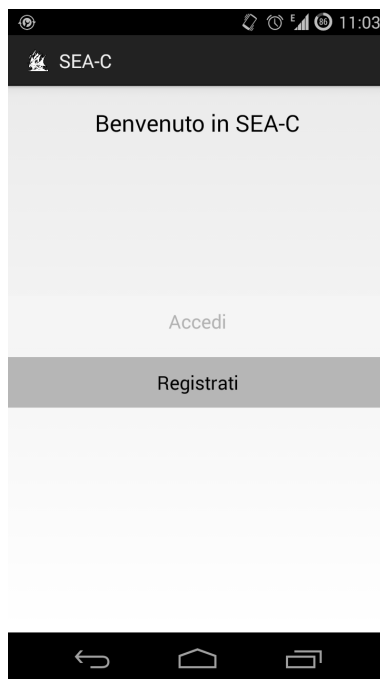


Figura 17: HomeActivity, schermata di accesso

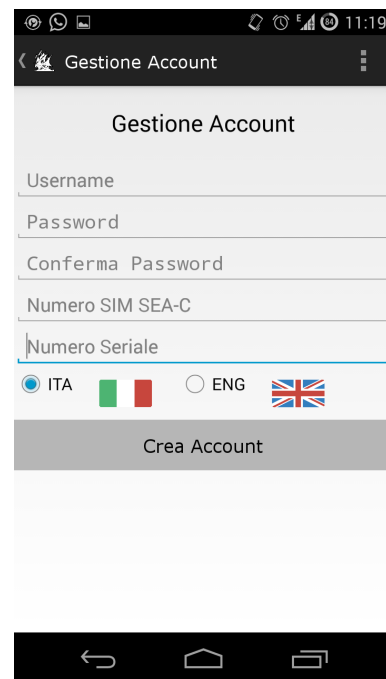


Figura 18: SignUpActivity, schermata di creazione utente

Si approfondisce ora il codice che gestisce gli eventi delle activity appena presentate:

- *HomeActivity*: i primi due controlli che l'activity deve compiere al proprio avvio riguardano la lingua selezionata dall'utente (se la selezione non è stata fatta si imposta di default l'inglese) e la dimensione del database dedicato alle informazioni utente. Una dimensione nulla indica che non esistono record all'interno del database quindi, per evitare errori dovuti a *nullPointerException*, il pulsante *Accedi* viene reso inattivo in modo che il *listener* ad esso associato non possa eseguire il proprio codice.

Listing 46: HomeActivity, attivazione e disattivazione dei pulsanti

```
1 //Se non c'e' un account l'unico pulsante abilitato e' quello di creazione
2 if(databaseH.dbSize()>0)
```

```

3 {
4     btnSignIn.setBackgroundColor(Color.parseColor("#B6B6B6"));
5     btnSignUp.setBackgroundColor(Color.parseColor("#F5F5F5"));
6
7     btnSignIn.setEnabled(true);
8     btnSignUp.setEnabled(false);
9 }
10 else
11 {
12     btnSignIn.setBackgroundColor(Color.parseColor("#F5F5F5"));
13     btnSignUp.setBackgroundColor(Color.parseColor("#B6B6B6"));
14
15     btnSignIn.setEnabled(false);
16     btnSignUp.setEnabled(true);
17 }

```

Oltre a rendere i pulsanti attivi o meno (righe 7-8 e 15-16) vengono anche modificati i colori di sfondo per rendere intuibile visivamente l'inaccessibilità di tale pulsante (righe 4-5 e 12-13).

L'azione collegata alla pressione del pulsante *Registrati* è mostrata in Listing 47:

Listing 47: HomeActivity, listener del pulsante *Registrati*

```

1 btnSignUp.setOnClickListener(new View.OnClickListener() {
2     public void onClick(View v) {
3         Intent intentSignUp=new
4             Intent(getApplicationContext(),SignUpActivity.class);
5         startActivity(intentSignUp);
6     }
7 });

```

Il *listener*, quando percepisce l'evento *onClick*, genera un *Intent* allo scopo di chiamare l'esecuzione di *SignUpActivity*.

- *SignUpActivity* (Listing 48): al proprio avvio, anche questa activity effettua un controllo sulla lingua selezionata e presenta il layout coerente con l'impostazione dell'utente. Prima ancora di provvedere alla raccolta delle informazioni inserite dal cliente, l'activity provvede a creare il database per lo stato dell'imbarcazione (tabella 3) inserendo una riga i cui valori sono tutti 0 (riga 5).

Listing 48: SignUpActivity, inizializzazione database

```

1 databaseStatoNatante = new DatabaseStatoNatante(this);
2
3 if(databaseStatoNatante.dbSize()==0)
4 {
5     databaseStatoNatante.createStatoNatante(new StatoNatante(/*Lista di "0"*/));
6 }
7
8 databaseStatoNatante.close();

```

Siccome questa activity utilizza diversi database, i quali hanno strutture differenti tra di loro, ogni qualvolta occorre aprirne uno lo si inizializza come indicato in riga 1, mentre al termine del suo utilizzo occorre chiuderlo (riga 8). Questo evita di produrre eccezioni

nella gestione delle tabelle.

In Listing 49 si approfondisce il rilevamento delle informazioni inserite dall'utente:

Listing 49: SignUpActivity, listener del pulsante *Crea Account*

```
1 btnCreateAccount.setOnClickListener(new View.OnClickListener(){
2     public void onClick(View v)
3     {
4         String userName=editTextUserName.getText().toString();
5         String password=editTextPassword.getText().toString();
6         String confirmPassword=editTextConfirmPassword.getText().toString();
7         String phone="+39"+editTextPhone.getText().toString();
8         String usernameServer = editUserServer.getText().toString();
9
10        reperisciPassword(usernameServer, phone);
11    ...
}
```

Una volta compilati i campi di testo il cliente preme il pulsante *Crea Account* il quale registra l'evento ed estrae sotto forma di stringa le informazioni (righe 4-8).

Successivamente, alla riga 10, l'applicazione esegue un metodo particolare, incaricato di connettersi al server per reperire la password associata all'utente (si veda sezione 3.3.6): tale metodo effettua un *Asynchronous Task*, ossia un task che viene eseguito in background parallelamente al normale ciclo vita dell'applicazione. Esso è composto da tre sottoroutine fisse:

1. *onPreExecute*: è la porzione di codice che stabilisce quali istruzioni vengono eseguite prima di lanciare in background il task. In generale serve per allestire le variabili e le strutture che verranno successivamente utilizzate. Nel caso dell'applicazione in questione, tale metodo non viene utilizzato in quanto superfluo
2. *doInBackground*: le istruzioni asincrone vengono eseguite all'interno di questa subroutine. L'applicazione, per mezzo del codice riportato in Listing 50, esegue la richiesta allo script PHP presente sul server e ne gestisce il risultato

Listing 50: SignUpActivity, connessione al server

```
1 URL url = new URL(urlServer);
2 HttpURLConnection conn = (HttpURLConnection)url.openConnection();
3 conn.setRequestMethod("GET");
4 conn.setDoInput(true);
5 conn.connect();
6
7 int response = conn.getResponseCode();
8 is = conn.getInputStream();
9
10 String contentAsString = readIt(is, len);
11 return contentAsString;
```

Viene inizializzato l'oggetto *URL* per la connessione al server (riga 1), quindi vengono impostati alcuni parametri per specificare che tipo di connessione si sta creando (righe 2-5). Il metodo, una volta ottenuta la risposta del server, gestisce il flusso di caratteri (righe 7-8), quindi effettua una conversione del contenuto in stringa (righe 10-11).

3. *onPostExecute*: infine il metodo (il cui codice è riportato nel Listing 51) gestisce la chiusura del task asincrono. In questo caso si utilizza questa subroutine per alloggiare i dati appena scaricati all'interno dell'apposito database:

Listing 51: SignUpActivity, aggiornamento del database

```
1 LoginDataBaseAdapter settaggiUtente = databaseH.readSettings(1);
2
3 String[] separated = result.split(":");
4
5 settaggiUtente.setPswServer(separated[0]);
6 databaseH.updateSettings(settaggiUtente);
7 databaseH.close();
8
9 sendSMS(phone, "COD00:" + userServer);
```

Per inserire le informazioni nel database desiderato è necessario inizializzare la variabile ad esso associata (riga 1), quindi il metodo separa i campi della stringa precedentemente scaricata dal server (riga 3). Il campo di interesse è il primo dell'array, ossia quello che contiene la password per l'accesso al server (riga 5), quindi viene inserito all'interno del database (riga 6).

Infine, per comunicare i dati appena reperiti ad eKEEPER2, l'applicazione invia un SMS contenente un codice identificativo delle informazioni allegate (riga 9). Per la trattazione dei codici scambiati via SMS tra eKEEPER2 e l'applicazione si veda [16].

Listing 52: SignUpActivity, verifica dei valori inseriti

```
1 ...
2
3 int selectedId = radioLinguaGroup.getCheckedRadioButtonId();
4 String linguaggio=radioLinguaButton.getText().toString();
5 String logged = "false";
6
7 if(linguaggio.equals("ITA"))
8     lingua("it");
9 else if(linguaggio.equals("ENG"))
10    lingua("en");
11
12 if(userName.equals("")||password.equals("")||
13 confirmPassword.equals("")||phone.equals("")||usernameServer.equals(""))
14 {
15     Toast.makeText(getApplicationContext(),R.string.campoVuoto,
16         Toast.LENGTH_LONG).show();
17     return;
18 }
19 if(!password.equals(confirmPassword))
20 {
21     Toast.makeText(getApplicationContext(),R.string.stringPassErrata,
22         Toast.LENGTH_LONG).show();
23     return;
24 }
25 else
26 {
```

```

25     databaseH.createSettings(new LoginDataBaseAdapter(/*Valori da inserire
        nel database*/));
26 }
27 });

```

La *SignUpActivity* prosegue con il codice riportato in Listing 52 rilevando quale lingua è stata selezionata dal cliente e provvede ad impostarla (righe 3-10), quindi esegue un controllo di correttezza dei campi compilati, in particolare che nessuno di essi sia stato lasciato vuoto (righe 12-16) oppure che la password inserita combaci con quella riconfermata (righe 18-21). Se tutti i controlli vengono superati, i valori inseriti vengono introdotti nel database (riga 25), in caso contrario l'utente riceve una notifica a schermo e deve provvedere a modificare i campi di testo.

4.2 Schermata *Home* ed accesso all'applicazione

Dopo aver completato la routine di *SignUpActivity*, il cliente deve premere il tasto di navigazione *indietro*, il quale scorre la *back stack* alla *HomeActivity*.

Ora l'activity in esecuzione verifica che i database abbiano dimensione non nulla, quindi reperisce le credenziali di accesso e si connette al server per scaricare le informazioni più recenti: questo avviene come illustrato in Fig 19.

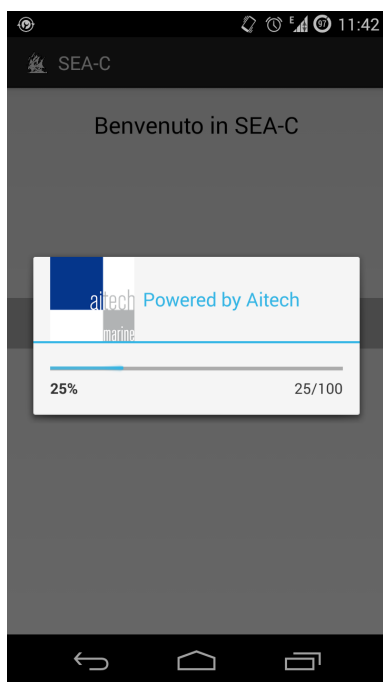


Figura 19: HomeActivity, collegamento al server

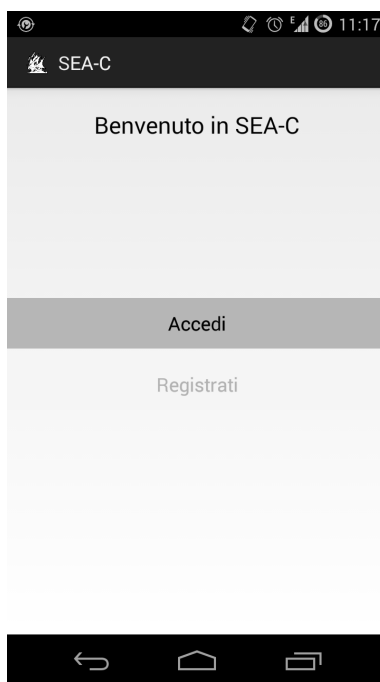


Figura 20: HomeActivity, schermata di accesso

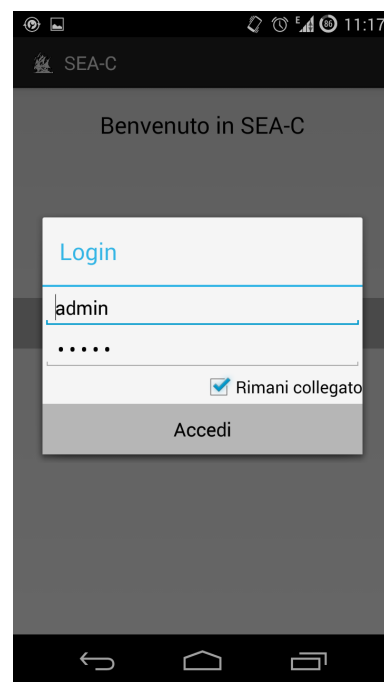


Figura 21: HomeActivity, inserimento delle credenziali

Il task eseguito è asincrono, come visto precedentemente, tuttavia in questo caso è utile sfruttare anche il metodo *onPreExecute* per generare l'interfaccia grafica del caricamento, indicando titolo icona dimensione e modalità di riempimento della barra.

Successivamente l'activity si connette in background al server richiedendo lo script PHP per il download dello stato (sezione 3.3.2) quindi, tramite il metodo *onPostExecute*, inserisce le informazioni aggiornate nel database dedicato.

Ora viene visualizzata la schermata *Home* di inizio applicazione, il cui pulsante *Accedi* ora risulta attivo in quanto sono presenti dei dati relativi al cliente nel database (Fig. 20). Al contrario, non è più necessario creare un account, quindi il pulsante *Registrati* viene disabilitato.

Il cliente per accedere alle altre activity dell'applicazione deve fornire le credenziali precedentemente impostate: questo costituisce un minimo livello di protezione e tutela da chi non è il possessore dello smartphone, poichè per arrivare ai successivi livelli dell'applicazione è necessario conoscere *username* e *password*.

In Fig. 21 è mostrata la finestra di dialogo che richiede l'inserimento dei dati utente, oltre alla possibilità di richiedere l'autocompilazione degli stessi selezionando *Rimani collegato*.

Listing 53: HomeActivity, finestra di accesso

```
1 String username=editTextUsername.getText().toString();
2 String password=editTextPassword.getText().toString();
3
4 String logged = "";
5 if(check.isChecked()==true)
6     logged = "true";
7 else
8     logged = "false";
9
10 list = databaseH.getAllSettings();
11 LoginDataBaseAdapter listSettaggi = databaseH.readSettings(1);
12 listSettaggi.setLogged(logged);
13 databaseH.updateSettings(listSettaggi);
14 String storedUsername=list.get(0).getUsr();
15 String storedPassword=list.get(0).getPsw();
16
17 if(username.equals(storedUsername) && password.equals(storedPassword))
18 {
19     dialog.dismiss();
20     Intent intentLoadingScreen=new
        Intent(getApplicationContext(),TabularActivity.class);
21     startActivity(intentLoadingScreen);
22 }
23 else
24 {
25     Toast.makeText(HomeActivity.this,R.string.stringUserPwd,
        Toast.LENGTH_LONG).show();
26 }
```

Premendo il pulsante *Accedi* della finestra di dialogo, l'activity compie le operazioni riportate nel codice in Listing 53: inizialmente preleva i valori di *username* e *password* forniti dall'utente (righe 1-2), quindi interroga il database dell'applicazione ricercando i medesimi valori (righe 9-15). Ora viene effettuato il confronto tra i dati forniti e quelli memorizzati quindi, se il confronto dà esito positivo, viene richiamata l'activity successiva: *TabularActivity* (riga 20). In caso di dati errati viene richiesto tramite popup a schermo di reinserire i dati corretti (riga 25).

4.3 TabularActivity, il contenitore di activity

Dopo aver effettuato con successo il login, l'utente passa dalla *HomeActivity* ad una activity che funge da contenitore: *TabularActivity*. Questo ambiente permette di organizzare la schermata suddivisa per sezioni, ognuna delle quali contengono un'activity specifica. In Fig. 22, nella zona superiore, si possono vedere tre caselle individuate da *Stato*, *Dispositivi* e *Viaggi*: queste sono le tre sezioni generate da *TabularActivity* associate ad altrettante activity. Nel Listing

54 si riporta il codice per dichiarare una tab (riga 1), indicandone il titolo (riga 3) e l'activity che contiene (riga 4), quindi la si aggiunge a *TabularActivity*.

Listing 54: TabularActivity, contenitore di activity

```
1 TabSpec tab1 = tabHost.newTabSpec("First Tab");
2
3 tab1.setIndicator(getString(R.string.tabStato));
4 tab1.setContent(new Intent(this, StatoActivity.class));
5
6 tabHost.addTab(tab1);
```

Nella parte superiore di Fig. 22 si nota, a fianco del titolo dell'activity, l'icona di una busta: tale simbolo viene utilizzato per comunicare all'utente quale via di comunicazione utilizzerà l'applicazione per scambiare dati con eKEEPER2. Nel dettaglio, l'icona viene aggiornata con cadenza di 1 minuto, periodo che intercorre tra una visita al server e la successiva. Durante la connessione al server, viene richiamato lo script PHP responsabile del controllo del bit *eKEEPEROnline* (sezione 3.3.6).

Nel Listing 55 è riportato il codice responsabile della generazione dell'evento periodico:

Listing 55: TabularActivity, gestione della connessione periodica al server

```
1 public void timer()
2 {
3     t = new Timer();
4     task = new TimerTask()
5     {
6         @Override
7         public void run()
8         {
9             runOnUiThread(new Runnable()
10             {
11                 @Override
12                 public void run()
13                 {
14                     bitOnline();
15                 }
16             });
17         }
18     };
19     t.scheduleAtFixedRate(task, 0, 60000);
20 }
```

Il codice comprende un oggetto *Timer* (riga 3) che si occupa di monitorare il tempo trascorso tra un'azione e la successiva, ed un oggetto *TimerTask* che si occupa di gestire un task in un determinato istante di tempo (riga 4). Come indicato a riga 19, il timer viene resettato ogni 60000 millisecondi, quindi con cadenza di un minuto viene eseguito il task che reperisce il bit indicante la presenza di eKEEPER2 sul server (righe 7-14).

Una volta ottenuto il bit in questione, scaricato tramite il task asincrono, si effettua il refresh degli elementi del menù (Listing 56).

Listing 56: TabularActivity, refresh del contenuto del manù

```
1 tabMenu.clear();
2 onCreateOptionsMenu(tabMenu);
```

```

3
4 @Override
5 public boolean onCreateOptionsMenu(Menu menu)
6 {
7     getMenuInflater().inflate(R.menu.manage, menu);
8
9     MenuItem item = menu.add(Menu.NONE, R.id.action_settings, 10, "");
10    item.setEnabled(false);
11
12    databaseStatoNatante = new
13 DatabaseStatoNatante(getApplicationContext());
14    initStateNatante = databaseStatoNatante.readStatoNatante(1);
15
16    if((initStatoNatante.getOnline()).equals("0"))
17    {
18        item.setIcon(R.drawable.email);
19    }
20    else
21    {
22        item.setIcon(R.drawable.antenna);
23    }
24    databaseStatoNatante.close();
25 }

```

Nel Listing 56 viene richiamata l'azione di refresh dei contenuti del menù (righe 1-2). Il metodo *onCreateOptionsMenu* ridisegna il menù dell'applicazione aprendo il file di stile *menu.xml* (riga 7), quindi si inizializzano gli elementi del menù (righe 9-11) ed infine si impostano i valori ricavati dal database ad essi associati (righe 12-21). In questo caso se il valore restituito da *getOnline* è 0 allora l'icona mostrata è una busta, viceversa si mostra l'icona di un'antenna. In questo modo si segnala visivamente all'utente se eKEEPER2 è connesso al server o meno.

4.4 StatoActivity, monitoraggio dell'imbarcazione

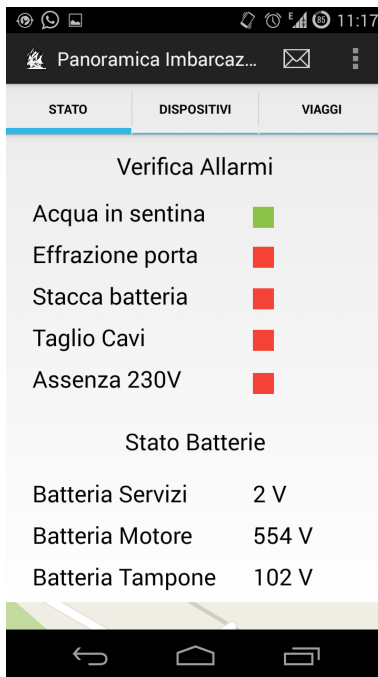


Figura 22: StatoActivity, parte superiore

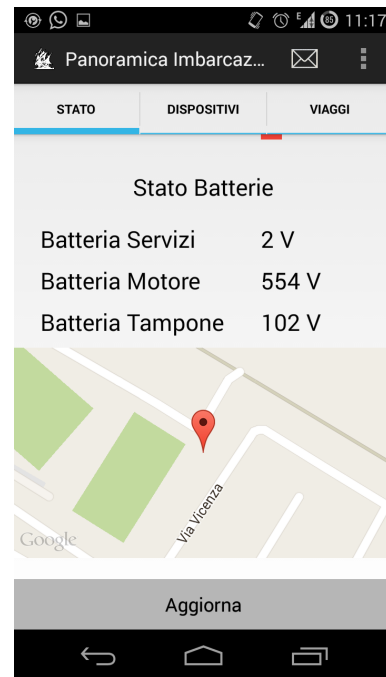


Figura 23: StatoActivity, parte inferiore

Si giunge all'analisi della prima activity che mostra all'utente i dati relativi all'imbarcazione: la prima schermata, *Stato*, inserisce tutte le informazioni provenienti dai punti sensibili associando un riquadro di colore verde se non vengono rilevate anomalie, altrimenti rosso in caso di problema.

Listing 57: StatoActivity, gestione colore allarmi

```
1 if((databaseStatoNatante.readStatoNatante(1).getAcqua()).equals("0"))
2 {
3     buttonAcqua.setBackgroundColor(Color.parseColor("#8BC34A"));
4 }
5 else if((databaseStatoNatante.readStatoNatante(1).getAcqua()).equals("1"))
6 {
7     buttonAcqua.setBackgroundColor(Color.parseColor("#F44336"));
8 }
```

Per ognuno dei punti monitorati da eKEEPER2 viene ricercato all'interno del database il bit ad essi associati, il quale viene confrontato come mostrato nel Listing 57, dove si imposta il colore opportuno per ogni riquadro (righe 3 e 7).

Successivamente viene visualizzato lo stato delle batterie, altra zona sensibile che può indicare possibili malfunzionamenti o guasti nel sistema. L'indicazione che viene associata alla descrizione riportante il nome della batteria è la tensione della stessa. L'utente, conoscendo che tipo di batteria monta la propria imbarcazione, può giudicarne il livello di carica ed in caso intervenire.

Listing 58: StatoActiviti, tensione delle batterie

```
1 tensBattServ.setText((databaseStatoNatante.readStatoNatante(1).getBattServ()+" V");
2 tensBattMot.setText((databaseStatoNatante.readStatoNatante(1).getBattMot()+" V");
3 tensBattTamp.setText((databaseStatoNatante.readStatoNatante(1).getBattTamp()+" V");
```

Come mostrato nel Listing 58, la procedura è simile a quella riportata per i riquadri rossi o verdi, ovvero si ricava la tensione dal database dello stato e si inserisce tale informazione all'interno di un TextView (righe 1-3).

In Fig. 23 mostra l'ultima parte della schermata che riporta l'ultima coordinata dell'imbarcazione registrata da eKEEPER2.

Listing 59: StatoActivity, mappa ultima posizione

```
1 mappa = ((MapFragment) getFragmentManager().findFragmentById(R.id.map)).getMap();
2 posNatante = new LatLng(Double.valueOf(latLong[1]), Double.valueOf(latLong[0]));
3 mappa.addMarker(new MarkerOptions().position(posNatante).title("Ultima Posizione"));
4 mappa.moveCamera(CameraUpdateFactory.newLatLng(posNatante));
5 mappa.animateCamera(CameraUpdateFactory.zoomTo(17));
```

Nel riquadro Listing 59, alla riga 1 viene inizializzato l'oggetto di tipo mappa, quindi si crea un oggetto *LatLng* contenente le coordinate del punto in questione (riga 2). Tale coordinata viene aggiunta alla mappa (riga 3) ed infine vengono sistemate le impostazioni della camera puntata sulla mappa (righe 4-5).

L'ultimo elemento interattivo di questa activity è il pulsante *Aggiorna* in fondo a Fig 23: questo elemento è responsabile dell'aggiornamento del database relativo allo stato. Effettuando un click su tale pulsante viene generato un task asincrono che effettua la connessione al server dove lo script *downloadStatoeKEEPER.php* restituisce tutte le informazioni aggiornate del dispositivo. La stringa ricevuta viene suddivisa nei diversi campi e quindi inserita nel database dell'applicazione. Ora l'activity deve essere ricaricata per permettere gli aggiornamenti grafici dovuti ai nuovi dati scaricati:

Listing 60: StatoActivity, pulsante aggiorna

```
1 databaseStatoNatante.updateStatoNatante(settaggiUtente);
2
3 if(databaseStatoNatante.readStatoNatante(1).getOnline().equals("0"))
4 {
5     sendSMS(numeroeKEEPER,"COD09:1");
6 }
7
8 Context context = getBaseContext();
9 Intent finale = new Intent(context, TabularActivity.class);
10 Bundle b=new Bundle();
11 b.putString("condition","tab1");
12 finale.putExtras(b);
13 finale.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
14 context.startActivity(finale);
```

Nel Listing 60 è riportata la procedura completa per l'aggiornamento: dopo aver inserito le nuove informazioni nel database (riga 1), l'applicazione effettua il controllo sul bit *eKEEPEROnline*. Questo passaggio è necessario in quanto, se tale bit fosse 0, indicherebbe che eKEEPER2 non è connesso al server, quindi le informazioni presenti online potrebbero non essere aggiornate. In questo scenario l'applicazione provvede automaticamente a comunicare con il dispositivo tramite la via alternativa, ossia l'SMS (riga 4), per la gestione del quale si rimanda alla sezione 4.8.

Al termine della comunicazione l'applicazione viene rilanciata per inserire i nuovi dati in schermata (righe 7-13).

4.5 DispositiviActivity, la gestione da remoto

La seconda activity contenuta nelle tab è *DispositiviActivity*: questa interfaccia è dedicata al controllo di tre contatti a bordo dell'imbarcazione, al monitoraggio di temperatura ed umidità all'interno della cabina ed infine permette di modificare il funzionamento di eKEEPER2.

I tre contatti elettrici che si possono attivare o disattivare da remoto sono collegati all'impianto elettrico dell'imbarcazione tramite dei relays, i quali non sono inclusi nel prodotto finito ma le cui specifiche per essere compatibili con eKEEPER2 vengono fornite da Aitech s.r.l.. L'utente utilizza questi contatti per poter accendere o spegnere alcuni dispositivi che ha installato a bordo, una lista dei quali può essere la seguente:

- dispositivo di riscaldamento
- dispositivo per generare acqua calda
- dispositivo di raffreddamento

Tali dispositivi possono fornire dei servizi di utilità per l'utente. Ad esempio, mentre il possessore dell'imbarcazione è in viaggio verso di essa, può richiedere tramite l'applicazione di azionare uno dei tre dispositivi elencati così da trovare l'ambiente già riscaldato o raffreddato all'arrivo. L'utilità dell'azionamento a distanza è ancora maggiore nel caso della stagione fredda durante la quale la barca è esposta alle intemperie che accumulano umidità all'interno della cabina: per evitare che la presenza di forte umidità rovini gli interni è possibile azionare il riscaldamento e monitorare come evolvono la temperatura e l'umidità misurate dagli appositi sensori.



Figura 24: DispositiviActivity, assenza sensore di temperatura e umidità

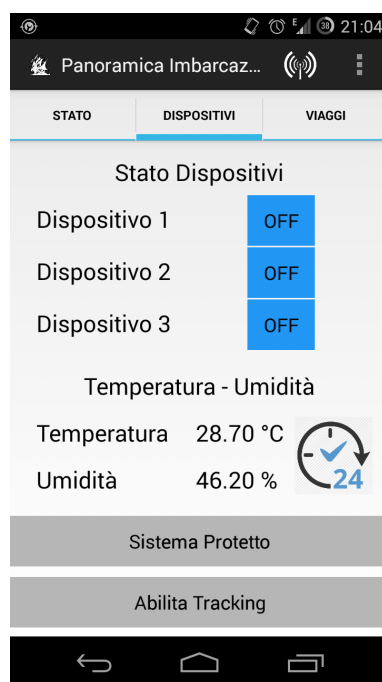


Figura 25: DispositiviActivity, presenza sensore di temperatura e umidità

La parte che implementa i controlli da remoto è mostrata in Fig. 24 e Fig. 25, nella parte superiore. L'utente vede lo stato dei dispositivi a fianco del nome che identifica gli stessi: il funzionamento è individuato da una scritta *ON/OFF* associata al colore verde (*ON*) oppure blu (*OFF*).

Listing 61: DispositiviActivity, indicazione di stato dispositivo

```

1 if((databaseStatoNatante.readStatoNatante(1).getRele1()).equals("0"))
2 {
3     rele1.setBackgroundColor(Color.parseColor("#2196F3"));
4     rele1.setText("OFF");
5 }
6 else if((databaseStatoNatante.readStatoNatante(1).getRele1()).equals("1"))
7 {
8     rele1.setBackgroundColor(Color.parseColor("#8BC34A"));
9     rele1.setText("ON");
10 }

```

In Listing 61 si riporta la procedura di estrapolazione delle informazioni dal database (righe 1 e 6) e le successive modifiche apportate agli elementi grafici in accordo con lo stato assunto dal dispositivo (righe 3-4 e 8-9).

Per variare lo stato di un dispositivo è sufficiente premere sul pulsante che ne riporta lo stato, il quale esegue il codice riportato in Listing 62

Listing 62: DispositivitiActivity, commutazione stato di dispositivo

```

1 rele1.setOnClickListener(new View.OnClickListener() {
2     public void onClick(View v) {
3         if((databaseStatoNatante.readStatoNatante(1).getRele1()).equals("0"))
4         {
5             rele1.setBackgroundColor(Color.parseColor("#607D8B"));
6             rele1.setText("-");
7             if(databaseStatoNatante.readStatoNatante(1).getOnline().equals("0"))
8             {
9                 sendSMS(numeroKeeper, "COD05:1:1");
10            }
11            else
12            {
13                statoDispositivo("releD1", "1");
14            }
15        }
16        else if((databaseStatoNatante.readStatoNatante(1).getRele1()).equals("1"))
17        {
18            rele1.setBackgroundColor(Color.parseColor("#607D8B"));
19            rele1.setText("-");
20            if(databaseStatoNatante.readStatoNatante(1).getOnline().equals("0"))
21            {
22                sendSMS(numeroKeeper, "COD05:1:0");
23            }
24            else
25            {
26                statoDispositivo("releD1", "0");
27            }
28        }
29    }
30 });

```

Una volta rilevato il tocco sul pulsante, l'applicazione verifica qual'era lo stato precedentemente assunto dal dispositivo (righe 3 e 16), quindi ne modifica l'aspetto, impostando un colore grigio e sostituendo lo stato *ON/OFF* con il carattere "-" (righe 5-6 e 18-19): questo serve a comunicare all'utente che il cambiamento di stato è in corso. La successiva operazione consiste

nel verificare quale via di comunicazione è adatta per inviare ad eKEEPER2 le informazioni desiderate: alla riga 7 viene verificato il valore del bit *online* e, nel caso in cui esso sia 0, viene inviato un SMS contenente un codice identificativo dell'operazione, il relay da azionare e lo stato che deve assumere. Se il bit *online* risulta 1 allora le informazioni vengono inviate al server attraverso un task asincrono simile a quelli già descritti: lo script PHP che verrà richiamato è *uploadStato.php* (sezione 3.3.2).

Al termine del task asincrono l'applicazione salva i dati nei propri database e ricarica l'activity per apportare i cambiamenti grafici necessari.

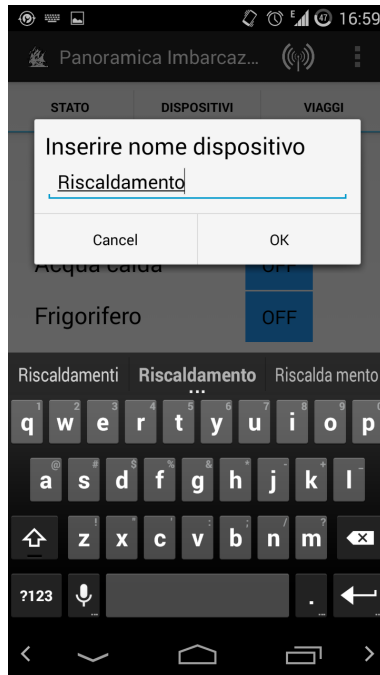


Figura 26: DispositiviActivity, personalizzazione dei dispositivi: inserimento nome



Figura 27: DispositiviActivity, personalizzazione dei dispositivi: risultato finale

In Fig. 26 e Fig. 27 viene mostrato un aspetto lasciato alla personalizzazione del cliente: l'applicazione permette infatti di rinominare i dispositivi in modo da individuare chiaramente quello installato a bordo che si vuole comandare. Per realizzare questa funzionalità ogni TextView dei dispositivi è dotato di un *listener* il quale si attiva in caso di pressione prolungata sull'elemento, come mostrato nel Listing 63

Listing 63: DispositiviActivity, personalizzazione dei dispositivi

```

1 dispositivo1.setOnLongClickListener(
2     new OnLongClickListener() {
3
4         @Override
5         public boolean onLongClick(View v) {
6             ...
7             alertDialogBuilder.setCancelable(false)
8                 .setPositiveButton("OK",new DialogInterface.OnClickListener()
9                 {
10                 public void onClick(DialogInterface dialog,int id)
11                 {
12                     dispositivo1.setText(userInput.getText().toString());
13                     init= databaseH.readSettings(1);
14                     init.setRele1(userInput.getText().toString());

```

```

15         databaseH.updateSettings(init);
16     }
17     }).setNegativeButton("Cancel",new DialogInterface.OnClickListener()
18     {
19         public void onClick(DialogInterface dialog,int id)
20         {
21             dialog.cancel();
22         }
23     });
24     ...
25 }
26 });

```

Quando viene rilevato il tocco prolungato (riga 1), l'applicazione reagisce generando una finestra di dialogo (riga 7) dove viene richiesto di inserire il nome da associare al dispositivo. La finestra di dialogo dispone di due tasti:

- *OK* il quale ricava la stringa inserita dall'utente, la memorizza nel database dell'applicazione nella cella dedicata al dispositivo selezionato (righe 12-15)
- *Cancel* il quale chiude la finestra senza apportare modifiche (riga 21)

Facendo nuovamente riferimento alle Fig. 24 e Fig. 25, si può notare come in una di esse non siano presenti informazioni circa temperatura ed umidità mentre nell'altra siano mostrate ed affiancate ad un'icona riportante un orologio: il motivo è dovuto alla presenza o meno del sensore di temperatura ed umidità collegato ad eKEEPER2, il quale effettua un check sulla lettura dei dati provenienti dal sensore (si veda sezione DHT ARDUINO). Nel caso in cui il sensore non sia stato acquistato dal cliente, eKEEPER2 rileva un errore in lettura quando interroga il sensore, non essendo fisicamente presente. Di conseguenza anche sull'applicazione tale valore viene sostituito dal carattere "-", il quale segnala all'utente che nessuna informazione circa temperatura ed umidità è disponibile.

Qualora il sensore venga acquistato e funzioni correttamente, l'applicazione riceve le informazioni e provvede ad inserirle nella schermata, come mostrato nel Listing 64

Listing 64: DispositiviActivity, inserimento di temperatura ed umidità

```

1 temperatura.setText(databaseStatoNatante.readStatoNatante(1).getTemperatura()+" C");
2 umidita.setText(databaseStatoNatante.readStatoNatante(1).getUmidita()+" %");
3
4 if(temperatura.getText().equals("- C") && umidita.getText().equals("- %"))
5 {
6     btnGraph.setEnabled(false);
7     btnGraph.setClickable(false);
8     btnGraph.setActivated(false);
9     btnGraph.setAlpha(0);
10 }

```

Alle righe 1 e 2 i valori di temperatura ed umidità vengono inseriti nei TextView dedicati. Qualora il sensore non fosse presente si inserisce il simbolo "-" e si procede a disattivare e rendere non visibile il pulsante con l'icona dell'orologio (righe 4-9).

Viceversa, se il pulsante è attivo e visibile, ossia nel caso in cui le misure continuino ad arrivare correttamente da eKEEPER2, ad esso è associato il *listener* riportato nel Listing 65:

Listing 65: DispositiviActivity, pulsante misure 24 ore

```

1 btnGraph.setOnClickListener(new View.OnClickListener() {
2     @Override
3     public void onClick(View v) {
4
5         download24Temp();
6         download24Umid();
7
8         Intent intentGraph=new Intent(getApplicationContext(),GraphActivity.class);
9         intentGraph.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
10        startActivity(intentGraph);
11    }
12 });

```

Il tocco di questo elemento richiama due procedure asincrone per il download dal server delle misure registrate nelle ultime 24 ore da eKEEPER2 (righe 5-6) e successivamente lancia l'activity dedicata alla visualizzazione a schermo dei dati scaricati (Fig. 28):

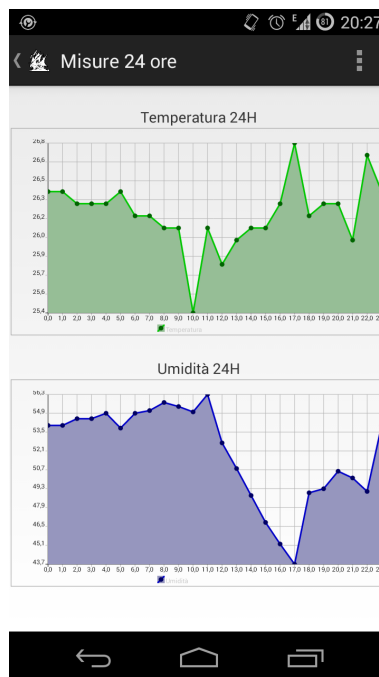


Figura 28: GraphActivity, grafici di temperatura ed umidità di 24 ore

Listing 66: GraphActivity, generazione dei grafici

```

1 database24 = new Database240re(this);
2 tempGiornata = database24.readTemp240re(1);
3
4 seriesTemp[0]=Double.valueOf(tempGiornata.getTemp1());
5 ...
6
7 XYSeries series1 = new SimpleXYSeries(Arrays.asList(seriesTemp),
8     SimpleXYSeries.ArrayFormat.Y_VALS_ONLY, "Temperatura");
9 TemperaturePlot.addSeries(series1, seriesTemp);

```

La generazione dei grafici avviene nella modalità riportata in Listing 66, estraendo in primo luogo i dati conservati nei database dell'applicazione (righe 1-2), i quali vengono inseriti all'in-

terno di un array di 24 elementi, uno per ogni ora (riga 4). Successivamente viene generata una serie numerica conforme a quello che richiede l'oggetto grafico, convertendo l'array in una lista, indicando che i valori contenuti sono riferiti all'asse *Y* ed associando a tali dati una didascalia (riga 7). Infine si inserisce all'interno del grafico la serie numerica (riga 9).

Infine, gli ultimi elementi di `DispositiviActivity` sono i due pulsanti riservati al cambio di funzionamento di `eKEEPER2`:

- *Abilita Protezione*: tramite questo tasto viene inoltrata la richiesta ad `eKEEPER2` di entrare in funzionamento di protezione, ovvero si abilita l'invio di notifiche via squillo ed SMS al cellulare del cliente ogni qualvolta uno dei punti sensibili monitorati (si veda sezione 4.4) risulta in stato di allarme.

Listing 67: `DispositiviActivity`, abilitazione protezione

```

1 protetto.setOnClickListener(new View.OnClickListener() {
2     public void onClick(View v) {
3         if(!(databaseStatoNatante.readStatoNatante(1).getFunzionamento()).equals("2"))
4         {
5             protetto.setBackgroundColor(Color.parseColor("#607D8B"));
6             initStateNatante.setFunzionamento("2");
7
8             if(databaseStatoNatante.readStatoNatante(1).getOnline().equals("0"))
9             {
10                sendSMS(numeroeKEEPER, "COD08:1");
11            }
12            else
13            {
14                modificaFunzionamento("funzionamento", "2");
15            }
16        }
17        else
18        {
19            protetto.setBackgroundColor(Color.parseColor("#607D8B"));
20            initStateNatante.setFunzionamento("1");
21
22            if(databaseStatoNatante.readStatoNatante(1).getOnline().equals("0"))
23            {
24                sendSMS(numeroeKEEPER, "COD08:0");
25            }
26            else
27            {
28                modificaFunzionamento("funzionamento", "1");
29            }
30        }
31        databaseStatoNatante.updateStatoNatante(initStatoNatante);
32    }
33 });

```

L'invio della richiesta avviene come indicato nel Listing 67: dal database dell'applicazione si preleva il valore che indica il funzionamento attuale di `eKEEPER2` (righe 3 e 17) quindi, nel caso in cui il sistema non sia protetto, si cambia il colore del pulsante rendendolo verde e notificando che il sistema è stato messo in protezione (righe 5-6). Ora l'applicazione verifica la via di comunicazione adatta per inviare la richiesta ad `eKEEPER2` (riga 8), quindi invia un SMS con un codice identificativo oppure invia le informazioni al server

tramite lo script PHP *uploadStato.php* (righe 10-14).

Se il funzionamento di allarme fosse invece già attivo, il click sul pulsante riporta il funzionamento in libero, colorando il pulsante di grigio (righe 19-20). La comunicazione avviene nella stessa modalità appena descritta, ovviamente cambiando i parametri inviati (righe 24-28).

Infine, le modifiche vengono salvate all'interno del database (riga 31).

- *Abilita Tracking*: il funzionamento di questo pulsante è analogo a quello di *Abilita Protezione*, dove l'unica differenza sta nell'informazione inviata. Nel Listing 68 sono riportate le istruzioni che cambiano rispetto al caso precedente

Listing 68: DispositiviActivity, abilitazione del tracciamento

```
1 if(databaseStatoNatante.readStatoNatante(1).getOnline().equals("0"))
2 {
3     sendSMS(numeroeKEEPER,"COD07:1");
4 }
5 else
6 {
7     modificaFunzionamento("trackON", "1");
8 }
```

4.6 MappaActivity, visualizzazione dei viaggi effettuati

L'ultima tab all'interno di *TabularActivity* è dedicata alla sezione viaggi registrati da eKEEPER2. L'activity in questione dedica la quasi totalità della schermata all'oggetto mappa, ponendo sopra ad esso un semplice menù per selezionare il viaggio da disegnare (Fig. 29).

Nel momento della creazione dell'activity viene controllata la dimensione del database contenente i viaggi memorizzati in applicazione e, qualora risultasse nulla, il pulsante *Visualizza* viene disabilitato per evitare *nullPointerException*.

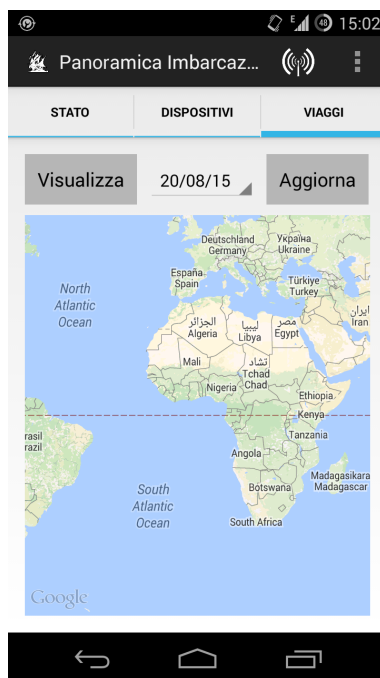


Figura 29: MappaActivity, schermata generale

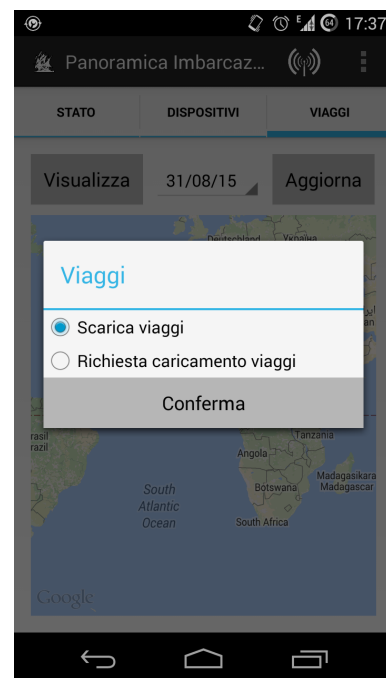


Figura 30: MappaActivity, comunicazione con il server

Se il database contiene dei viaggi allora le date che identificano gli stessi vengono inserite all'interno di un menù a tendina, come mostrato nel Listing 69

Listing 69: MappaActivity, creazione del menù a tendina

```
1 listViaggi = databaseViaggi.getAllViaggi();
2 List<String> listData = new ArrayList<String>();
3
4 selezionaViaggio = (Spinner) findViewById(R.id.selezionaViaggi);
5 List<String> list = new ArrayList<String>();
6
7 for (int i = 0; i < listViaggi.size(); i++)
8 {
9     listData.add(i, listViaggi.get(i).getData());
10    list.add(listViaggi.get(i).getData());
11 }
12
13 ArrayAdapter<String> dataAdapter = new ArrayAdapter<String>(this,
14     android.R.layout.simple_spinner_item, list);
15 dataAdapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
16 selezionaViaggio.setAdapter(dataAdapter);
```

Le date dei viaggi vengono prelevate dal database sotto forma di lista concatenata (righe 1-2), quindi tramite un ciclo *for*, ogni elemento della lista viene inserito all'interno della lista che costituisce il menù a tendina (righe 7-10). Infine si imposta l'attributo che determina la direzione di apertura del menù a tendina, ovvero verso il basso (righe 13-15).

Per aggiungere i viaggi ai database dell'applicazione si utilizza il pulsante *Aggiorna* il quale produce la finestra di dialogo mostrata in Fig. 30.

Nel Listing 70 sono riportate le istruzioni eseguite per mostrare la finestra di dialogo (riga 1-3) e gestire gli input che riceve (riga 6).

Listing 70: MappaActivity, azioni di aggiornamento

```
1 final Dialog dialog = new Dialog(MappaActivity.this);
2 dialog setContentView(R.layout.viaggi_server);
3 dialog.setTitle(R.string.tabViaggi);
4
5 radioRichiestaGroup=(RadioGroup)dialog.findViewById(R.id.radioRichiesta);
6 btnConferma.setOnClickListener(new View.OnClickListener()
7 {
8     public void onClick(View v)
9     {
10         int selectedId = radioRichiestaGroup.getCheckedRadioButtonId();
11         radioRichiestaButton = (RadioButton)dialog.findViewById(selectedId);
12         String radioId = (String) radioRichiestaButton.getText();
13
14         if(radioId.equals(getResources().getString(R.string.radio_Server)))
15         {
16             scaricaViaggi();
17         }
18         else
19         {
20             richiestaViaggi();
21         }
22         dialog.dismiss();
```

```

23     }
24 });
25 dialog.show();

```

Quando l'evento di click sul pulsante *Conferma* viene rilevato l'activity provvede a rilevare quale delle due opzioni è stata selezionata e quindi esegue l'istruzione ad essa collegata (righe 10-14).

Le due differenti azioni che si possono eseguire sono le seguenti:

- *Scarica viaggi*: selezionando questo elemento della lista il cliente richiede all'applicazione di connettersi al server ricercando nella tabella ad egli dedicata i viaggi non ancora scaricati nell'applicazione (riga 16).
- *Richiesta caricamento viaggi*: l'azione associata a questa opzione consiste nell'inviare tramite il server un bit di richiesta ad eKEEPER2 (riga 20), il quale dovrà inviare ai database online le stringhe contenenti le coordinate di viaggio registrate.

Delle due azioni appena elencate solamente la prima apporta degli effettivi cambiamenti nell'applicazione, inserendo nuove informazioni all'interno del database dedicato ai viaggi ed aggiungendo i nuovi elementi nel menù a tendina che permette di selezionare la data del viaggio. Dopo aver selezionato il viaggio tramite la data presente nel menù a tendina il cliente preme su *Visualizza*, eseguendo il codice riportato in Listing 71:

Listing 71: MappaActivity, creazione del viaggio sulla mappa

```

1 listViaggi = databaseViaggi.getAllViaggi();
2 int idRiga = listViaggi.get(selezionaViaggio.getSelectedItemPosition()).getId();
3 coordViaggio = databaseViaggi.readViaggio(idRiga).getCoordinate();
4
5 PolylineOptions rectOptions = new PolylineOptions();
6 mMap.clear();
7
8 while(i<coordViaggio.length())
9 {
10     if(!coordViaggio.substring(i, i+1).equals("!"))
11         sottoStringa += coordViaggio.substring(i, i+1);
12     else if(coordViaggio.substring(i, i+1).equals("!"))
13     {
14         for(int k=0; k<sottoStringa.length();k++)
15         {
16             lat_lon = sottoStringa.split(":");
17             lat_lon_divise = lat_lon[1].split("-");
18
19             if(inizio==1)
20             {
21                 inizio=0;
22
23                 LatLng posIniz = new LatLng(Double.valueOf(lat_lon_divise[1]),
24                     Double.valueOf(lat_lon_divise[0]));
25                 mMap.addMarker(new MarkerOptions().position(posIniz).title("Inizio"));
26                 mMap.moveCamera(CameraUpdateFactory.newLatLng(posIniz));
27                 mMap.animateCamera(CameraUpdateFactory.zoomTo(15));
28
29                 lat_lon_prec[0] = lat_lon_divise[0];
30                 lat_lon_prec[1] = lat_lon_divise[1];

```

```

30     }
31
32     Polyline polyline = mMap.addPolyline(new PolylineOptions().add(new
        LatLng(Double.valueOf(lat_lon_prec[1]),
        Double.valueOf(lat_lon_prec[0])), new
        LatLng(Double.valueOf(lat_lon_divise[1]),Double.valueOf(lat_lon_divise[0])))
33     .width(5).color(Color.RED));
34
35     sottoStringa="";
36     lat_lon_prec[0] = lat_lon_divise[0];
37     lat_lon_prec[1] = lat_lon_divise[1];
38     }
39 }
40 i++;
41 }
42 }

```

Dalla lista dei viaggi presenti nel database l'activity preleva quello corrispondente al valore dell'*id* associato alla data selezionata (righe 1-3), quindi inizializza e ripulisce la mappa da possibili viaggi caricati precedentemente (righe 5-6).

Successivamente procede ad estrarre le coordinate dalla stringa ottenuta dal database, la quale è formata nel seguente modo:

```

1 ...!gg/mm/aaaa:lon-lat!gg/mm/aaaa:lon-lat!gg/mm/aaaa:lon-lat...

```

Il ciclo *while* a riga 8 serve per eseguire la scansione dell'intera stringa caricata, sulla quale si estrapolano le informazioni relative alla singola coordinata. La sottostringa di interesse è racchiusa tra due caratteri "!", e la si individua tramite le istruzioni alle righe 10-12. Una volta ottenuta una singola sottostringa del tipo *gg/mm/aaaa : lon – lat*, si provvede ad estrarne le singole informazioni di latitudine e longitudine (righe 16-17) per inserirle nella mappa.

Per individuare il punto di inizio del viaggio, la prima coordinata estratta viene associata ad un *marker* intitolato *Inizio* (riga 24): la condizione che individua la prima coordinata è individuata dal bit *inizio* = 1, il quale viene subito settato a 0 dopo aver impostato la prima coordinata.

Il punto viene quindi inserito nella mappa (righe 23-26) e ad esso viene associata una variabile temporanea: l'utilizzo di quest'ultima è fondamentale per creare la linea che collega le diverse coordinate registrate, poichè l'oggetto *Polyline* richiede le coordinate dei due punti che individuano gli estremi del segmento che li collega (righe 32-33).

In Fig. 31 è riportato l'aspetto di MappaActivity dopo il caricamento di un viaggio.

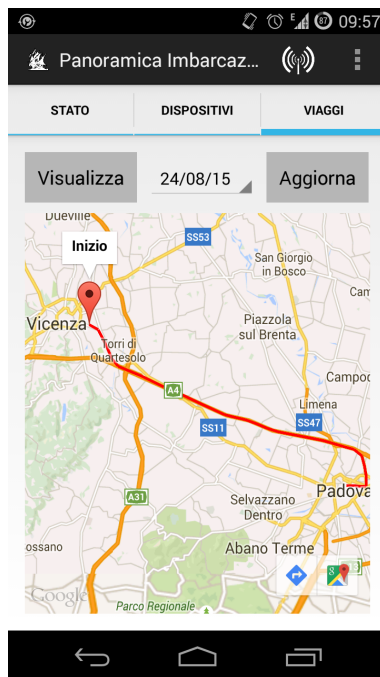


Figura 31: MappaActivity, visualizzazione di un viaggio

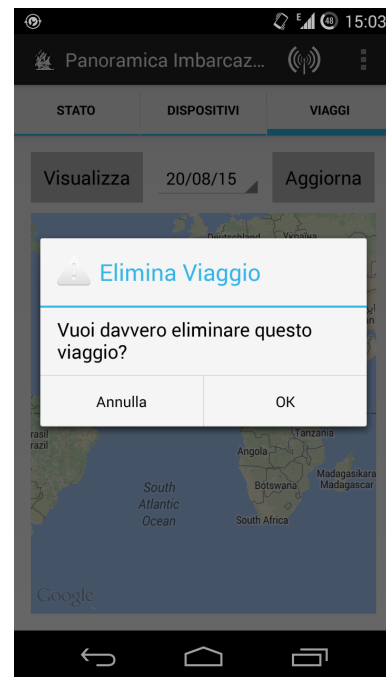


Figura 32: MappaActivity, eliminazione di un viaggio

In Fig. 32 è mostrata la procedura di eliminazione dalla memoria di un viaggio: una volta selezionata la data del viaggio da cancellare, l'utente deve cliccare a lungo sull'elemento così da generare la finestra di dialogo mostrata in figura.

Listing 72: MappaActivity, eliminazione di un viaggio

```

1 selezionaViaggio.setOnLongClickListener(new OnLongClickListener()
2 {
3     @Override
4     public boolean onLongClick(View v)
5     {
6         confirmDialog = new AlertDialog.Builder(MappaActivity.this)
7             .setPositiveButton(android.R.string.yes, new
8                 DialogInterface.OnClickListener() {
9                     public void onClick(DialogInterface dialog, int whichButton)
10                    {
11                        listViaggi = databaseViaggi.getAllViaggi();
12                        int idRiga =
13                            listViaggi.get(selezionaViaggio.getSelectedItemPosition()).getId();
14                        databaseViaggi.deleteViaggio(idRiga);
15
16                        Context context = getBaseContext();
17                        Intent finale=new Intent(context,TabularActivity.class);
18                        startActivity(finale);
19                    }
20                }).setNegativeButton(android.R.string.no, null).show();
21    return true;
22    }
23 });

```

Il Listing 72 mostra la procedura di rilevamento della pressione prolungata e successiva eliminazione del viaggio: quando il *listener* rileva l'evento (riga 1) crea la finestra di dialogo

(riga 6), quindi vengono specificate le azioni da svolgere in caso di conferma (riga 7) o di cancellazione (riga 21).

Nel primo caso l'activity trova nel database viaggi l'*id* corrispondente alla data selezionata e lo utilizza per eliminare la riga contenente tale viaggio (righe 10-12). Infine ricarica l'activity in modo da apportare i dovuti cambiamenti all'interfaccia grafica (righe 14-16).

Nel secondo caso la finestra di dialogo viene chiusa senza apportare alcuna modifica ai database (riga 18).

4.7 Impostazioni: gestione dell'account e dei numeri

Dopo aver descritto le tre principali activity che può utilizzare l'utente per comunicare da remoto con eKEEPER2 e scambiare dati, si approfondisce la parte dell'applicazione relativa alle impostazioni dell'utente. In Fig. 33 è mostrato il collocamento e il contenuto del menù delle impostazioni:

- *Modifica Profilo* serve per apportare modifiche all'account creato durante la prima inizializzazione dell'applicazione
- *Gestione Numeri* serve per gestire i numeri abilitati ed allertarli alla rubrica

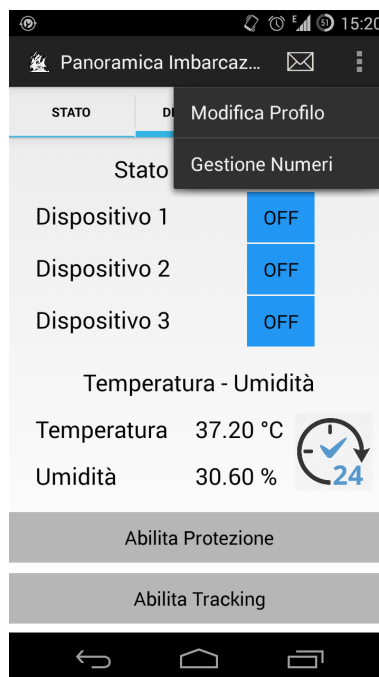


Figura 33: Menù delle impostazioni utente

Selezionando la voce *Modifica Profilo* viene eseguita un'activity graficamente identica a quella utilizzata per la creazione dell'account (Fig. 18) tuttavia, poichè i database dell'applicazione contengono già queste informazioni, alcune di esse vengono già inserite nei corrispettivi campi di testo (ad esempio l'username nella prima cella). Questa activity è concepita per permettere al cliente di cambiare le proprie chiavi di sicurezza per accedere all'applicazione, oppure di inserire un nuovo numero SIM di eKEEPER2 nel caso di un cambio di operatore telefonico o, infine, un nuovo numero seriale in caso di sostituzione dello strumento.

Il funzionamento dell'activity è del tutto analogo a quanto già detto nei Listing 48, 49, 50, 51 e 52.

L'altra voce del menù, *Gestione Numeri*, rimanda all'activity mostrata in Fig. 34, dove è possibile vedere la rubrica di numeri abilitati ed allertabili che l'utente imposta per eKEEPER2.



Figura 34: GestioneActivity, lista dei numeri memorizzati

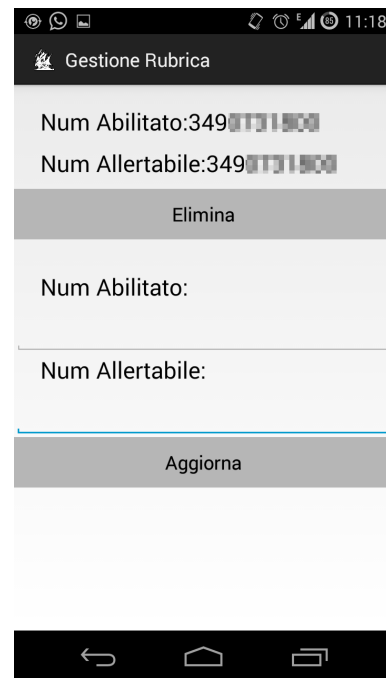


Figura 35: NumeriActivity, inserimento di nuovi numeri abilitati ed allertabili

Il funzionamento di questa activity è riportato nel Listing 73:

Listing 73: GestioneActivity inizializzazione della schermata

```

1  if(db.dbSize()==0)
2  {
3      for(int k=0;k<5;k++)
4      {
5          db.createNumeri(new Numeri("", ""));
6      }
7  }
8
9  list = db.getAllNumeri();
10 List<String> listNumeri = new ArrayList<String>();
11
12 for (int i = 0; i < list.size(); i++)
13 {
14     listNumeri.add(i, list.get(i).abilToString());
15 }
16 for (int i = 0; i < list.size(); i++)
17 {
18     listNumeri.add(i, list.get(i).alertToString());
19 }
20
21 adapterNumeri = new ArrayAdapter<String>(this, R.layout.row_layout, R.id.listAbil,
22     listNumeri);
23 listView().setOnClickListener(this);
24 setListAdapter(adapterNumeri);

```

Onde evitare problemi di *nullPointerException* l'activity controlla la dimensione del database dedicato ai numeri e, in caso risultasse nullo, provvede ad inizializzare la rubrica con stringhe vuote (righe 1-5).

Successivamente preleva dal database una lista concatenata contenente tutti i numeri, sia allertabili che abilitati (righe 14 e 18), i quali verranno inseriti in un oggetto *adapter* il quale si presta a contenere la lista dei numeri (righe 21-23).

Da notare che, sempre in Fig. 34, il pulsante *Conferma* appare in trasparenza in quanto non è abilitato a causa dell'assenza di numeri da inviare ad eKEEPER2.

Selezionando uno slot di memoria si passa all'activity mostrata in Fig. 35, la quale è dedicata all'inserimento dei numeri, indicando quali sono abilitati, quali allertabili e quali entrambi.

Listing 74: NumeriActivity, inserimento dei numeri allertabili ed abilitati

```
1 public void initializeViews()
2 {
3     numAbil.setText(selectedNumeri.getAbil());
4     numAlert.setText(selectedNumeri.getAlert());
5 }
6
7 public void update(View v)
8 {
9     db.updateNumeri(selectedNumeri);
10 }
11
12 public void delete(View v)
13 {
14     db.deleteNumeri(selectedNumeri);
15 }
```

Facendo riferimento al codice contenuto nel Listing 74, se la cella selezionata contiene un numero, esso viene riportato sulla parte superiore della schermata tramite il metodo *initializeViews* (righe 1-4).

Le azioni collegate ai due pulsanti presenti sono collegate ai metodi *update* e *delete* rispettivamente per aggiungere (o aggiornare) un numero della rubrica e per cancellarlo. Tali metodi richiamano direttamente le istruzioni della classe *DatabaseAdapter* (di cui si è discusso in sezione 4), inserendo come parametro la cella di memoria in cui devono essere inseriti i numeri. Quindi, una volta inseriti i numeri desiderati negli appositi campi di testo, premendo il pulsante *Aggiorna* questi vengono memorizzati nel database e l'applicazione torna su *GestioneActivity*. Al contrario, una volta selezionata la cella contenente un numero, il pulsante *Elimina* rimuove il dato dal database.

Ritornando a *GestioneActivity* i numeri vengono visualizzati sulla lista: ora viene attivato il pulsante *Conferma* di Fig. 34 perchè esiste almeno un numero da trasmettere ad eKEEPER2. La procedura di invio dei numeri è riportata in Listing 75:

Listing 75: GestioneActivity, invio dei numeri ad eKEEPER2

```
1 String aggiornaAbil="COD02";
2 String aggiornaAlert="COD03";
3 String aggiornaAbilServer="";
4 String aggiornaAlertServer="";
5
6 list = db.getAllNumeri();
7
8 for (int i = 0; i < list.size(); i++)
```

```

9 {
10     aggiornaAbil += ":";
11     aggiornaAbil+=list.get(i).getAbil();
12     aggiornaAbilServer += list.get(i).getAbil()+":";
13 }
14 for (int i = 0; i < list.size(); i++)
15 {
16     aggiornaAlert+=":";
17     aggiornaAlert+=list.get(i).getAlert();
18     aggiornaAlertServer+=list.get(i).getAlert()+":";
19 }
20 db.close();
21
22 databaseStatoNatante = new DatabaseStatoNatante(getApplicationContext());
23
24 if(databaseStatoNatante.readStatoNatante(1).getOnline().equals("0"))
25 {
26     sendSMS(numeroArduino, aggiornaAbil);
27     sendSMS(numeroArduino, aggiornaAlert);
28 }
29 else
30 {
31     aggiornaNumeri(aggiornaAbilServer+aggiornaAlertServer);
32 }
33 databaseStatoNatante.close();

```

In questo caso i dati inviati dall'applicazione ad eKEEPER2, sia via SMS che via server, consistono di una stringa contenente i numeri abilitati ed allertabili separati tra di loro dal carattere `:`. La creazione di queste stringhe inizia dalla dichiarazione delle relative variabili (righe 1-4), dopodichè a tali stringhe vengono concatenati i numeri inseriti in database abilitati (righe 8-12) e allertabili (righe 14-18).

Infine, come già fatto in altri casi, viene controllata la presenza di eKEEPER2 sul server e, in caso positivo, l'applicazione effettua la connessione al server tramite un task asincrono (riga 31) e fornisce i propri dati allo script *uploadNumeri.php* (sezione 3.3.3), mentre in caso negativo invia un SMS (righe 26-27).

4.8 SMSActivity, la gestione degli SMS

L'activity che gestisce la ricezione e l'invio degli SMS è leggermente diversa da quelle descritte finora: essa non è presente visivamente all'interno dell'applicazione ma è un componente in ascolto di determinati messaggi, chiamati Intents Broadcast.

Gli Intents Broadcast sono un particolare tipo di Intent che vengono spediti attraverso il metodo *sendBroadcast*.

Vengono utilizzati per notificare determinati eventi alle applicazioni del sistema che sono in ascolto in modo che possano reagire.

Il metodo principale di questa activity è *onReceive*, il quale si attiva nel caso in cui venga rilevato un evento specifico, ovvero la ricezione di un SMS. Nel Listing 76 è mostrato come viene ricevuto un SMS e la successiva estrazione dei dati contenuti

Listing 76: SMSReceiver, ricezione di un messaggio

```

1 Bundle bundle = intent.getExtras();

```

```

2

```

```

3 if (bundle != null)
4 {
5     Object[] pdus = (Object[]) bundle.get("pdus");
6     for (int i = 0; i < pdus.length; i++)
7     {
8         recMsg = SmsMessage.createFromPdu((byte[]) pdus[i]);
9         try
10             data = recMsg.getUserData();
11         catch (Exception e)
12             { }
13         if (data != null)
14             recMsgString = recMsg.getMessageBody().toString();
15
16         fromAddress = recMsg.getOriginatingAddress();
17     }
18 }

```

Prima di tutto, una volta rilevato l'evento di ricezione SMS, il *receiver* ne estrae i contenuti (riga 1): questa azione produce un oggetto *Bundle* il quale contiene i pacchetti dell'SMS. Appurato che il contenuto non sia nullo (riga 3) si procede alla conversione da *Bundle* ad array di oggetti (riga 5), ed infine a stringa (righe 6-14). Alla riga 16 viene estratto il numero del mittente il quale verrà utilizzato successivamente per un confronto con il numero SIM di eKEEPER2 (Listing 77).

Listing 77: SMSReceiver, estrazione delle informazioni

```

1 if(fromAddress.equals(numeroeKEEPER))
2 {
3     String[] dati = { /*Array di stringhe vuote*/ };
4     dati = recMsgString.split(":");
5
6     if(dati[0].equals("COD09")) //Stato imbarcazione
7         {...}
8     else if(dati[0].equals("COD05")) //ON/OFF dispositivi
9         {...}
10    else if(dati[0].equals("COD07")) //tracking
11        {...}
12    else if(dati[0].equals("COD08")) //Protetto
13        {...}
14    else if(dati[0].equals("Allarmi riscontrati"))
15        {...}
16 }

```

Il numero del mittente precedentemente estratto viene confrontato con quello di eKEEPER2 (riga 1) e, in caso di riscontro positivo, si procede all'estrazione del contenuto: la stringa ottenuta alla riga 14 del Listing 76 viene suddivisa, inserendo ogni sottostringa individuata dal carattere `:` in una cella di un array. Ora viene confrontato il contenuto della prima cella di tale array (ovvero la prima parte del contenuto dell'SMS) con dei codici prefissati (si veda [16]). A seconda del codice individuato l'applicazione intraprende azioni differenti:

- *COD09*: il codice indica che l'SMS contiene lo stato globale dell'imbarcazione, quindi le rimanenti stringhe dell'array *dati* vengono inserite all'interno dell'apposito database
- *COD05*: l'SMS contiene la conferma che il dispositivo indicato dall'applicazione ha cambiato correttamente lo stato, quindi viene aggiornato il relativo database

- *COD07* e *COD08*: l'SMS conferma che eKEEPER2 ha correttamente cambiato il proprio funzionamento relativo al tracking o alla protezione
- *Allarmi riscontrati*: in caso di segnalazione di allarme eKEEPER2 effettua una segnalazione via chiamata e successivamente via SMS, il cui testo iniziale è appunto Allarmi riscontrati. Tale messaggio viene notificato dall'applicazione facendo comparire tra le notifiche una segnalazione (si veda Fig. 37) il cui click rimanda direttamente all'applicazione. La notifica viene generata come mostrato nel Listing 78

Listing 78: SMSReceiver, generazione notifica di allarme

```

1 private void showNotification(Context context)
2 {
3     PendingIntent contentIntent = PendingIntent.getActivity(context, 0, new
4         Intent(context, SMSReceiver.class), 0);
5
6     NotificationCompat.Builder mBuilder = new
7         NotificationCompat.Builder(context)
8         .setSmallIcon(R.drawable.logo)
9         .setContentTitle("Allarme riscontrato")
10        .setContentText("Accedi per verificare");
11    mBuilder.setContentIntent(contentIntent);
12    mBuilder.setDefaults(Notification.DEFAULT_SOUND);
13    mBuilder.setAutoCancel(true);
14
15    Intent notificationIntent = new Intent(context, TabularActivity.class);
16
17    TaskStackBuilder stackBuilder = TaskStackBuilder.create(context);
18    stackBuilder.addParentStack(TabularActivity.class);
19    stackBuilder.addNextIntent(notificationIntent);
20    PendingIntent resultPendingIntent =
21        stackBuilder.getPendingIntent(0, PendingIntent.FLAG_UPDATE_CURRENT);
22    mBuilder.setContentIntent(resultPendingIntent);
23    NotificationManager mNotificationManager = (NotificationManager)
24        context.getSystemService(Context.NOTIFICATION_SERVICE);
25    mNotificationManager.notify(1, mBuilder.build());
26 }

```

SPIEGARE IL CODICE

L'activity SMSReceiver ha un'ulteriore funzione: partendo dal presupposto che, se la comunicazione tra app ed eKEEPER2 avviene tramite SMS, allora quest'ultimo per diversi motivi non è in grado di connettersi al server. Questo scenario fa sì che le informazioni presenti online possano diventare obsolete o non rispecchino l'effettivo stato dell'imbarcazione. Per evitare questa situazione, qualora i dati vengano scambiati via SMS, sarà l'applicazione a fornire una copia delle informazioni appena ricevute al server, utilizzando l'apposito script PHP *uploadStatoApp.php* (trattato in sezione 3.3.2).

Come appena descritto, gli SMS ricevuti da eKEEPER2 vengono gestiti dall'applicazione, tuttavia questo non esclude che possano essere visualizzati tramite l'applicazione per messaggi presente di default in ogni smartphone: questo rende possibile l'utilizzo di eKEEPER2 anche solo via SMS, sebbene con funzioni limitate. In Fig. 36 è riportato un esempio di conversazione tra utente ed eKEEPER2 esclusivamente via SMS.



Figura 36: Esempio di scambio SMS tra eKEEPER2 e lo smartphone

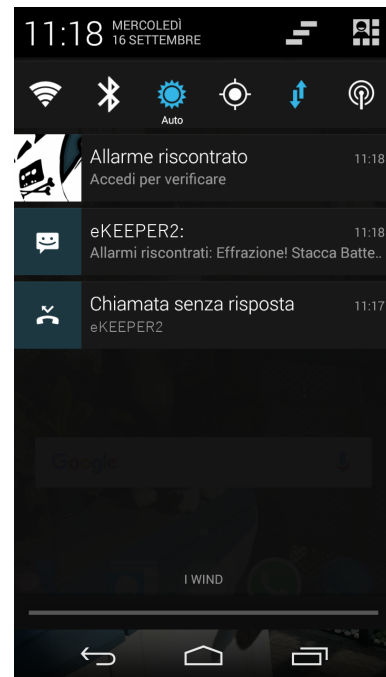


Figura 37: Notifica di allarme via SMS

4.9 Analisi consumo dati

Viene proposta infine una breve analisi del consumo dati da parte dell'applicazione. Come è stato esposto, la maggior parte delle informazioni scambiate tra applicazione e server avvengono in seguito alla pressione di determinati tasti, mentre esiste una routine che è sempre attiva quando l'applicazione stessa è attiva: come scritto in sezione 4.3, l'applicazione si connette con cadenza di un minuto al server per verificare che il bit *eKEEPEROnline* sia 1.

La connessione al server avviene per via di un *three – way handshake TCP* dove avvengono le seguenti sequenze:

- Il Device 1 invia il proprio numero di sequenza *TCP* e la dimensione massima del segmento al Device 2
- Il Device 2 risponde inviando il proprio numero di sequenza e la dimensione massima del segmento al Device 1
- Il Device 1 notifica la ricezione del numero di sequenza e della dimensione del segmento

Ogni pacchetto è composto di un header *IP* e dal *payload*. In questo caso, la sezione dati contiene il *TCP*. L'header *TCP* contiene diversi campi incluse le porte di invio e destinazione, le sequenze e i numeri di acknowledgment, dimensione della finestra, flag *TCP*, puntatori urgenti e bit riservati.

Come per l'header *IP*, anche l'header *TCP* può contenere delle opzioni, e poichè le opzioni del *TCP* cambiano la lunghezza dell'header *TCP*, la lunghezza è dichiarata nell'header.

L'header IPv4 è composto da cinque segmenti di 4 bytes, oppure da 20 bytes in totale.

Generalmente *TCP* utilizza 24 bytes dell'header per l'handshake e 20 per la trasmissione di un pacchetto normale. La disconnessione dal server, invece, richiede la trasmissione di 4 pacchetti: all'interno del primo frame il client invia un *FIN* che è seguito da un bit di *ACK*. Il parametro *FIN* informerà il server che non ci sono più dati da inviare.

La risposta consiste in un *ACK* da parte del server che notifica la ricezione del *FIN* del client.

A sua volta il server invia un *FIN* al cliente, il quale a sua volta replica tramite un *ACK*. Inoltre, l'offset di ogni frame inviato è tipicamente di 20 bytes. Sommando il totale si giunge ai seguenti risultati:

- 128-136 bytes per stabilire la connessione
- 1 bytes per il download della risposta del server (1 carattere)
- 160 bytes per disconnettersi dal server

Se l'applicazione fosse attiva (in primo piano o in background) 24 ore al giorno per tutto il mese il consumo dovuto alla sola connessione ciclica al server è commensurabile in 12.8 MB.

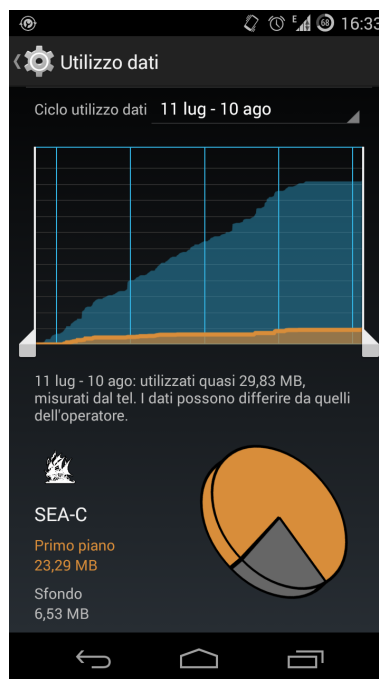


Figura 38: Consumo dati mensile applicazione

Oltre a questa operazione schedulata nel tempo (e quindi calcolabile), l'utente può effettuare altre richieste a proprio piacimento, andando ad aggiungere altri dati al traffico consumato dall'applicazione. I bytes scambiati per effettuare le diverse operazioni (intesi per il solo payload) sono riassunti in tabella 13

Operazione	Quantità dati
Aggiornamento stato	92-100 bytes
Attivazione di un dispositivo	5 bytes
Temperatura ed umidità 24 ore	144 bytes
Impostazione protezione sistema	5 bytes
Impostazione tracciamento	5 bytes

Tabella 13: Consumo di traffico dati per operazione eseguita

A questi possibili payload si aggiungono i dati ben più onerosi e più difficilmente quantificabili scaricati dall'applicazione per visualizzare le mappe Google. In questo caso si è cercato di limitare il consumo massimo di dati predisponendo la visualizzazione di mappe schematiche piuttosto che la visualizzazione che utilizza le immagini acquisite dal satellite.

Facendo riferimento alla Fig. 38, il consumo dati dell'applicazione durante un mese di test è di circa 30 MBytes, e si può concludere che l'applicazione non rischia di consumare una porzione elevata del traffico dati sullo smartphone del cliente.

5 Sviluppi futuri

In conclusione vengono presentati gli sviluppi futuri di tale progetto:

- Applicazione iOS: la creazione di un'app per sistemi operativi Android sicuramente non copre la totalità dei possibili clienti interessati ad eKEEPER2. Per questo motivo si è deciso di replicare, nell'immediato futuro, la stessa applicazione qui presentata per sistemi operativi Apple.
- Ampliamento servizi assistenza: ad oggi Aitech s.r.l. ha la possibilità di accedere alle diverse tabelle presenti nel server in modo da poter fornire assistenza ai clienti in caso di malfunzionamenti o di guasti. Verranno quindi sviluppati dei tools grafici i quali estrapoleranno determinate informazioni dai database e li presenteranno sotto forma di grafici temporali, fornendo così una storia dell'imbarcazione. I campi di maggiore interesse riguardano le batterie servizi, motore e tampone, le quali indicano lo stato di funzionamento dell'impianto elettrico complessivo della barca e di eKEEPER2.
- Replica delle funzionalità dell'applicazione su portale web: in linea con quanto operato da molti produttori di applicazioni (ad es. Facebook, Whatsapp,...), si vuole introdurre la possibilità di monitorare l'imbarcazione ed effettuare le operazioni descritte nelle sezioni precedenti anche tramite un apposito sito web. In questo modo anche utenti che non possiedono uno smartphone, oppure che non ne sono frequenti utilizzatori, possono facilmente interagire con eKEEPER2
- Adattamento del progetto eKEEPER2 al settore dei camper: il principio è simile a quello dell'imbarcazione, in cui l'utente è interessato a poter sorvegliare il proprio mezzo: anche in questo caso eKEEPER2 viene utilizzato come sistema di monitoraggio dei punti sensibili del camper (batterie, temperatura ambientale, intrusioni,...).

Riferimenti bibliografici

- [1] Mauro Lecce. Guida Android. 2012. URL: <http://www.html.it/guide/guida-android>
- [2] Google. Activities. URL: <http://developer.android.com/guide/components/activities.html>
- [3] Mauro Lecce. Activity. URL: <http://www.html.it/pag/19499/le-attivitaactivity>
- [4] Androidiani Lunard. Capire e programmare le Activity. 2009. URL: <http://www.androidiani.com/applicazioni/sviluppo/capire-e-programmare-le-activity-869>
- [5] AndroidGeek. Activity. URL: <http://www.androidgeek.it/tutorials/programmazione-android/capire-le-activity-ed-il-ciclo-di-vita-di-unapplicazione-android>
- [6] Wikipedia. Android. URL: <http://it.wikipedia.org/wiki/Android>
- [7] Wikipedia. Java (linguaggio di programmazione). URL: http://it.wikipedia.org/wiki/Java_linguaggio_di_programmazione
- [8] Wikipedia. XML. URL: <http://it.wikipedia.org/wiki/XML>
- [9] Satya Komatineni. Understanding R.java. URL: <http://knowledgefolders.com/akc/display?url=displaynoteimpurl&ownerUserId=satya&reportId=2883>
- [10] Google. Android Developers. URL: <http://developer.android.com/design/index.html>
- [11] Mauro Lecce. View. url: <http://www.html.it/pag/19498/la-visualizzazioneview>
- [12] Redazione Io Programmo. Android Programming. Punto Informatico Libri, Edizioni Master, 2011
- [13] Marco Lecce. La gestione dei database in Android. url: <http://www.html.it/articoli/la-gestione-dei-database-in-android-1>
- [14] Elisa Barindelli. Utilizzare le Google Maps API v.2 su Android. URL: <http://android.devapp.it/t022-utilizzare-le-google-maps-api-v-2-su-android-parte-1>
- [15] Karen Kent Frederick. Studying Normal Traffic, Part Three: TCP Headers. URL: <http://www.symantec.com/connect/articles/studying-normal-traffic-part-three-tcp-headers>
- [16] Mirko Perazzolo. Sistema di Monitoraggio da Remoto per Imbarcazioni: Progetto ed Implementazione.